ЧАСТИНА V

Web-технології Java

ГЛАВА 25

Web-інструменти Java

У цьому розділі ми коротко перерахуємо важливі аспекти технології Java, досі не освітлені в книзі, але необхідні для подальшого викладу.

Архіватор *jar*

Для упаковки кількох файлів в один архівний файл, зі стиском або без стиснення, у технології Java розроблено формат архівування JAR. Ім'я архівного jar-файлу може бути будь-яким, але зазвичай воно отримує розширення jar. Спосіб упаковки та стиснення тия заснований на методі ZIP. Назва JAR (Java ARchive) перегукується з назвою відомої утиліти TAR (Tape ARchive), розробленої в UNIX.

Відмінність jar-файлів від zip-файлів тільки в тому, що jar-файли автоматично включають ється каталог META-INF, що містить кілька файлів з інформацією про упаковку. них в архів файли.

Архівні файли дуже зручно використовувати в аплетах, про чим вже говорилося в *гла- ве 18* оскільки весь архів завантажується по мережі відразу ж, одним запитом. Усі файли аплету з байт-кодами, зображеннями, звукові файли упаковуються в один або кілька архівів. Для їх завантаження достатньо в тезі `<applet>` вказати імена архівів в параметрі `archive` , наприклад:

```
<applet code = "MillAnim.class" archive = "first.jar, second.jar"
        width = "100%" height = "100%"></applet>
```

Основний файл MillAnim.class повинен знаходитися в якомусь із архівних файлів first.jar чи second.jar. Інші файли знаходяться в архівних файлах, а якщо не знайдені там, то на сервері, у тому ж каталозі, що й HTML-файл. Втім, файли ап- пліта можна упаковувати не тільки в jar-архів, але і в zip-архів зі стисненням або без стиснення. ня.

Архівні файли зручно використовувати і додатках (applications). Усі файли при- Положення упаковуються в архів, наприклад appl.jar. Додаток виконується прямо з архів, інтерпретатор запускається з параметром `-jar` , наприклад:

```
java -jar appl.jar
```

Ім'я основного класу програми, що містить метод `main()` , вказується у файлі MANIFEST.MF, мова про яке піде трохи пізніше.

При установці JDK на MS Windows автоматично створюється асоціація розширення імені файлу jar з інтерпретатором javaw, яка діє при подвійному клацанні ми- ши по імені файлу, а саме:

```
"C:\jre1.6.0_02\bin\javaw.exe" -jar "%1" %*
```

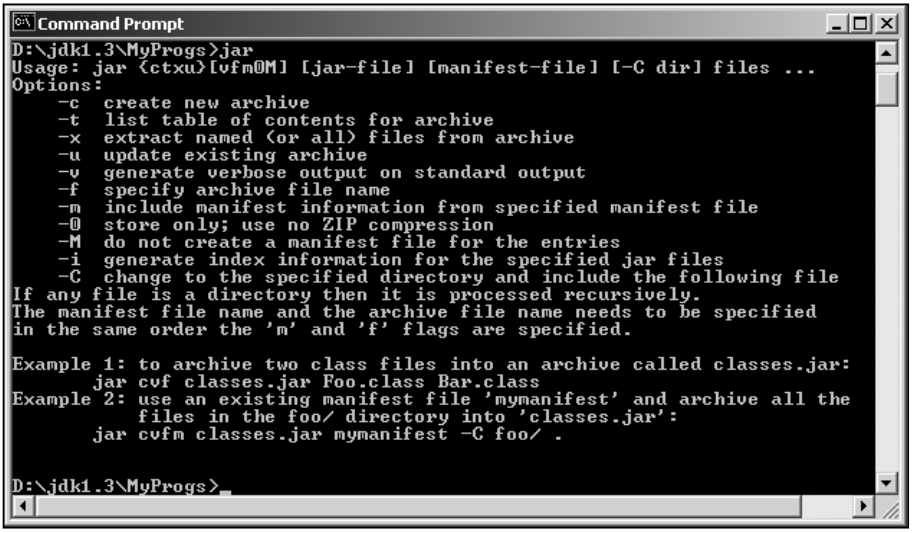
Якщо такий асоціації ні, то її легко створити засобами Windows.

Архівні файли зручні і прості для компактного зберігання всієї необхідної роботи програми інформації. Програма може працювати з файлами архів прямо з архів, не розпаковуючи їх, з допомогою класів пакету `java.util.jar` .

створення архів

Jar-архіви створюються за допомогою класів пакету `java.util.jar` або за допомогою утилі- ти командного рядка `jar` .

Правила використання утиліти `jar` дуже схожі на правила застосування утиліти `tar` . Набравши в командному рядку слово `jar` і натиснувши клавішу <Enter>, ви отримаєте коротке пояснення, подібне до того, що показано на Мал. 25.1.



Мал. 25.1. Правила вживання утиліти `jar`

Параметри утиліти `jar` змінюються від версії до версії, на час написання книги виглядали так:

```
jar {ctxui}[vfm0Me] [jar-file] [manifest-file] [entry-point] [-C dir] files...
```

У цьому рядку зашифровано правила застосування утиліти. Фігурні дужки покази- ють, що після слова `jar` і пробілу треба написати одну з літер: `c` , `t` , `x` , `u` або `i` . Ці літери означають наступні операції:

- `c` (create) - створити новий архів;
- `t` (table of contents) - Направити в стандартний висновок список вмісту архіву;

- `x` (extract) - витягти з архіву один або декілька файлів;
- `u` (update) - оновити архів, замінивши або додавши один або кілька файлів.

Після літери, без пробілу, можна, можливо написати одну або кілька літер, перерахованих вквадратних дужках. Вони означають наступне:

- `v` (verbose) - виводити повідомлення про процесі роботи з архівом в стандартний ви-вод;
- `f` (file) - записаний далі параметр `jar-file` показує ім'я архівного файлу;
- `m` (Маніфест) - записаний далі параметр `manifest-file` показує ім'я файлу опі-санія;
- `0` (нуль) - не стискати файли, записуючи їх в архів;
- `M` (Маніфест) - не створювати файл описи;
- `e` (entry) – використовується при створенні архіву. Записаний далі параметр `entry-point` означає ім'я основного класу, що містить метод `main()` , з якого починається ється виконання програми. Це ім'я буде занесено в створюваний файл MANIFEST.MF (Див. далі).

Параметр `-i` (index) пропонує створити в архіві файл INDEX.LIST. Він використовуєтьсявже після формування архівного файлу.

Після літерних параметрів-файлів через пропуск записується ім'я архівного файлу `jar-file` , потім, через пробіл, ім'я файлу описи `manifest-file` , далі, після пробілу, ім'я основного класу `entry-point` , потім перераховуються імена файлів, які потрібно занести до архіву або витягти з архіву. Якщо це імена каталогів, то операція виконняється рекурсивно зі усіма файлами каталогу.

Перед першим ім'ям каталогу може стояти параметр `-C` . Конструкція `-C dir` означає, що на час виконання утиліти `jar` поточним каталогом стане каталог `dir` .

Необов'язкові параметри занесені в квадратні дужки.

Отже, в кінці командного рядка має бути записано хоча б одне ім'я файлу або каталогу. Якщо серед параметрів є буква `f` , то перший із цих файлів розуміється як архівний jar-файл. Якщо серед параметрів знаходиться буква `m` , то перший файл по- німається як файл опису (manifest-file). Якщо серед параметрів присутні обидві літери, то ім'я архівного файлу та ім'я файлу опису повинні йти в тому ж порядку, що та літери `f` і `m` .

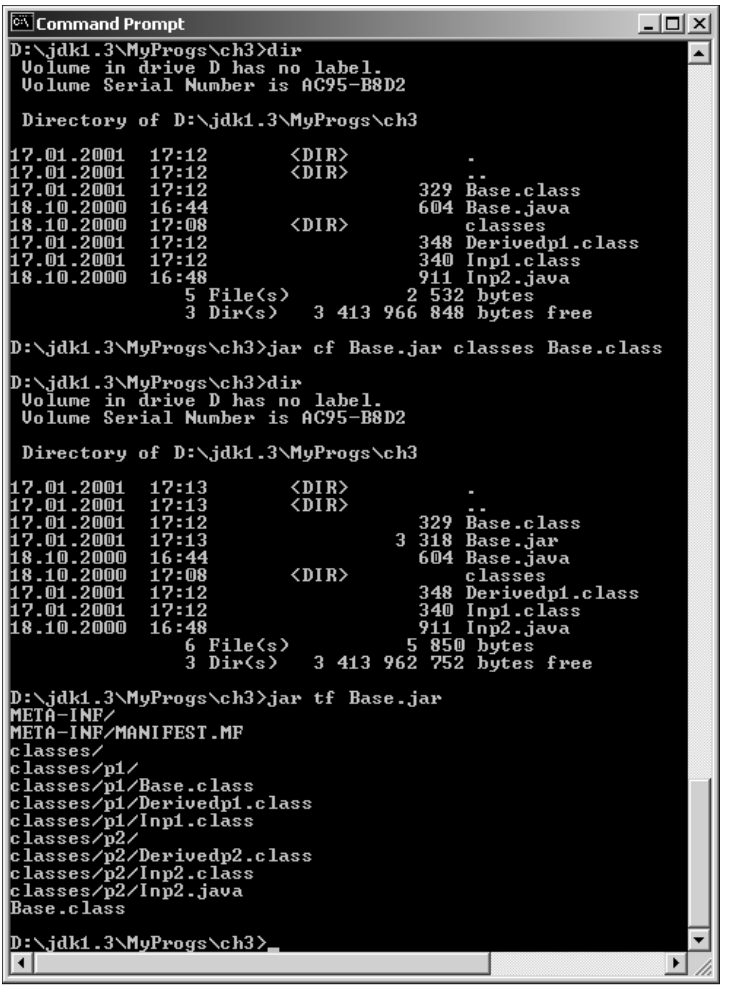
Якщо параметр `f` та ім'я архівного файлу відсутні, то архівним файлом буде слу- жити стандартний висновок.

Якщо параметр `m` і ім'я файлу описи відсутні, то по замовчуванням файл MANIFEST.MF, лежачий в каталозі META-INF архівного файлу, буде утримувати тільки номер версії.

на Мал. 25.2 показаний процес створення архів Base.jar в каталозі ch3.

Спочатку показаний вміст каталогу ch3. Потім створюється архів, який включає- ється файл Base.class і весь вміст підкаталогу classes. Знову виводиться вміст- моє каталогу ch3. У ньому з'являється файл Base.jar. Потім виводиться вміст архів.

Як бачите, в архіві створено каталог META-INF, а в ньому файл MANIFEST.MF.



Мал. 25.2. Робота з утилітою `jar`

Файл описи MANIFEST.MF

Файл MANIFEST.MF, розташований в каталозі META-INF архівного файлу, предназначний для кількох цілей:

- ❒ перерахування файлів з архів, забезпечених цифровий підписом;
- ❒ перерахування компонентів JavaBeans, розташованих в архіві;
- ❒ вказівки імені основного класу для виконання програми з архіву;
- ❒ вказівки імені файлу, містить зображення завантажувального вікна;
- ❒ записи відомостей про версії пакет.

Вся інформація спочатку записується у звичайному текстовому файлі з будь-яким ім'ям, наприклад `manif` . Потім запускається утиліта `jar` , в якій цей файл вказується як значення параметра `m` , наприклад:

```
jar cmf manif Base.jar classes Base.class
```

Утиліта перевіряє правильність записів в файлі `manif` та переносить їх в файл MANIFEST.MF, додаючи свої запис.

Файл опису `manif` має бути написаний за суворими правилами, викладеними у специфікації JAR File Визначення. Її можна, можливо знайти в документації Java SE, в файлі docs/technotes/guides/jar/jar.html.

Наприклад, якщо ми хочемо виконувати додаток з головним файлом Base.class з архів Base.jar, то файл `manif` повинен утримувати як мінімум дві рядки:

```
Main-Class: Base
```

Перший рядок містить відносний шлях до головного класу, але не файлу, тобто без розширення class. У цьому рядку кожен символ має значення навіть пробіл. Друга рядок порожній - файл обов'язково повинен закінчуватися порожнім рядком, точніше гово- ря, символом перекладу рядка `'\n'` .

Ім'я файлу, наприклад name.gif, із зображенням для завантажувального вікна (splash screen) вказується рядком

```
SplashScreen-Image: name.gif
```

Після того як створено архів Base.jar, можна, можливо виконувати додаток прямо з нього:

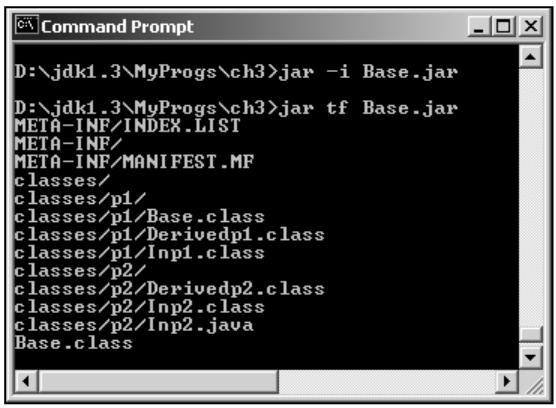
```
java -jar Base.jar
```

Файл INDEX.LIST

Для прискорення пошуку файлів та більш швидкого їх завантаження можна створити файл пошуку INDEX.LIST. Це робиться після формування архіву. Утиліта `jar` запускається ще раз з параметром `-i` , наприклад:

```
jar -i Base.jar
```

Після цього у каталозі META-INF архіву з'являється файл INDEX.LIST. На рис. 25.3 представлено, як створюється файл пошуку та як виглядає вміст архіву після його створення.



Мал. 25.3. створення файлу пошуку

Компоненти JavaBeans

Багато програмістів вважають за краще розробляти додатки з графічним ін- терфейсом користувача за допомогою візуальних засобів розробки IDE (Integrated Development Environment), таких як NetBeans, IntelliJ IDEA, Eclipse, JBuilder та ін. засоби дозволяють поміщати компоненти в контейнер графічно, з допомогою миші.

У вікні програми центральне місце займає форма, на якій розміщуються ком- поненти. Самі компоненти показані ярликами на панелі компонентів. ній зазвичай вище форми або збоку від форми.

Щоб помістити компонент на форму, треба клацнути кнопкою миші на ярлику ком- понента, перенести курсор миші у потрібне місце форми та клацнути кнопкою миші ще раз.

Далі слід визначити властивості (properties) компонента: текст, колір тексту та фону, вид курсору миші, коли він з'являється над компонентом. Властивості визначаються в вікно властивостей, розташованому зазвичай праворуч від форми. Вікно властивостей з'являється частіше всього при виборі пункту меню **Properties** з контекстного меню, що з'являється при клацання правою кнопкою миші на компоненті. У лівій колонці вікна властивостей перечис- лени імена властивостей, в праву колонку треба записати їх значення.

Потім можна задати обробку подій, відкривши другу сторінку вікна властивостей або обравши відповідний пункт контекстного меню.

Для того щоб компонент можна було застосовувати в такому візуальному засобі ботки, як Eclipse, він повинен мати додаткові якості. У нього винен бути ярлик, що міститься на панель компонентів. Серед полів компонента повинні бути виділені властивості (properties), які будуть показані у вікні властивостей. Слід визначити методи доступу `get` *`Xxx`* `()` / `set` *`Xxx`* `()` / `is` *`Xxx`* `()` до кожної властивості. Цими мето- дамі буде користуватися IDE, щоб визначити властивості компонента.

Компонент, забезпечений цими та іншими необхідними якостями, у технології Java називається компонентом JavaBean. До нього може входити один чи кілька класів. Як правило, файли цих класів упаковуються в jar-архів і відзначаються в файлі MANIFEST.MF як `Java-Bean: True` .

Усі компоненти AWT та Swing є компонентами JavaBeans. Якщо ви створюєте свій графічний компонент за правилами, викладеними в *частині III* , то ви теж напів- чаєте свій JavaBean. Але для того щоб не упустити будь-яких важливих якостей JavaBeans, краще використовувати для їх розробки спеціальні засоби, вхідні в склад всіх IDE, наприклад, в NetBeans.

Останні зміни правил створення JavaBeans та приклади дані в документації Java SE, в каталог technotes/guides/beans.

Правила оформлення компонентів JavaBeans викладені у специфікації JavaBeans API Specification, яку можна знайти по адресою:

http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html .

Візуальні засоби розробки – це основне застосування JavaBeans. Головне гідність компонентів, оформлених як JavaBeans, в тому, що вони без праці вбудовуються в будь-який додаток. Більше того, програму можна зібрати з готових JavaBeans як з будівельних блоків, залишається тільки налаштувати їх властивості.

Фахівці пророкують велике майбутнє компонентного програмування. Вони вважають, що скоро будуть створені тисячі компонентів JavaBeans на Усе випадки життя і програмування зведеться до пошуку в Інтернеті потрібних компонентів та збирання з них програми.

Зв'язок з базами даних через JDBC

Здебільшого інформація зберігається над файлах, а базах даних. Додаток повинен вміти зв'язуватися з базою даних для отримання з неї інформації або для поміще- ня інформації в базу даних. Справа тут ускладнюється тим, що СУБД (системи управління базами даних) сильно відрізняються один від одного і зовсім по-різному керують базами даних. Кожна СУБД надає власний набір функцій для доступу до баз даних, і доводиться для кожної СУБД писати свою програму. Але що робити при роботі по мережі, коли невідомо, яка СУБД керує базою сервер?

Вихід було знайдено корпорацією Microsoft, яка створила набір інтерфейсів ODBC (Open Database Connectivity) для зв'язку з базами даних, оформлених як прототипи функцій. цій мови С. Ці прототипи однакові для будь-якої СУБД, вони просто описують на- бору дій з таблицями бази даних. Додаток, що звертається до бази даних, записуються дзвінки функцій ODBC. Для кожної системи управління базами даних розробляється так званий *драйвер ODBC* , що реалізує ці функції для кон- кретної СУБД. Драйвер переглядає додаток, знаходить звернення до бази даних них, передає їх СУБД, отримує від неї результати і підставляє їх у додаток. Ідея виявилася дуже вдалою, і використання ODBC для роботи з базами даних стало загальноприйнятим.

Компанія Sun підхопила цю ідею і розробила набір інтерфейсів та класів, на- званий JDBC, призначений до роботи з базами даних. Ці інтерфейси та клас- си склали пакет `java.sql` , а також пакет `javax.sql` та його підпакети, що входять до Java SE.

"JDBC" - це не абревіатура, а самостійне слово, хоча іноді розшифровується. ється як "Java Database Connectivity".

Крім класів з методами доступу до баз даних для кожної СУБД необхідний драй- вір JDBC - проміжна програма, реалізуюча інтерфейси JDBC методами цієї СУБД. Драйвери JDBC можуть бути написані розробниками СУБД або неза- висимими фірмами. В даний час написано кілька сотень драйверів JDBC для різних СУБД під різні їхні версії та платформи. Їхній список можна подивитися на сторінці **http://developers.sun.com/product/jdbc/drivers** або на **http://www.sqlsummit.com/JDBCVend.htm** .

Існують чотири типу драйверів JDBC:

- драйвер, реалізуючий методи JDBC викликами функцій ODBC. Це так званий *міст* (bridge) JDBC-ODBC. Безпосередню зв'язок з базою при цьому здійснює драйвер ODBC, Котрий повинен бути встановлений на тій машині, на якої працюєпрограма;
- драйвер, що реалізує методи JDBC викликами функцій API СУБД. В цьому випадку на машині повинен бути встановлений клієнт СУБД;

- драйвер, що реалізує методи JDBC викликами функцій мережевого протоколу, незалежного від СУБД, наприклад HTTP. Цей протокол має бути, потім, реалізований засобами СУБД;
- драйвер, реалізує методи JDBC викликами функцій мережевого протоколу СУБД.

Перед зверненням до бази даних необхідно встановити потрібний драйвер, наприклад міст JDBC-ODBC:

```
try{
   Class dr = sun.jdbc.odbc.JdbcOdbcDriver.class;
} catch (ClassNotFoundException e) {
   System.err.println("JDBC-ODBC bridge not found " + e);
}
```

Об'єкт `dr` не знадобиться в програмі, але такий синтаксису.

Інший спосіб установки драйвера показаний у лістингу

25.1.

Після встановлення драйвера можна зв'язатися з базою даних. Методи зв'язку описані в інтерфейсі `Connection` . Примірник класу, що реалізує цей інтерфейс, можна отримати одним із статичних методів `getConnection()` класу `DriverManager` , на- приклад:

```
String url = "jdbc:odbc:mydb";
String login  = "admin";
String password = "1nF4vb";
Connection con = DriverManager.getConnection(url, login, password);
```

Зверніть увагу, як формується адреса бази даних `url` . Він починається зі рядки `"jdbc:"` , потім записується *підпротокол* (subprotocol), в даному прикладі використовується міст JDBC-ODBC, тому записується `"odbc:"` . Далі вказується *адреса* (subname) за правилами підпротоколу, тут просто ім'я локальної бази `"mydb"` . Другий і третій аргументи це ім'я та пароль для з'єднання із базою даних.

Зв'язавшись із базою даних, можна надсилати запити. Запит зберігається в об'єкті, реалізуючому інтерфейс `Statement` . Цей об'єкт створюється методом `createStatement()` , опи-санним в інтерфейсі `Connection` . Наприклад:

```
Statement st = con.createStatement();
```

Потім запит (query) заноситься в цей об'єкт методом `execute()` та потім виконується методом `getResultSet()` . У простих випадках це можна, можливо зробити одним методом `executeQuery()` , наприклад:

```
ResultSet rs = st.executeQuery("SELECT name, code FROM tbl1");
```

Тут з таблиці `tbl1` витягується вміст двох стовпців `name` і `code` і заноситься в об'єкт `rs` класу, що реалізує інтерфейс `ResultSet` .

SQL-оператори `INSERT` , `UPDATE` , `DELETE` , `CREATE TABLE` та ін. у простих випадках виконуютьсяметодом `executeUpdate()` .

Залишається методом `next()` перебрати елементи об'єкта `rs` - рядки отриманого вибору. ки - і отримати дані численними методами `get Xxx ()` інтерфейсу `ResultSet` , на- приклад:

```
while (rs.next()){
   emp[i] = rs.getString("name");
   num[i] = rs.getInt("code");
   i++;
}
```

Методи інтерфейсу `ResultSetMetaData` дозволяють дізнатися кількість отриманих стовпчиків. ців, їх імена та типи, назва таблиці, ім'я її власника та інші відомості про перед- ставлених в об'єкті `rs` відомостях.

Якщо об'єкт `st` отримано методом

```
Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                   ResultSet.CONCUR_UPDATABLE);
```

то можна перейти до попереднього елемента вибірки методом `previous()` , до першого елементу - методом `first()` , до останнього - методом `last()` . Можна також модицировать об'єкт `rs` методами `update` *`Xxx`* `()` і навіть змінювати, видаляти і додавати відповід- ні рядки бази даних. Не Усе драйвери забезпечують ці можливості, тому треба перевірити реальний тип об'єкта `rs` методами `rs.getType()` та `rs.getConcurrency()` .

Інтерфейс `Statement` розширений інтерфейсом `PreparedStatement` , що дозволяє створювати попередньо відкомпільований запит, перед виконанням якого можна за- вати аргументи методами `set` *`Xxx`* `()` .

Інтерфейс `PreparedStatement` , в свою чергу, розширений інтерфейсом `CallableStatement` , в якому описані методи виконання збережених процедур.

У лістингу 25.1 наведено типовий приклад запиту до бази Oracle через драйвер Oracle Thin. Аплет виводить у вікно браузера чотири поля введення для адреси бази, імені та па- роль користувача, та запиту. За замовчуванням формується запит до стартової бази Oracle на локальному комп'ютері. Результат запиту виводиться у вікно браузер.

Листинг 25.1. Апплет, обращающийся к базе Oracle

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

public class JdbcApplet extends Applet implements ActionListener, Runnable{
   private TextField tf1, tf2, tf3;
   private TextArea ta;
   private Button b1, b2;
   private String url = "jdbc:oracle:thin:@localhost:1521:ORCL",
                  login = "Scott",
               password = "tiger",
                  query = "SELECT * FROM dept";
   private Thread th;
   private Vector результатів;
```

```
 public void init(){
   setBackground(Color.white);
   try{
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
   } catch (SQLException e)
      {
      System.err.println(e);
   }
   setLayout(null);
   setFont(new Font("Serif", Font.PLAIN, 14));
   Label l1 = new Label("URL бази:", Label.RIGHT);
   l1.setBounds(20, 30, 70, 25); add(l1);
   Label l2 = new Label("Ім'я:", Label.RIGHT);
   l2.setBounds(20, 60, 70, 25); add(l2);
   Label l3 = new Label("Пароль:", Label.RIGHT);
   l3.setBounds(20, 90, 70, 25); add(l3);
   tf1 = новий TextField(url, 30);
   tf1.setBounds(100, 30, 280, 25); add(tf1);
   tf2 = New TextField (login, 30);
   tf2.setBounds(100, 60, 280, 25); add(tf2);
   tf3 = New TextField (password, 30);
   tf3.setBounds(100, 90, 280, 25); add(tf3);
   tf3.setEchoChar('*');
   Label l4 = new Label("Запит:", Label.LEFT);
   l4.setBounds(10, 120, 70, 25); add(l4);
   ta = new TextArea(query, 5, 50, TextArea.SCROLLBARS_NONE);
   ta.setBounds(10, 150, 370, 100); add(ta);
   Button b1 = New Button ("Відправити");
   b1.setBounds(280, 260, 100, 30); add(b1);
   b1.addActionListener(this);
}
public void actionPerformed(ActionEvent ae){
   url      = tf1.getText();
   login    = tf2.getText();
   password = tf3.getText();
   query    = ta.getText();
   if (th == null) {
      th = new Thread(this);
      th.start();
   }
}
public void run(){
   try{
      Connection con = DriverManager.getConnection(url, login, password);
      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery (query);
      ResultSetMetaData rsmd = rs.getMetaData();
      // Дізнаємося число стовпців
      int n = rsmd.getColumnCount();
      results = new Vector();
```

```
        while (rs.next()){
           String s = " ";
               // Номери стовпців починаються з 1!
           for (int i = 1; i <= n; i++)
              s += "  " + rs.getObject(i);
           results.addElement(s);
        }
        rs.close();
        st.close();
        con.close();
        repaint();
     } catch (Exception e) {
        System.err.println(e);
     }
     repaint();
  }
  public void paint(Graphics g) {
     if (results == null) {
        g.drawString("Can't execute the query", 5, 30);
        return;
     }
     int y = 30, n = results.size();
     for (int i = 0; i < n; i++)
        g.drawString((String)results.elementAt(i), 5, y += 20);
  }
}
```

ПРИМІТКА _ ПЗ Налагодження

У *розділі 24* згадувалося, що для налагодження мережевої програми зручно запустити і клієнт- ську, і серверну частина на одному комп'ютері, звертаючись до серверної частини по адресою
127.0.0.1 або доменне ім'я `localhost` . Не забувайте, що аплет може зв'язатися по мережі тільки з тим хостом, звідки він завантажений. Отже, на комп'ютері повинен робо- ти Web-сервер. Якщо Web-сервер прослуховує порт 8080, то щоб завантажити HTML- сторінку з аплетом, треба в браузері вказувати адресу URL виду **http://localhost:8080/public/JdbcApplet.html** . При цьому врахуйте, що Web-сервер встановлює свою ієрархію каталогов, і каталог public насправді може бути каталогом usr/local/http/public або яким- небудь іншим.

Таким чином, JDBC дозволяє зробити весь цикл роботи з базою даних. Детально з усіма можливостями JDBC можна познайомитись, прочитавши специфікацію JDBC, наявну за адресою: **http://java.sun.com/products/jdbc/** .

У документації Java SE, у каталозі technotes/guides/jdbc/, є посилання на посібники з використанню JDBC.

Запитання для самоперевірки

1. Чому в Java створено свій формат архівування JAR?
2. Потрібно чи розпаковувати jar-архіви для використання класів, містяться в них?
3. Можна, можливо чи упаковувати аплети в jar-архів?

4. Куди треба поміщати jar-файли для використання їх в додатку?
5. Що таке JavaBeans: класи, інтерфейси, пакети, правила оформлення класів?
6. Що повинно бути в класі, званому JavaBean?
7. Де застосовуються JavaBeans?
8. Що таке JDBC?
9. Які існують типи драйверів JDBC?
10. Які фірми розробляють драйвери JDBC?

ГЛАВА 26

Сервлети

Спочатку перед HTTP-серверами стояло просте завдання: знайти та відправити клієнту файл, вказаний в отриманому від клієнта запиті. Запит складався теж дуже просто по правилам протоколу HTTP в спеціально придуманою формі URL.

Потім знадобилося зробити на сервері якусь невелику попередню про- роботу відправляється файлу. З'явилися включення на стороні сервера SSI (Server Side Include) і різні прийоми динамічної генерації сторінок HTML. HTTP- сервер ускладнився і став називатися Web-сервер.

Потім виникла потреба виконувати на сервері процедури. У запит URL вста- вілі можливість виклику процедур, а на сервері реалізували технологію CGI (Common Gateway Interface), про яку ми говорили у попередніх розділах. Тепер у запиту URL вказується процедура, яку треба виконати на сервері, та запису- ються аргументи цієї процедури в вигляді пар "ім'я - значення", наприклад:

http://some.firm.com/cgi-bin/mycgiprog.pl?name=Ivanov&age=27

Для складання таких запитів в мова HTML введено тег `<form>` .

Web-сервер, отримавши запит, завантажує і запускає CGI-процедуру (в попередньому прикладі це процедура mycgiprog.pl), розташовану на сервері в каталозі cgi-bin, та передає їй значення "Ivanov" та "27" аргументів name та age. Процедура оформляє свій відповідь в вигляді сторінки HTML, яку Web-сервер відправляє клієнту.

Процедуру CGI можна написати будь-якою мовою, аби він сприймав стандарт- ний введення і міг направити результат роботи процедури стандартний висновок. Неожи-цю популярність отримала мова Perl. Виявилося, що на ньому зручно писати CGI-програми. Виникли спеціальні мови: PHP, ASP, серверний варіант JavaScript.

Технологія Java не могла пройти повз таку насущну потребу і відгукнулася на її створенням сервлетів і мовою JSP (JavaServer Pages).

Сервлети (servlets) виконуються під керуванням Web-сервера подібно до того, як аплітки виконуються під керуванням браузера, звідки і походить їх назва. Для стеження за роботою сервлетів та управління ними створюється спеціальний програмний модуль, який називається *контейнером сервлетів* (servlet container). Слово "контейнер" у російською означає пасивну ємність стандартних розмірів, але контейнер серв- літів активний, він завантажує сервлети, ініціалізує їх, передає їм запити клієнтів. тов, приймає відповіді. Сервлети не можуть працювати без контейнера, як аплети не

можуть працювати без браузера. Жаргонне вираз "сервлетний движок", відбува- ще від англійського "servlet engine", краще виражає суть справи, ніж вираз "кон- тейнер сервлетів".

Web-сервер, з контейнером сервлетів та іншими контейнерами, став назвою. ватися *сервером додатків* (application server, AS).

Щоб сервлет міг працювати, він повинен бути зареєстрований у контейнері, за термнології специфікації "Java Servlet Specification" *встановлений* (deploy) до нього. *Встановка* (deployment) сервлета в контейнер включає отримання унікального імені та опрозподіл початкових параметрів сервлета, запис їх у конфігураційні файли, соз- дання каталогів для зберігання всіх файлів сервлет і інші операції. Процес установки сильно залежить від контейнера. Одному контейнеру достатньо скопіювати сервлет у певний каталог, наприклад autodeploy/ або webapps/, іншому треба по- після цього перезапустити контейнер, для третього треба скористатися утилітою уста- новки. У стандартному контейнері Java EE SDK така утиліта називається deploytool.

Один контейнер може керувати роботою кількох встановлених у нього сервлетів. При цьому один контейнер здатний в один і той же час працювати в кількох вірту- альних машинах Java, утворюючи розподілене Web-додаток. Самі ж віртуальні машини Java можуть працювати на одному комп'ютері або на різних комп'ютери.

Контейнери сервлетів створюються як частина Web-сервера або як вбудований в нього модуль. Велику популярність отримали вбудовані контейнери Tomcat танні спільнотою Apache Software Foundition у рамках проекту Jakarta, Resin фірми Caucho, JRun фірми Macromedia. Точне розподіл обов'язків між Web- сервером та контейнером сервлетів випадає на частку їх виробників.

Сервер Tomcat можна скопіювати зі сторінки **http://tomcat.apache.org/** та встановити окремо. Зручніше скопіювати його дистрибутив у вигляді zip-файлу, його треба просто раз- повернути в якийсь каталог. Для виконання всіх прикладів цього та наступного розділу знадобиться Tomcat версії не менше 7, "вміє" виконувати Servlet 3.0 і JSP 2.2.

Web-додаток

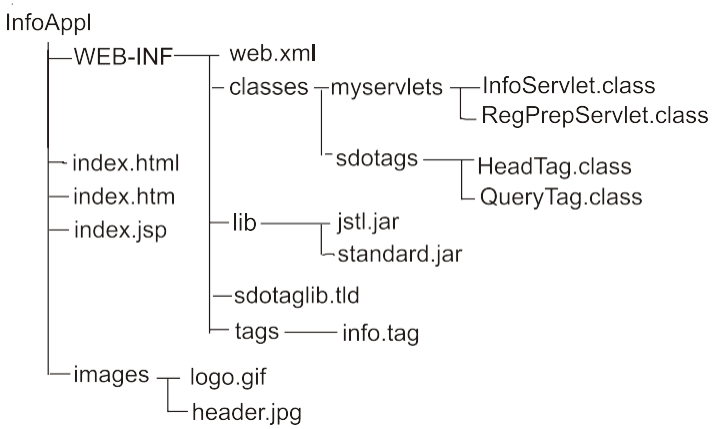
Як правило, сервлет не виконується один. Він працює у складі Web-додатку. *Webдодаток* (web application) складають усі ресурси, написані для обслуговування. вання запитів клієнта: сервлети, JSP, сторінки HTML, документи XML, інші до- кументи, зображення та креслення, музичні та відеофайли. Специфікація "Java Servlet Specification" описує структуру каталогів, що містять усі ці ресурси. Вона зображена на Мал. 26.1.

Як видно з малюнка, все, що відноситься до даного Web-додатку, міститься в одному каталозі, ім'я якого буде ім'ям Web-програми. У прикладі це каталог InfoAppl. У цьому каталозі обов'язково має бути каталог WEB-INF і необов'язково інші каталоги та файли. Все, що знаходиться в каталозі WEB-INF та його підкаталогах, недоступне для клієнта. Це "внутрішня кухня" Web-програми. Те, що розташовано в додатку поза каталогу WEB-INF, доступно клієнту.

У каталозі WEB-INF повинен бути конфігураційний XML-файл з ім'ям web.xml, в якому описано Web-додаток: його ресурси, їх адреси і зв'язку між ресурсами.

Це мінімальний склад Web-додатку - каталог з його ім'ям, у ньому підкаталог WEB-INF, а у ньому файл web.xml. Втім, при використанні анотацій навіть файл web.xml стає необов'язковим.

Скомпільовані сервлети зазвичай розташовуються в підкаталоги каталогу WEB-INF/classes відповідно зі своєю структурою пакетів і підпакети.



Мал. 26.1. Структура каталогів Web-додатки

Всі Web-додаток повністю часто упаковується в один файл за технологією JAR. Такий файл зазвичай отримує розширення war (Web ARchive). Цей файл можна пере- носити з одного Web-сервера на інший, при цьому багато контейнерів сервлетів можуть запускати Web-додаток прямо з архіву, не розпаковуючи його. Серверу Tomcat достатньо занести WAR-архів або всю структуру каталогів програми в каталог webapps. Сервер його "побачить" і негайно, без перезапуску, залучить до роботи. Необхідність розпакування архіву або робота прямо з архівом вказується при налаштуванні Tomcat в його конфігураційному файлі server.xml.

Інтерфейс *Servlet*

Як водиться в технології Java, поняття "сервлет" описується інтерфейсом `Servlet` . Розглянемо його докладніше.

Раніше вже говорилося про те, що сервлет виконується в контейнері подібно до того, як аплет виконується у браузері. Ця подібність посилюється тим, що контейнер ініціалізує сервлет методом `init()` так само, як браузер ініціалізує аплет, але, у від- личіє від аплету, у методу `init()` , описаного інтерфейсом `Servlet` , є аргумент типу `ServletConfig` :

```
public void init(ServletConfig conf);
```

Об'єкт, описаний інтерфейсом `ServletConfig` , створюється Web-додатком і передається контейнеру для ініціалізації сервлету. Інформація, необхідна для створення об'єкта, міститься в конфігураційному файлі Web-додатки.

Конфігураційний файл

Конфігураційний файл (Deployment descriptor) описує ресурси, складники Web-додаток: сервлети, їх фільтри та слухачі, сторінки JSP, документи HTML та XML, зображення та документи інших типів. Він формується під час створення Web- програми та заповнюється при установці сервлету та інших ресурсів в контейнер. Конфігураційний файл записується мовою XML і називається web.xml. Він розпо- лагается у каталозі WEB-INF, одному з каталогів Web-додатка, і створюється вруч- ну, утилітою установки сервлета в контейнер або за допомогою IDE, як NetBeans або Eclipse. Кожна фірма-виробник контейнера сервлетів надає свою утиліту установки або make-файл, містить команди установки. Треба помітити, що замість будівельника `make` у технології Java використовуються інші будівельники, на- писані на мовою Java: будівельник `ant` , розроблений Apache Software Foundation в рамках проекту Jakarta, будівельник Maven – ще одна розробка Apache Software Foundation, або Ivy - Знов-таки розробка Apache.

Утиліта установки контейнера Tomcat запускається з браузера, вона розташована на його сторінці /manager/html. Для того, щоб запустити утиліту, у браузері треба набрати приблизно такий рядок: **http://localhost:8080/manager/** .

У лістингу 26.1 показаний фрагмент конфігураційного файлу web.xml, створеного для контейнери Tomcat. З мовою XML ми познайомимося в *розділі 28* , а поки дивіться роз'- пояснення елементів XML в коментарях.

Листинг 26.1. Конфигурационный файл web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  " http://java.sun.com/j2ee/dtds/web-app_2_3.dtd ">

<web-app>
<!--
Якщо в запиті не вказано ресурс, то викликається сервлет по
замовчуванням.Нижче записано його ім'я "default" і повне ім'я класу
сервлету.
-->
    <servlet>
        <servlet-name>default</servlet-name>
        <servlet-class>
            org.apache.tomcat.servlets.DefaultServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
<!--
Якщо прийшов запит до сервлету, не описаному в даним файл,то
викликається сервлет з ім'ям "Invoker".
-->
    <servlet>
        <servlet-name>invoker</servlet-name>
```

```
        <servlet-class>
            org.apache.tomcat.servlets.InvokerServlet
        </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
    </servlet>
<!--
Усе запити до сторінкам JSP на початку обробляються
сервлетом JspServlet. Йому дано ім'я "JSP".
-->
    <servlet>
        <servlet-name>jsp</servlet-name>
        <servlet-class>
            org.apache.jasper.runtime.JspServlet
        </servlet-class>

<!-- uncomment the following to use Jikes for JSP compilation
Заберіть коментар, якщо ви використовуєте компілятор jikes.
        <init-param>
            <param-name>jspCompilerPlugin</param-name>
            <param-value>
              org.apache.jasper.compiler.JikesJavaCompiler
            </param-value>
        </init-param>
-->
      <load-on-startup>
          -2147483646
      </load-on-startup>
    </servlet>
<!--
Деяким шляхам-псевдонімам зіставляється сервлет, який викликається
при вказівці в рядку URL цього шляхи. Шлях відраховуєтьсящодо
кореневого каталогу контейнера, найчастіше це public_html або
webapps.
Якщо в адреса URL вказано каталог servlet, то викликається
сервлетз ім'ям "invoker", т. е. InvokerServlet.
-->
    <servlet-mapping>
        <servlet-name>invoker</servlet-name>
        <url-pattern>/servlet/*</url-pattern>
    </servlet-mapping>
<!--
Якщо в адреса вказано шлях до сторінці JSP, то на початку
викликаєтьсясервлет з ім'ям "jsp", т. е. сервлет JspServlet.
-->
    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
```

```
        <url-pattern>*.jsp</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
<!--
Нижче йде довгий перелік відповідностей розширень імен
файліві MIME-типів вмісту цих файлів.
-->
    <mime-mapping>
        <extension>txt</extension>
        <mime-type>text/plain</mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>html</extension>
        <mime-type>text/html</mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>htm</extension>
        <mime-type>text/html</mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>gif</extension>
        <mime-type>image/gif</mime-type>
    </mime-mapping>
<!--
І так далі.
Зрештою, йде перелік файлів, які посилаються клієнту призверненні
тільки до каталогу без вказівки ресурсу.
-->
  <welcome-file-list>
    <welcome-file>index.jsp </welcome-file>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
  </welcome-file-list>
</web-app>
```

Як бачите, конфігураційний файл web.xml дуже об'ємний. У великому Web- Додаток він стає складним і важко читається. Починаючи з версії Servlet 3.0, його можна скласти з декількох файлів, що містять окремі фрагменти з опи- саніями окремих сервлетів і інших ресурсів. Кожен фрагмент, в відмінність від основного файлу web.xml, обрамляється XML-елементом <web-fragment>, а не елемент- том <web-app>. Файл з фрагментом повинен називатися web-fragment.xml і розташовується. гатися в каталозі META-INF. Порядок підключення фрагментів вказується елемент- тамі XML в кожному фрагменті або в файлі web.xml.

Інтерфейс *ServletConfig*

Кожен об'єкт типу `ServletConfig` містить ім'я сервлету, витягнуте з елемента `<servlet-name>` конфігураційного файлу, набір початкових параметрів, взятих з еле- ментів `<init-param>` , і контекст сервлета як об'єкта типу `ServletContext` . Ці конфі- гураційні параметри сервлет може отримати методами

```
public String getServletName();
public Enumeration getInitParameterNames();
public String getInitParameter(String name);
public ServletContext getServletContext();
```

описаними в інтерфейсі `ServletConfig` .

Початкові параметри записуються в конфігураційний файл web.xml во час установки сервлета вручну або за допомогою утиліти установки. Механізм завдання та читання початкових параметрів сервлета дуже схожий на механізм визначення пара- метрів аплету, що записуються в теги `<param>` і читаються методами `getParameter()` ап- пліту.

У лістингу 26.2 наведено найпростіший сервлет, що відправляє клієнту своє ім'я та на- чальні параметри.

Листинг 26.2. Начальные параметры сервлета

```
package myservlets;
import java.io.*;
import java.util.*;import javax.servlet.*;
public class InfoServlet extends GenericServlet{
private ServletConfig sc;

    @Override
    public void init(ServletConfig conf)
           throws ServletException{
       super.init(conf);
       sc = conf;  }

    @Override
    public void service(ServletRequest req, ServletResponse resp)
           throws ServletException, IOException{

    resp.setContentType("text/html; charset=windows-1251");

    PrintWriter pw = resp.getWriter();

        pw.println("<html><head>");
        pw.println("<title>Параметри сервлету</title>");
        pw.println("</head><body><h2>Відомості про
        сервлеті</h2>");
    pw.println("Ім'я сервлет - " + sc.getServletName() + "<br>");

    pw.println("Параметри сервлету: <br>");

    Enumeration names = sc.getInitParameterNames();
```

```
        while (names.hasMoreElements()){
           String name = (String)names.nextElement();
           pw.print(name + ": ");
           pw.println(sc.getInitParameter(name) + "<br>");
    }

        pw.println("</body></html>");
        pw.flush();
        pw.close();
     }
     public void destroy(){
        sc = null;
     }
}
```

Повністю код листингу 26.2 буде детально роз'яснено пізніше, а поки що запам'ятайте два правила:

- перевизначаючи метод `init(ServletConfig)` , викликайте метод `init(ServletConfig)` супер- класу;
- кодування відповіді встановлюйте методом `setContentType()` *перед* отриманням посилання- кі на вихідний потік методом `getWriter()` . Це відноситься і до іншим заголовкам від-вета.

Код лістингу 26.2 компілюється звичайним чином, а потім отриманий файл з серв- Влітку `InfoServlet.class` встановлюється в контейнер. Процедура установки виконує- ся відповідною утилітою, що входить до складу контейнера сервлетів або сервера додатків, make-файлом або ant-файлом. Часто достатньо помістити додаток у каталог, званий webapps, autodeploy, або якось ще залежно від сервера додатків. Після закінчення процедури встановлення сервлет можна викликати з браузера, набравши в нім рядок адреси виду:

http://<servlethost>:8000/InfoAppl/servlet/InfoServlet

Браузер покаже сторінку HTML, сформовану сервлетом. Для інших контейнері-рів і серверів додатків номер порту та шлях до сервлету буде, звісно, іншим.

> ***ПРИМІТКА*** _ ***ПЗ Налагодження***
>
> Для багатьох контейнерів та серверів додатків недостатньо просто перекомпілювати змінений код сервлета, щоб контейнер сприйняв оновлений сервлет. Треба ще сервлет перевстановити.

Багато серверів додатків, серед них і останні версії Tomcat, не запускають сервлети окремо, лише у складі Web-приложения. Зазвичай утиліти уста- новки створюють необхідну для Web-програми структуру каталогів, але це можна зробити і вручну. Звернення до сервлету в складі Web-додатки виглядає простіше:

http://<servlethost>:8000/InfoAppl/InfoServlet

Як видно з сигнатури методу `getServletContext()` , контейнер отримує доступ до ще одному об'єкту - об'єкту типу `ServletContext` , що містить контекст сервлету.

Контекст сервлет

Для всіх сервлетів, працюючих в рамках одного Web-додатки, створюється один контекст. *Контекст* (context) сервлетів складають каталоги та файли, що описують Web-додаток. Вони містять, зокрема, код сервлета, зображення, креслення, конфігурації. раціональний файл web.xml і його фрагменти, коротше кажучи, все, що стосується сервлета. При ініціалізації сервлет деякі відомості про його контексті заносяться в об'єкт ти- па `ServletContext` . Методи цього інтерфейсу дозволяють сервлету отримати відомості, містяться в контексті.

Метод `getServerInfo()` дозволяє отримати ім'я і версію Java EE SDK, методи `getMajorVersion()` і `getMinorVersion()` повертають номер версії і модифікації Servlet API.

У контексті можна, можливо визначити початкові параметри, загальні для всього Web-при- кладення. Вони задаються при створенні Web-програми вручну або за допомогою ути- літи установки і зберігаються в конфігураційному файлі web.xml в елементах `<context-param>` . Їх імена і значення можна, можливо отримати методами

```
public Enumeration getInitParameterNames();
public String getInitParameter(String name);
```

Крім рядкових параметрів у контексті допустимо визначити атрибути, значеннями яких можуть бути об'єкти будь-яких типів Java. Їх імена і значення можна напів- читати методами

```
public Enumeration getAttributeNames();
public Object getAttribute(String name);
```

Встановити і видалити атрибути можна, можливо методами

```
public void setAttribute(String name, Object value);
public void removeAttribute(String name);
```

Атрибути це зручний спосіб зберігати об'єкти, загальні для всього Web-при- ложения, поділювані всіма сервлетами, які входять у Web-додаток, і незалежні- ми е від окремих запитів.

Метод *Service*

Основна робота сервлет укладена в методі

```
public void service(ServletRequest req, ServletResponse resp);
```

До цього методу контейнер автоматично звертається після завершення методу `init()` і передає йому об'єкт `req` типу `ServletRequest` , що містить всю інформацію, знаходячи- у запиті клієнта. Створенням та заповненням об'єкта `req` теж займається кон- тейнер сервлетів. Крім того, контейнер створює та передає методу `service()` посилання на порожній об'єкт `resp` типу `ServletResponse` .

Метод `service()` обробляє відомості, що містяться в об'єкті `req` , і заносить результат- тати обробки в об'єкт `resp` . Заповнений об'єкт `resp` передається контейнеру, який через Web-сервер надсилає відповідь клієнту. Всі ці дії виконуються мето- дамі, описаними в інтерфейсах `ServletRequest` і `ServletResponse` .

Інтерфейс *ServletRequest*

В інтерфейсі `ServletRequest` , який повинен реалізувати кожен контейнер серв- тов, описана маса методів `get Xxx ()` , що повертають параметри запиту або `null` , якщо параметр невідомий.

Методи `getRemoteAddr()` , `getRemoteHost()` та `getRemotePort()` повертають IP-адресу, повну DNS-ім'я відправника запиту або proxy-сервера і його номер порту, а методи `getServerName()` і `getServerPort()` повертають ім'я та номер порту сервера, що прийняв запит.

Методи `getLocalAddr()` , `getLocalName()` та `getLocalPort()` повертають IP-адресу, повну DNS-ім'я мережевого інтерфейсу, з якого отримано запит, та його номер порту.

Метод `getScheme()` повертає схему запиту: `http:` , `https:` , `ftp:` і т. буд., а метод `getProtocol()` - ім'я протоколу в вигляді рядки, наприклад `"HTTP/1.1"` .

Методи `getContentType()` і `getCharacterEncoding()` повертають MIME-тип і кодування запиту, якщо вони вказані в заголовку запиту, а метод `getContentLength()` — довжину тіла запиту в байтах, якщо вона відома, або -1, якщо довжина невідома.

Метод `setCharacterEncoding(String)` встановлює кодування, якщо воно не визначається методом `getCharacterEncoding()` , або листує кодування, зазначену в запит. До цього методу часто доводиться звертатися для правильного перетворення пара- метрів запиту, що прийшли в байтовий кодування, в рядок типу `String` . приклад такого звернення наведено у лістингу 26.5. Цей метод слід застосовувати до розбору пара- метрів запиту.

Імена і значення параметрів, що прийшли з запитом, можна, можливо отримати методами

```
public Enumeration getParameterNames();
public String getParameter(String name);
public String[] getParameterValues(String name);
public Map getParameterMap();
```

Якщо запит має якісь атрибути, то їх імена і значення можна отримати ме- тодами

```
public Enumeration getAttributeNames();
public Object getAttribute(String name);
```

Нарешті, інтерфейс описує два вхідні потоки отримання тіла запиту: байто- вий та символьний. Байтовий потік реалізується спеціально розробленим класом `ServletInputStream` .

Клас `ServletInputStream` - це абстрактний клас, що розширює клас `InputStream` . Він додає до методів свого суперкласу лише один метод

```
public int readLine(byte[] buf, int offset, int length);
```

читає рядок тіла запиту в заздалегідь певний буфер `buf` . Читання починається з байта з номером `offset` і продовжується до досягнення символу перекладу рядки `'\n'` або до досягнення кількості прочитаних символів `length` . Метод повертає число прочитаних байтів або -1, якщо вхідний потік вже вичерпано.

Отримати байтовий потік з запиту `req` можна, можливо методом

```
public ServletInputStream getInputStream();
```

Другий потік - символьний - це потік класу `BufferedReader` , який ми розглянули. релі в *главі 23* . Отримати його можна методом

```
public BufferedReader getReader();
```

У пакеті `javax.servlet` є пряма реалізація інтерфейсу `ServletRequest` - клас `ServletRequestWrapper` . Об'єкт цього класу створюється конструктором

```
public ServletRequestWrapper(ServletRequest req);
```

і має всі методи інтерфейсу `ServletRequest` . Розробники, які бажають роз- ширити можливості об'єкта, що містить запит, або створити фільтр, можуть розширити. рити клас `ServletRequestWrapper` .

Інтерфейс *ServletResponse*

Результати своєї роботи метод `service()` заносить до об'єкта типу `ServletResponse` , посилання на Котрий надано другим аргументом методу `service()` .

Методи `setContentType(String)` та `setLocale(Locale)` встановлюють у заголовок відповіді MIME-тип та локаль тіла відповіді, а метод `setContentLength(int)` записує довжину тіла відповіді. Якщо треба встановити тільки кодування символів у відповіді, то можна спро- зуватися методом `setCharacterEncoding(String)` .

Тіло відповіді передається контейнеру через байтовий чи символьний вихідний потік. Байтовий потік спеціально розробленого класу `ServletOutputStream` повертає метод

```
public ServletOutputStream getOutputStream();
```

Абстрактний клас `ServletOutputStream` розширює клас `OutputStream` , додаючи до нього методи `print( xxx )` для виведення типів `boolean` , `char` , `int` , `long` , `float` , `double` , `String` та методи `println( xxx )` для тих самих типів, що додають до даних символи `"\r\n"` . Ще один метод `println()` без аргументів просто заносить в вихідний потік символи `"\r\n"` .

Символьний потік можна, можливо отримати методом

```
public PrintWriter getWriter();
```

Саме він використаний у лістингу 26.2.

У пакеті `javax.servlet` є пряма реалізація інтерфейсу `ServletResponse` - клас `ServletResponseWrapper` . Об'єкт цього класу створюється конструктором

```
public ServletResponseWrapper(ServletResponse resp);
```

і має всі методи інтерфейсу `ServletResponse` . Розробники, які бажають роз- ширити можливості об'єкта, що містить відповідь, наприклад для створення фільтра, мо- гут розширити клас `ServletResponseWrapper` .

Цикл роботи сервлет

Сервлет завантажується контейнером, як правило, при першим запиті до ньому або во вре-мя запуску контейнер. Після виконання запиту сервлет може бути залишений в спя-

ному стані, чекаючи наступного запиту, або вивантажений, попередньо виконавши метод `destroy()` . Це залежить від реалізації контейнера сервлетів.

Робота сервлета починається з методу `init()` , потім виконується метод `service()` , кото- рий може створювати об'єкти, звертатися до їх методів, зв'язуватися з базами даних та віддаленими об'єктами, виконуючи звичайну роботу звичайного класу Java. При цьому треба враховувати, що сервлет може бути направлено відразу кілька запитів. Число одночасних запитів у промислових системах досягає сотень та тисяч. Для об- Роботи кожного запиту контейнер створює новий підпроцес (thread), що виконує метод `service()` . Тому виконуйте наступні правила:

- розробляючи метод `service()` , завжди майте на увазі, що він буде паралельно виконуватися кількома підпроцесами, та вживайте заходів до синхронізації їх роботи;
- виносите створення об'єктів, загальних для сервлета, визначення параметрів, пула з'єднань з базами даних та віддаленими об'єктами, у поля класу та метод `init()` ;
- завершальні дії, такі як закриття потоків, запис результатів на диск, закриття з'єднань, виносите метод `destroy()` , що виконується при закритті сервлету.

Клас *GenericServlet*

Абстрактний клас `GenericServlet` реалізує одночасно інтерфейси `Servlet` та `ServletConfig` . Крім реалізації методів обох інтерфейсів до нього введено порожній метод `init()` без аргументів. Цей метод виконується автоматично після методу `init(ServletConfig)` , точніше кажучи, останній метод реалізовано так:

```
public void init(ServletConfig config) throws ServletException{
   this.config = config;
   log("init");
   this.init();
}
```

Тому зручно всю ініціалізацію записувати в метод `init()` без аргументів, не переймаючись про виклик `super.init(config)` .

Клас `GenericServlet` залишає нереалізованим лише метод `service()` . Зручно створити вати сервлети, розширюючи цей клас і перевизначаючи тільки метод `service()` .

Так і зроблено в лістингу 26.2. Можна, можливо записати його простіше, не визначаючи метод `init()` , а прямо використовуючи реалізацію методів інтерфейсу `ServletConfig` , зроблену в класі `GenericServlet` . Цей варіант наведено в лістингу 26.3. У нього додано ще отриманняконтексту сервлету.

Листинг 26.3. Упрощенное чтение начальных параметров сервлета

```
package myservlets;
import java.io.*;
```

```
import java.util.*;import javax.servlet.*;
public class InfoServlet extends GenericServlet{

  @Override
  public void service(ServletRequest req, ServletResponse resp)
           throws ServletException, IOException{

    ServletContext cont = getServletContext();

    resp.setContentType("text/html; charset=windows-1251");

    PrintWriter pw = resp.getWriter();

        pw.println("<html><head>");
        pw.println("<title>Параметри сервлету</title>");
        pw.println("</head><body><h2>Відомості про
        сервлеті</h2>");
    pw.println("Ім'я сервлет - " + getServletName() + "<br>");

    pw.println("Параметри та контекст сервлета: <br>");

    Enumeration names = getInitParameterNames();

        while (names.hasMoreElements()){
             String name = (String)names.nextElement();
             pw.print(name + ": ");
             pw.println(getInitParameter(name) + "<br>");
     }

     pw.println("Сервер: " + cont.getServerInfo() +"<br>");

         pw.println("</body></html>");
         pw.flush();
         pw.close();
    }
}
```

Робота по протоколу HTTP

Більшість запитів до сервлетів відбувається за протоколом HTTP, який у настою- ще час реалізується за рекомендацією RFC 2616. Для зручності роботи з цим про- токолом інтерфейси `ServletRequest` та `ServletResponse` розширено інтерфейсами `HttpServletRequest` та `HttpServletResponse` відповідно. При розширення інтерфейсів у них додані методи, характерні для протоколу HTTP.

Інтерфейс *HttpServletRequest*

Декілька методів інтерфейсу `HttpServletRequest` дозволяють розібрати HTTP-запит.

Перший рядок запиту, виконаного за протоколом HTTP, складається з методу передачі даних, адреси URI та версії протоколу. Рядок завершується символами CRLF. Елементи рядка поділяються пробілами, тому всередині кожного елемента першої стро- кі запиту прогалин бути не повинно.

Перша рядок запиту виглядає приблизно так:

```
GET http://some.firm.com/MyWebAppl/servlet/MyServlet?age=27 HTTP/1.1
```

Запит HTTP починається з одного зі слів `GET` , `POST` , `HEAD` або іншого слова, позначаю-ного метод передачі даних. Дізнатись HTTP-метод передачі дозволяє метод інтерфейсу

```
public String getMethod();
```

Далі в запит, після пробілу, йде адреса URI, Котрий розбирається декількома мето-жінки:

- `public String getRequestURL()` - повертає адреса URL від назви схеми `http` до запитального знак;
- `public String getServletPath()` - повертає частина цього адреси, показує шляхдо сервлету.

Частина шляхи, визначальна контекст сервлету, повертається методом

```
public String getContextPath();
```

Частина URI після запитального знаку повертається методом

```
public String getQueryString();
```

Після імені сервлет може йти додатковий шлях до якомусь файлу, кото-рий можна отримати методом

```
public String getPathInfo();
```

Цей ж шлях, доповнений до абсолютного шляхи до каталогу документів сервера, мож-але отримати методом

```
public String getPathTranslated();
```

Після першою рядки запиту можуть йти заголовки запиту: `Accept` , `Accept-Charset` ,`Accept-Language` , `User-Agent` і інші заголовки, описані в рекомендації RFC 2616. Дізнатись заголовки запиту і їх значення можна, можливо методами

```
public  Enumeration  getHeaderNames();
public Enumeration getHeaders(String name);
public String getHeader(String name);
public int getIntHeader(String name);
public long getDateHeader(String name);
```

Нарешті, можна отримати cookies, що зберігаються в браузері клієнта, у вигляді масиву ектів класу `Cookie` методом

```
public Cookie[] getCookies();
```

У пакеті `javax.servlet.http` є пряма реалізація інтерфейсу `HttpServletRequest` - клас `HttpServletRequestWrapper` , що розширює клас `ServletRequestWrapper` . Об'єкт цього класу створюється конструктором

```
public HttpServletRequestWrapper(HttpServletRequest req);
```

і має всі методи інтерфейсу `HttpServletRequest` . Розробники, які бажають розширити можливості об'єкта, що містить HTTP-запит, наприклад для написання фільтра, можуть розширити клас `HttpServletRequestWrapper` .

Інтерфейс *HttpServletResponse*

При складанні відповіді по протоколу HTTP можна, можливо використовувати додаткові ме-тоді, увімкнені в інтерфейс `HttpServletResponse` .

Метод

```
public void setHeader(String name, String value);
```

встановлює заголовок відповіді з ім'ям `name` та значенням `value` . Старе значення, якщо воно існувало, при цьому стирається.

Якщо треба дати кілька значень заголовку з ім'ям `name` , слід скористатися-ся методом

```
public void addHeader(String name, String value);
```

Для заголовків з цілими значеннями то ж саме робиться методами

```
public void setIntHeader(String name, int value);
public void addIntHeader(String name, int value);
```

Заголовок з датою записується методами

```
public void setDateHeader(String name, long date);
public void addIntHeader(String name, int value);
```

Код відповіді (status code) встановлюється методом

```
public void setStatus(int sc);
```

Як і Усе заголовки, цей метод записується перед отриманням потоку класу `PrintWriter` . Аргумент методу `sc` - це одна з безлічі констант, наприклад констан- та `SC_OK` , відповідна коду відповіді 200 - успішна обробка запиту, `SC_BAD_REQUEST` - код відповіді 400 і т. д. Близько сорока таких статичних констант при- ведено в документації до інтерфейсу `HttpServletRequest` . Метод `setStatus()` застосовується для повідомлень про успішну обробку з кодами 200-299.

Повідомлення про помилці посилаються методом

```
public void sendError(int sc, String message);
```

Зазвичай код відповіді заноситься до повідомлення про помилку `message` . Якщо треба надіслати стан- дартне повідомлення про помилка, наприклад "404 Not Found", то застосовується другий метод:

```
public void sendError(int sc);
```

Цей метод використаний в лістингу 26.6.

Повідомлення методом `sendError()` надсилається замість результатів обробки запиту. Якщо спробувати надіслати після нього відповідь, то система викине виняток класу `IlegallegalException` .

Зрештою, до запиту можна, можливо додати cookie методом

```
public void addCookie(Cookie cookie);
```

У пакеті `javax.servlet.http` є пряма реалізація інтерфейсу `HttpServletResponse` - клас `HttpServletResponseWrapper` , що розширює клас `ServletResponseWrapper` . Об'єкт цього класу створюється конструктором

```
public HttpServletResponseWrapper(HttpServletResponse resp);
```

і має всі методи інтерфейсу `HttpServletResponse` . Розробники, які бажають розширити можливості об'єкта, містить запит, можуть розширити клас `HttpServletResponseWrapper` .

Клас *HttpServlet*

Для використання особливостей протоколу HTTP клас `GenericServlet` розширений абстрактним класом `HttpServlet` . Головна особливість цього класу полягає в тому, що, розширюючи його, не треба перевизначати метод `service()` . Він уже визначений, причому реа- лізований так, що служить диспетчером, що викликає методи `doGet()` , `doPost()` та інші методи, що обробляють HTTP-запити з конкретними методами передачі даних `GET` , `POST` і ін.

На початку метод

```
public void service(ServletRequest req, ServletResponse resp);
```

аналізує типи аргументів `req` та `resp` . Ці типи повинні бути на самому справі `HttpServletRequest` та `HttpServletResponse` . Якщо це не так, то метод викидає вик- чення класу `ServletException` та завершується.

Якщо аргументи `req` та `resp` відповідного типу, то методом `getMethod()` визначається HTTP-метод передачі даних і викликається метод, відповідний цьому HTTP- методом, а саме один з методів

```
protected void do Xxx (HttpServletRequest req, HttpServletResponse resp);
```

де *XX* означає `Get _ Post _ Head , Delete , Options , Put` або `Trace` .

Ось ці методи і треба перевизначати, розширюючи клас `HttpServlet` . Методи `doHead()` , `doOptions()` і `doTrace()` вже реалізовані відповідно до рекомендації RFC 2616, їх рідко доводиться перевизначати. Інші методи "реалізовані" таким чином, що просто надсилають повідомлення про те, що вони не реалізовані. Найчастіше приходить- ся перевизначати методи `doGet()` і `doPost()` .

Анотації сервлет

Усю конфігурацію сервлета типу HttpServlet, яка була описана в попередніх пунктах і яка зазвичай записується в конфігураційний файл web.xml, тожно зробити за допомогою анотацій прямо в коді сервлета. У такому випадку конфігурацій- ний файл web.xml стає необов'язковим. Якщо ж файл web.xml є, то записані в ньому значення перекриватимуть значення, вказані в інструкції. Це зручно в тих випадках, коли треба змінити конфігурацію сервлета без його перекомпі- ляції.

Анотації знаходяться в пакеті `javax.servlet.annotation` , який треба вказати в операторе `import` . Ось як вони виглядають:

```
import javax.servlet.*;
import javax.servlet.annotation.*;

@WebServlet(name="informer", urlPatterns={"/InfoServlet"},
   initParams={
      @WebInitParam(name="unit", value="1"),
```

```
      @WebInitParam(name="invoke", value="yes")
   })
public class InfoServlet extends HttpServlet {

  // Код сервлет ...
}
```

Анотація `@WebServlet` відповідає елементу конфігураційного файлу `<servlet>` web.xml. Якщо в інструкції немає параметра `name` , то ім'я сервлета збігатиметься з пів- ним ім'ям класу сервлета, включаючи його пакет. Параметр `urlPatterns` можна замінити параметром `value` . Не можна записувати обидва ці параметри в одній інструкції, хоча один з них обов'язково повинен бути присутнім.

Анотація `@WebInitParam` відповідає елементу `<init-param>` . Вона містить ім'я та значення початкового параметра. Сукупність початкових параметрів записується в анотації `@WebServlet` параметром `initParams` , в фігурних дужках.

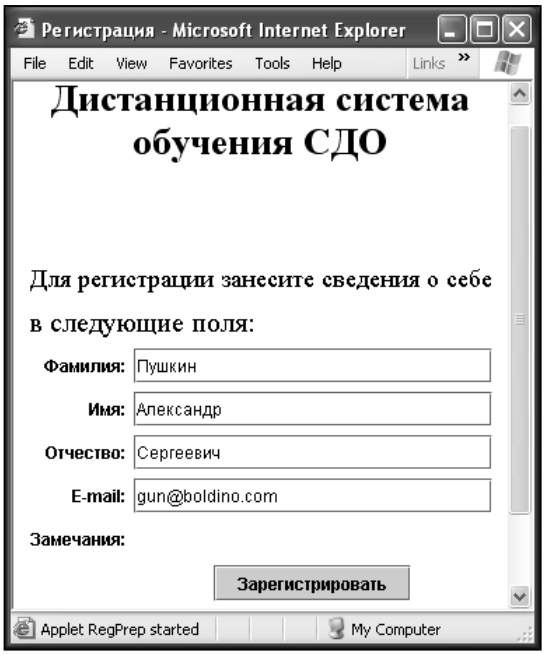
приклад сервлет класу *HttpServlet*

Наведемо приклад сервлету, здійснюючого реєстрацію клієнта Web-прило- ня - деякої системи дистанційного навчання (СДО). Сервлет `RegPrepServlet` приймає запит від HTML-форми, з'єднується з базою даних, заносить до неї напів- ну інформацію та надсилає клієнту підтвердження реєстрації у вигляді країни- ци HTML, що містить форму для вибору навчального курсу. У лістингу 26.4 наведено HTML-форма реєстрації клієнта. Її вид показаний на Мал. 26.2.

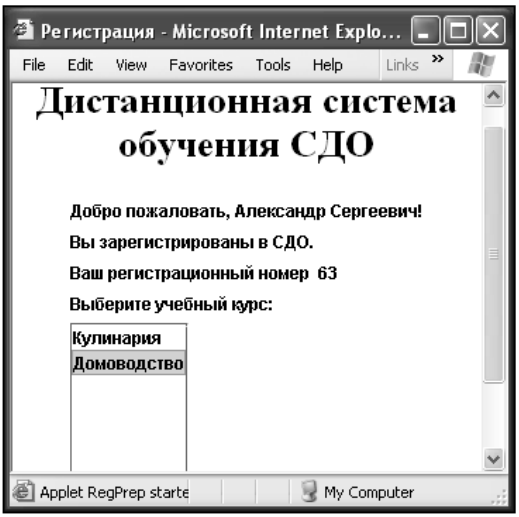
Листинг 26.4. Форма регистрации клиента СДО

```
<html><head>
  <title>Реєстрація</title>
  <META http-equiv=Content-Type
        content="text/html; charset=windows-1251">
  </head>
  <body><h2 align="center">Дистанційна система навчання СДН</h2>
    <p>Для реєстрації занесіть відомості про собі в наступні поля:</p>
    <br>
    <form method="POST"
          action=
          "http://some.firm.com:8000/WebAppl/servlet/RegPrepServlet">
      <pre>
          Прізвище: <input type="text" size="40"
    name="surname">Ім'я: <input type="text" size="40"
  name="name"> По-батькові: <input type="text" size="40"
                    name="secname">
E-mail: <input type="text" size="40" name="addr">

              <input type="submit" value="Зареєструвати">
     </pre>
   </form>
  </body>
</html>
```



Мал. 26.2. Сторінка реєстрації



Мал. 26.3. Сторінка підтвердження реєстрації

Форма посилає сервлету `RegPrepServlet` чотири параметри: `surname` , `name` , `secname` та `addr` за HTTP-методом `POST` . Сервлет повинен прийняти їх, обробити і надіслати клієнту результати запиту або зауваження з реєстрації. Код сервлет наведено в листин- ге 26.5. Сторінка підтвердження реєстрації показано на Мал. 26.3.

Листинг 26.5. Сервлет регистрации клиента СДО

```
package myservlets;

import java.io.*;
import java.sql.*;
import java.util.*;import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet()public class RegPrepServlet extends HttpServlet{

   private String driver = "oracle.jdbc.driver.OracleDriver",
         url = "jdbc:oracle:thin:@homexp:1521:SDO",
     user = "sdoadmin", password = "sdoadmin";

private Connection con;

private Підпорядкованийдержавний

pst;

   @Override
   public void init(){
```

```
    try{
        Class.forName(driver);
        con = DriverManager.getConnection(
           url, user, password);
        pst = con.prepareStatement(
                "INSERT INTO students (id, name, address) " +
                "VALUES(reg_seq.NEXTVAL, ?, ?)");
    } catch (Exception e) {
        System.err.println("From init(): " + e);
    }
}

   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse resp){
     doPost(req, resp);   }
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse resp){
       try{
       req.setCharacterEncoding("Cp1251");

           String surname = req.getParameter("surname");
           String name    = req.getParameter("name");
           String secname = req.getParameter("secname");
       String addr    = req.getParameter("addr");

       resp.setContentType("text/html; charset=windows-1251");

       PrintWriter pw = resp.getWriter();

           if (surname.length() * name.length() *
           secname.length() * addr.length() == 0) {

               pw.println("<html><head>");
               pw.println("<title>Продовження реєстрації</title>");
               pw.println("</head><body><h2 align=center> +
                    "Дистанційна система навчання СДН</h2>");
               pw.println("<h3>Зауваження:</h3>");
               pw.println("Заповніть, будь ласка, усі поля.<br>");
               pw.println("</body></html>");
               pw.flush();
               pw.close();
               return;
       }

            String fullname = surname.trim() + " " + name.trim() +
             " " + secname.trim();
           pst.setString(1, fullname);
           pst.setString(2, addr);
           int count = pst.executeUpdate();
       Statement st = con.createStatement();

           ResultSet rs = st.executeQuery(
          "SELECT id, name FROM students ORDER BY id DESC");
```

```
        rs.next();

            int id = rs.getInt(1);
        fullname = rs.getString(2);

        rs.close();

            StringTokenizer sttok = новий
            StringTokenizer(fullname);sttok.nextToken();
            name = sttok.nextToken();
        secname = sttok.nextToken();

            pw.println("<html><head>");
            pw.println("<title>Реєстрація</title>");
            pw.println("</head><body><h2 align=center>" +
            "Дистанційна система навчання СДО</h2>");

            pw.println("Добро просимо, " + name + " " +
                   secname + "!<br>");
            pw.println("Ви зареєстровані в СДО.<br>");
            pw.println("Ваш реєстраційний номер  " + id + "<br>");
        pw.println("Виберіть навчальний курс:<br> ");

        rs = st.executeQuery("SELECT course_name FROM courses");

            pw.println("<form method=post action=" +
               "\"http://homexp:8000/InfoAppl/servlet/CoursesServlet\">");
        pw.println("<select size=5 name=courses>");

            while (rs.next())
           pw.println("<option>" + rs.getString(1));

            pw.println("</option></form></body></html>");
            pw.flush();
        pw.close();

            rs.close();     } catch (Exception
         e) {System.err.println(e);
      }
   }

   @Override
   public void destroy(){
      pst.close();
      con.close();
   }
}
```

У лістингу 26.5 всі запити користуються тим самим з'єднанням з базою даних. При великій кількості одночасних запитів це може знизити виробник. ність системи і навіть перевищити допустиму кількість з'єднань. У такому разі при ініціалізації сервлет треба в методі `init()` створити пул з'єднань з тим, щоб за-

проси брали з'єднання з цього пулу і повертали з'єднання в пул при своєму завиконанні. Ще краще створити цей пул засобами сервера додатків, а методі `init()` тільки звертатися до цьому пулу.

Слід зазначити, що система управління базою даних Oracle, з якою з'єднує- ється сервлет `RegPrepServlet` , є свій сервер додатків Oracle Application Server (OAS) із контейнером сервлетів Apache/JServ або Tomcat. Можна встановити сервлет прямо в Oracle AS і використовувати для з'єднання з базою серверний драйвер JDBC з ім'ям kprb. Це різко підвищить продуктивність сервлету.

Зрозуміло, сервлет може відправляти клієнту не тільки сторінки HTML, але й бродіння, тексти в різних форматах, наприклад PDF, звукові файли, коротше кажучи, дані будь-яких MIME-типів. У лістингу 26.6 наведено приклад сервлета, що дозволяє клієнту переглядати зображення типу GIF та JPEG у каталозі, заданому початковим параметром сервлету.

Листинг 26.6. Сервлет, отправляющий изображения клиенту

```
package myservlets;

import java.io.*;
import java.util.*;
import javax.servlet.*;import javax.servlet.http.*;import javax.servlet.annotation.*;


@WebServlet()

public class ImageServlet extends HttpServlet{

   Vector imFiles = new Vector();   int curIndex;

   @Override
   public void init() throws ServletException{
        File imDir = null;
    String imDirName = getInitParameter("imagedir");

       if (imDirName != null)
       imDir = new File(imDirName);

       if ((imDir != null) && imDir.exists() && imDir.isDirectory()){
       String[] files = imDir.list();

           for (int i = 0; i < files.length; i++)
              if (files[i].endsWith(".jpg") ||
              files[i].endsWith(".gif")){

                  File curFile = new File(imDir, files[i]);
                  imFiles.addElement(curFile);
         }

        }else log("Cannot find image dir: " + imDirName);
}
```

```
   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
              throws ServletException, IOException{
  int len = imFiles.size();

     if (Len > 0) {
        File curFile = (File)imFiles.elementAt(curIndex);
        String fileName = curFile.getName();
        ServletContext ctxt = getServletConfig().getServletContext();
    String ctype = ctxt.getMimeType(fileName);

        if (ctype == null) ctype = fileName.endsWith(".jpg") ?
             "image/jpeg" : "image/gif";

    resp.setContentType(ctype);

        try{
           BufferedInputStream bis = новий
                BufferedInputStream(new
                FileInputStream(curFile));
      OutputStream os = resp.getOutputStream();

           int cur = 0;
       while ((cur = bis.read()) != -1) os.write(cur);

           os.close();
       bis.close();

        } catch (FileNotFoundException e) {
           resp.sendError(HttpServletResponse.SC_NOT_FOUND);
        } catch (Exception e) {
           resp.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE);
        }
    curIndex= (curIndex + 1) % len;

      }else resp.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE); }
  public long getLastModified(){
      return System.currentTimeMillis();
  }
}
```

Сеанс зв'язку з сервлетом

Сервер HTTP не зберігає інформацію про клієнта, який зв'язався з ним. Хоча TCP-з'єднання може зберігатися аж до його явного закриття (persistent HTTP connection) і за цей час можна передати кілька запитів та відповідей, протокол HTTP не припускає засобів збереження інформації про клієнта.

багато завдання, розв'язувані Web-додатками, вимагають знання відомостей про клієнта. На- Наприклад, клієнт електронного магазину може зробити кілька замовлень протягом дня або навіть кількох днів. Сервер повинен знати, в чию "кошик" складати замовлені покупки. Ще приклад. У програмі лістингу 26.5 зав'язується діалог. Клієнт реєструй-

рується і посилає відомості про собі на сервер З ДО. Сервер пропонує клієнту підібрати собі навчальний курс. Клієнт обирає курс і посилає його назва серверу. Сервер долдружин занести ім'я курсу в облікову картку клієнта, а для цього йому треба знати регістра-ційний номер клієнта.

Для отримання відомостей про клієнта, що надіслав запит, доводиться застосовувати спокус- ні засоби, що не входять до протоколу HTTP. Найчастіше використовуються три засоби: cookie, параметр в рядку URL і приховане поле в HTML-форма.

Перший засіб – cookie – діє так. Отримавши перший запит, сервер складає заголовок відповіді `Set-Cookie` , в який заносить пару "ім'я = значення", зазвичай це ідентифікатор клієнта, а також діапазон URL, для якого діє cookie, термін зберігання цих відомостей та іншу інформацію. Браузер, отримавши відповідь з таким заговправно, створює невеликий, розміром не більше 4 Кбайт, cookie-файл з цими відомостями- ми і зберігає його в каталозі. Надсилаючи наступні запити, браузер шукає у себе відповідний cookie-файл і заносить в Заголовок `Cookie` запиту пару "ім'я = значення". Сервер з цього заголовку "дізнається" клієнта.

У пакеті `javax.servlet.http` є клас `Cookie` , методи якого забезпечують роботу з cookie. Об'єкт цього класу створюється конструктором

```
public Cookie(String name, String value);
```

Методи `set Xxx ()` дозволяють додати в об'єкт інші відомості, а методи `get Xxx ()` - прочитати їх.

Після створення і формування об'єкта cookie цей об'єкт встановлюється в заголо-вок відповіді методом

```
public void addCookie(Cookie cookie);
```

інтерфейсу `HttpServletResponse` .

Ось звичайна послідовність дій по створенню cookie і відправці його клієнту:

```
String value = "" + id;
Cookie ck = new Cookie("studentid", value); ck.setMaxAge (60 *
60 * 24 * 183);              // Cookie буде існувати півроку
resp.addCookie(ck);
```

Прочитати cookie з запиту клієнта можна, можливо методом

```
public Cookie[] getCookies();
```

інтерфейсу `HttpServletRequest` .

Ось звичайна послідовність дій по читання cookie:

```
int id = 0;
Cookie[] cks = req.getCookies();

if (cks != null)
   for (int i = 0; i < cks.length; i++)
      if (cks[i].getName().equals("studentid")){
         id = Integer.parseInt(cks[i].getValue());
         break;
      }
```

Використовувати cookie зручно, але біда в тому, що багато клієнтів забороняють запис cookie-файлів, справедливо вважаючи, що не можна записувати що-небудь на диск без відомих. ма господаря.

Другий засіб - параметр у рядку URL (rewriting URL) - просто записує в пер- вою рядок запиту після імені ресурсу ідентифікатор клієнта, наприклад:

```
GET /some.com/InfoAppl/index.html?jsessionid=12345678 HTTP/1.1
```

Це теж хороший спосіб, але клієнт може сам формувати рядок запиту, наприклад заходів, просто набираючи її в поле адреси браузера і забуваючи про ідентифікатор.

Третє засіб - приховане поле HTML-форми - це поле

```
<input type="hidden" name="studentid" value="12345678">
```

до якого можна записати ідентифікатор клієнта. Для застосування цього засобу треба на кожній сторінці HTML створювати форму.

Отже, якогось одного універсального засоби, для того щоб створити сеанс зв'язку з Web-сервером, ні. У пакет `javax.servlet.http` внесені інтерфейси та класи для об- полегшення розпізнавання клієнта. Вони автоматично перемикаються з одного засобу на інше. Якщо заборонені cookies, то формується ідентифікатор клієнта у рядку URL і т.д.

Основу засобів створення сеансу зв'язку з клієнтом складає інтерфейс `HttpSession` . Об'єкт типу `HttpSession` формується контейнером сервлета при отриманні запиту, а отримати або створити його можна, можливо методом

```
public HttpSession getSession(boolean create);
```

описаним в інтерфейсі `HttpServletRequest` . Метод повертає об'єкт типу `HttpSession` , якщо вона існує. Якщо ж такий об'єкт відсутній, то поведінка методу залежить від значення аргументу `create` - якщо він дорівнює `false` , то метод повертає `null` , якщо `true` - створює новий об'єкт.

Другий метод того ж інтерфейсу

```
public HttpSession getSession();
```

еквівалентний `getSession(true)` .

Логічним методом

```
public boolean isNew();
```

інтерфейсу `HttpSession` можна дізнатися, чи це новий, щойно створений сеанс (`true`) або триває, вже запитаний клієнтом (`False`).

У сеансі відзначається час його створення та час останнього запиту, які можна отримати методами

```
public long getCreationTime();
public long getLastAccessedTime();
```

Вони повертають час у мілісекундах, що пройшли з півночі 1 січня 1970 року (дата народження UNIX).

Створюючи сеанс, контейнер дає йому унікальний ідентифікатор. Метод

```
public String getId();
```

повертає ідентифікатор сеансу в вигляді рядки.

У сеансу можуть бути атрибути, якими можуть виступати будь-які об'єкти Java. Атрибут задається методом

```
public void setAttribute(String name, Object value);
```

Отримати імена і значення атрибутів можна методами

```
public Enumeration getAttributeNames();
public Object getAttribute(String name);
```

Атрибути - зручний засіб зберігання об'єктів, які повинні існувати на протягом сеансу.

Атрибут видаляється методом

```
public void removeAttribute(String name);
```

Контейнер слідкує за подіями, які відбуваються під час сеансу. Створення або видалення ня атрибута, зміна його значення - події класу `HttpSessionBindingEvent` . Цей клас є підкласом класу `HttpSessionEvent` , екземпляр якого створюється при будь-якій зміні в активних сеансах Web-додатки - створенні сеансу, припинення ні сеансу, закінченні терміну очікування запиту.

Сеанс завершується методом `invalidate()` або по закінченні часу очікування черг- ного запиту. Це час, у секундах, задається методом

```
public void setMaxInactiveInterval(int secs);
```

Сервлет може дізнатися призначене час очікування методом

```
public int getMaxInactiveInterval();
```

Фільтри

Суть роботи сервлет полягає в обробці отриманого з об'єкта типу `ServletRequest` запиту та формуванні відповіді у вигляді об'єкта типу `ServletResponse` . По- путно сервлет може виконати масу роботи, створюючи об'єкти, звертаючись до їх мето- жінок, завантажуючи файли, з'єднуючись з базами даних. Ці дії ускладнюють спо- але просту і чітку структуру сервлета. Щоб надати стрункість і впорядкований- ність сервлету, можна організувати ланцюжок фільтрів - об'єктів, послідовно пропускають через себе інформацію, що йде від запиту до відповіді, і перетворюю- щих її.

Зручність фільтрів полягає ще й у тому, що один фільтр може використовуватись декількома сервлетами і навіть усіма ресурсами Web-програми. Розробник може підготувати набір фільтрів на Усе випадки життя та застосовувати їх в своїх сервлетах.

При роботі з фільтрами відразу створюється ланцюжок фільтрів, навіть якщо в неї входить все- го один фільтр. Вся робота з ланцюжками фільтрів описується інтерфейсами `Filter` , `FilterChain` та `FilterConfig` . Інтерфейс `Filter` реалізується розробником програми, інші інтерфейси має реалізувати контейнер сервлетів.

Кожен фільтр у ланцюжку - це об'єкт типу `Filter` . Структура цього об'єкта нагадує. нає структуру сервлету. Інтерфейс `Filter` описує три методу:

```
public void init(FilterConfig cong);
public void doFilter(ServletRequest req, ServletResponse resp,
                     FilterChain chain);
public void destroy();
```

Як видно з цих описів, фільтр може виконати початкові дії методом `init()` , причому йому передається створений контейнером об'єкт типу `FilterConfig` , кото- рий дуже схожий на об'єкт типу `ServletConfig` . Він також містить початкові парамет- ри, які можна отримати методами

```
public Enumeration getInitParameterNames();
public String getInitParameter(String name);
```

У нього теж є ім'я, яке дається йому при установці фільтра в контейнер та записи. ється в конфігураційний файл web.xml в елемент `<filter-name>` . Ім'я класу-фільтра можна, можливо отримати методом

```
public String getFilterName();
```

Зрештою, він теж повертає посилання на контекст методом

```
public ServletContext getServletContext(String name);
```

Вся фільтрація виконується методом `doFilter()` , який отримує об'єкти `req` та `resp` , змінює їх і передає керування наступному фільтру в ланцюжку за допомогою аргу- мента `chain` .

Організація ланцюжка дістається частку контейнера, інтерфейс `FilterChain` описує тільки один метод

```
public void doFilter(ServletRequest req, ServletResponse resp);
```

Щоб передати керування фільтром, що йде в ланцюжку, фільтр повинен просто звернутися до цьому методом, передавши йому змінені об'єкти `req` і `resp` .

Наведемо приклад фільтра. Російськомовному програмісту постійно доводиться ду- мати про правильне кодування кирилиці. Параметри запиту йдуть від браузера частіше всього в MIME-типі `application/x-www-form-urlencoded` , що використовує байтову коди- ровку, прийняту за замовчуванням на машині клієнта. Це кодування має вказуватисяв заголовку `Content-Type` , наприклад:

```
Content-Type: application/x-www-form-urlencoded;charset=windows-1251
```

Але, як правило, браузер не надсилає цей заголовок Web-серверу. В такому випадку встає завдання визначити кодування параметрів запиту і заслати її в метод `setCharacterEncoding(String)` , щоб метод `getParameter(String)` правильно переклав ня параметра в Unicode. Це завдання має вирішувати метод `getCharacterEncoding()` , але він частіше всього реалізований так, що просто бере кодування з заголовка `Content-Type` . Лис- тинг 26.7 показує схему такої реалізації в одній із колишніх версій контейнера сервлетів Tomcat. Оцініть якість кодування та подивіться, чому, використовуючи цю версію Tomcat, між словом `"charset"` і знайомий рівності не можна залишати прогалини.

До речі кажучи, деякі контейнери сервлетів не сприймають кодування, якщозалишені прогалини між точкою з комою і словом `"charset"` .

Листинг 26.7. Фильтр определения кодировки параметров запроса

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

@WebFilter(
      urlPatterns={"/*"}, servletNames={""}
      initParams={ @WebInitParam(name="simpleParam", value="paramValue") }
)
public class SetCharEncFilter implements Filter{
 protected String enc;
 protected FilterConfig fc;
 @Override
 public void init(FilterConfig conf) throws ServletException{
       fc = conf;
       enc = conf.getInitParameter("encoding");
 }

 @Override

    public void doFilter(ServletRequest req,
                         ServletResponse resp,
                         FilterChain chain)
             throws IOException, ServletException{

       String encoding = selectEncoding(req);
       if (encoding! = null)
       req.setCharacterEncoding(encoding);

       chain.doFilter(req, resp);
 }

    protected String selectEncoding(ServletRequest req){
        String charEncoding =
            getCharsetFromContentType(req.getContentType());
        return (charEncoding == null) ? enc : charEncoding;
  }

// від org.apache.tomcat.util.RequestUtil.java

  public static String getCharsetFromContentType(String type){
    if (type == null) {
      return null;
    }
    int semi = type.indexOf(";");
    if (semi == -1) {
       return null;
    }
```

```
    String afterSemi = type.substring(semi + 1);
    int charsetLocation = afterSemi.indexOf("charset=");
    if (charsetLocation == -1) {
       return null;
    }
    String afterCharset = afterSemi.substring(charsetLocation + 8);
    String encoding = afterCharset.trim();
    return encoding;  }
  @Override
  public void destroy(){
    enc = null;
    fc = null;
  }
}
```

Фільтр класу `SetCharEncFilter` , описаний у лістингу 26.7, дуже простий. Він витягує із запиту `req` заголовок `Content-Type` методом `getContentType()` . Якщо такий заголовок є, то він намагається витягти з нього кодування. Якщо це не вдається, то бере кодиров- ку з свого початкового параметра `"encoding"` . Потім він заносить кодування в відповідь `resp`методом `setCharacterEncoding(String)` і передає керування наступному фільтру.

Будь-який розробник, який бажає поліпшити визначення кодування, може переопред- лити метод `selectEncoding()` , витягуючи інформацію з інших заголовків, наприклад: `User-Agent` ,`Accept-Language` ,`Accept-Charset` ,`Content-Language` .

У складніших випадках знадобиться розширити об'єкти `req` і `resp` . Ось тут і при- годяться класи `HttpServletRequestWrapper` та `HttpServletResponseWrapper` . Додаткові властивості запиту та відповіді можна занести до розширення цих класів-оболонок та вико- користувати їх в фільтрі за такою схемою.

```
public class MyRequestHandler extends HttpServletRequestWrapper{

   public MyRequestHandler(HttpServletRequest req) {
      super(req);
      // ...
   }
   // ...
}

public class MyResponseHandler extends HttpServletResponseWrapper{

   public MyResponseHandler(HttpServletResponse resp) {
      super(resp);
      // ...
   }
   // ...
}@WebFilter(urlPatterns={"/*"})

public class MyFilter implements Filter{

   private MyRequestHandler mreq;
   private MyResponseHandler mresp;
```

```
   @Override
   public void init(FilterConfig conf) {
      // ...
   }

  @Override
  public void doFilter(ServletRequest req,
                       ServletResponse resp,
                       FilterChain chain) {
      mreq = New MyRequestHandler((HttpServletRequest)req);
      mresp = new MyResponseHandler((HttpServletResponse)resp);

      // Дії до переходу до наступному фільтру.

      chain.doFilter(mreq, mresp);
      // Дії після повернення з сервлет і фільтрів.
   }

   @Override
   public void destroy(){
      mreq = null;
      mresp = null;
   }
}
```

Після того як клас-фільтр написаний і скомпільований, його треба встановити в контейнер і приписати (map) до одного або кількох сервлетів. Це виконується утилітою установки або засобами IDE, в яких вказується ім'я фільтра, його початкові параметри та сервлет, до якого приписується фільтр. Утиліта установки заносить свеения про фільтр в конфігураційний файл web.xml елемент `<filter>` . Це можна зробити і вручну. Приписка фільтра до сервлета відзначається всередині елемента `<filter-mapping>` парою вкладених елементів `<filter-name>` і `<Servlet-name>` . Наприклад:

```
<filter-mapping>
   <filter-name>MyFilter</filter-name>
   <servlet-name>RegServlet</servlet-name>
</filter-mapping>
```

Фільтр можна приписати як сервлетам, а й іншим ресурсам. Для цього запускається елемент `<url-pattern>` , наприклад після

```
<filter-mapping>
   <filter-name>MyFilter</filter-name>
   <url-pattern>*.html</url-pattern>
</filter-mapping>
```

фільтр буде застосований до всім викликам документів HTML.

Порядок фільтрів в ланцюжку відповідає порядку елементів `<filter-mapping>` у конфігураційному файлі web.xml. При зверненні клієнта до сервлет контейнер спочатку шукає фільтри і послідовно виконує їх, а потім запускає сервлет. Після роботи сервлета його відповідь проходить ланцюжок фільтрів у зворотному порядку.

Звернення до іншим ресурсів

У деяких випадках недостатньо вставити в сервлет фільтр або навіть ланцюжок фільтрів, а треба звернутися до іншого сервлета, сторінки JSP, документа HTML, XML чи іншого ресурсу. Якщо необхідний ресурс перебуває у тому контексті, як і серв-років, який його викликає, то для отримання ресурсу слід звернутися до методом

```
public RequestDispatcher getRequestDispatcher(String path);
```

описаному в інтерфейсі `ServletRequest` . Тут `path` - це шлях до ресурсу щодоконтексту. Наприклад:

```
RequestDispatcher rd = req.getRequestDispatcher("CourseServlet");
```

Якщо ж ресурс знаходиться в іншому контексті, потрібно спочатку отримати контекст методом

```
public ServletContext getContext(String uripath);
```

інтерфейсу `ServletContext` , а потім скористатися методом

```
public RequestDispatcher getRequestDispatcher(String path);
```

інтерфейсу `ServletContext` . Тут шлях `path` повинен бути абсолютним, т. е. починатися з похилій риси `/` . Наприклад:

```
RequestDispatcher rd = conf.getServletContext().
   getContext("/product").
   getRequestDispatcher("/product/servlet/CourseServlet");
```

Якщо необхідний ресурс - сервлет, поміщений у контекст під своїм ім'ям, то для його отримання можна звернутися до методу

```
public RequestDispatcher getNamedDispatcher(String name);
```

інтерфейсу `ServletContext` .

Усе три методу повертають `null` , якщо ресурс недоступний або сервер не реалізує інтерфейс `RequestDispatcher` .

Як видно з описи методів, до ресурсу можна, можливо звернутися тільки через об'єкт типу
`RequestDispatcher` . Цей об'єкт пропонує два методу звернення до ресурсу.Перший

метод,

```
public void forward(ServletRequest req, ServletResponse resp);
```

просто передає управління іншому ресурсу, надавши йому свої аргументи `req` і `resp` . Викликаючий сервлет виконує попередню обробку об'єктів `req` та `resp` і передає їх викликаному сервлету або іншому ресурсу, який остаточно фор- мує відповідь `resp` і відправляє його клієнту або знову-таки викликає інший ресурс. Наприклад:

```
if (rd != null) rd.forward(req, resp);
else resp.sendError(HttpServletResponse.SC_NO_CONTENT);
```

Сервлет, що викликає, не повинен виконувати будь-яку відправку клієнту до звернення. ня до методом `forward()` , інакше буде викинуто виняток класу `IlegallegalException` .

Якщо ж викликаючий сервлет вже щось відправляв клієнту, слід звернутися до другому методом,

```
public void include(ServletRequest req, ServletResponse resp);
```

Цей метод викликає ресурс, який на підставі об'єкта `req` може змінити тіло об'єкта `resp` . Але викликаний ресурс не може змінити заголовки та код відповіді об'єкта `resp` . Це природне обмеження, оскільки викликаючий сервлет міг уже відпра- ти заголовки клієнту. Спроба викликаного ресурсу змінити заголовок буде просто проігнорована контейнером. Можна, можливо сказати, що метод `include()` виконує таку ж роботу, як вставки на стороні сервера SSI (Server Side Include).

Після виконання методу `include()` управління повертається в сервлет.

Асинхронне виконання запитів

Якщо кілька запитів надходять одночасно, контейнер сервлетів створює під- процеси, виконують метод `service()` для кожного запиту. Кількість підпроцес- сов може виявитися занадто велике, а час їх роботи може затягнутися, особливо якщо сервлет очікує отримання даних з бази даних або файлу або закінчення тривалої обробки інформації У цьому випадку необхідно забезпечити швидкість роботи сервлету. Цього можна, можливо досягти, виконуючи запити асинхронно: сервлет приймає запит, передає його обробку асинхронному підпроцесу і завершує роботу, не чекаючи відповіді цього підпроцесу. Асинхронний підпроцес сам фор- мує відповідь або передає управління іншому сервлету в результаті виконання своєї роботи.

Така можливість надано сервлетам і фільтрів, починаючи з Версія 3.0.

Сервлет або фільтр, що виконує асинхронну роботу, відзначається в конфігурації. вінному файлі web.xml елементом `<async-supported>` зі значенням `true` або в анотаціях `@WebServlet` , `@WebFilter` параметром `asyncSupported` , наприклад,

```
@WebServlet(value="/MyAsyncServlet", asyncSupported="true")
```

Можливість виконання асинхронних дій можна, можливо перевірити методом `isAsyncSupported()` .

Асинхронна обробка запиту починається зверненням до методу `startAsync()` об'єкта типу `ServletRequest` , до якого передаються посилання на запит `ServletRequest` та відповідь `ServletResponse` . Цей метод повертає посилання на об'єкт типу `AsyncContext` , методи ко- торого дозволяють простежити за роботою асинхронного методу Перевірити, що запит обробляється асинхронно, можна, можливо логічним методом `isAsyncStarted()` .

Після виконання методу `startAsync()` обробка запиту буде завершена не по вікон- ні роботи методу `service()` , як звичайно, а методом `complete()` об'єкта `AsyncContext` або після часу, заданого попередньо методом `setTimeout()` . Крім того, можна створити об'єкт типу `AsyncListener` та його методи відстежують етапи асин- хронною обробки.

Інтерфейс `AsyncListener` описує чотири методу: `onStartAsync()` , `onError()` , `onComplete()` та `onTimeout()` . Реалізувавши ці методи, можна відреагувати на наступ відповідних чотирьох подій.

Асинхронну роботу природно виконувати в окремому підпроцес.

У лістингу 26.8 наведено типовий приклад асинхронного сервлету. Звернутися до ньому можна, можливо приблизно так:

http://localhost:8080/MyAsyncApp/AsyncServlet?id=1

Після набору цього адреси в адресною рядку браузера, зачекайте 10 секунд, і ви увидити відповідь сервлету.

Листинг 26.8. Асинхронный сервлет

```
package mypack;

import java.io.*;
import java.util.concurrent.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;
import java.util.Date;

@WebServlet(urlPatterns = "/AsyncServlet", asyncSupported=true)
public class AsyncServlet extends HttpServlet {

 @Override
 protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
     throws ServletException, IOException {
   if (!request.isAsyncSupported()){
    response.getWriter().println(
       "Asynchronous processing is not supported");
    return;
   }
   AsyncContext asyncCtx = request.startAsync();
   asyncCtx.addListener(new MyAsyncListener());
   asyncCtx.setTimeout(20000);

   Executor executor = new ThreadPoolExecutor(10, 10, 50000L,
    TimeUnit.MILLISECONDS,
    New LinkedBlockingQueue<Runnable>(100));
   executor.execute(new AsyncProcessor(asyncCtx));
 }
}

class AsyncProcessor implements Runnable {
 private AsyncContext asyncContext;

 public AsyncProcessor(AsyncContext asyncContext) {
   this.asyncContext = asyncContext;
 }
```

```
 @Override
 public void run() {
   String reqId = asyncContext.getRequest().getParameter("id");
   if (null == reqId || reqId.length() == 0) reqId = "unknown";
   Date before = new Date();

   String result = longStandingProcess(reqId);

   String resp = "Request id: " + reqId +
     "<br/> Started at: " + before.toString() +
     ".<br/> Completed at: " + result + ".<br/>";
   asyncContext.getResponse().setContentType("text/html");

   try {
     PrintWriter out = asyncContext.getResponse().getWriter();
     out.println(resp);
   } catch (Exception e) {
     System.out.println(e.getMessage() + ": " + e);
   }
   asyncContext.complete();
 }
 public String longStandingProcess(String reqId) {
   try {
     Thread.sleep(10000);
   } catch (InterruptedException ie) {
     System.out.println("Request: " + reqId +
         ", " + ie.getMessage() + ": " + ie);
   }
   return new Date().toString();
 }
}

@WebListener
public class MyAsyncListener implements AsyncListener {
 public MyAsyncListener() {  }

 public void onComplete(AsyncEvent ae) {
   System.out.println("AsyncListener: onComplete for request: " +
      ae.getAsyncContext().getRequest().getParameter("id"));
 }
 public void onTimeout(AsyncEvent ae) {
   System.out.println("AsyncListener: onTimeout for request: " +
     ae.getAsyncContext().getRequest().getParameter("id"));
 }
 public void onError(AsyncEvent ae) {
   System.out.println("AsyncListener: onError for request: " +
     ae.getAsyncContext().getRequest().getParameter("id"));
 }
```

```
 public void onStartAsync(AsyncEvent ae) {
    System.out.println("AsyncListener: onStartAsync");
 }
}
```

У цьому найпростішим прикладі тривалий процес - це просто затримка на 10 секунд у методі `longstandingProcess()` . Метод `longStandingProcess()` викликається в рамках підпроцесу, що виконує метод `run()` класу `AsyncProcessor` . Після виконання методу `longStandingProcess()` формується відповідь клієнту - рядок `resp` - і відправляється у вихідний потік, після чого асинхронне виконання завершується методом `complete()` . Роль методу `doGet()` полягає тільки в запуску нового підпроцесу методом `execute()` , після чого метод `doGet()` завершується, не формуючи ніякого відповіді.

У складнішій ситуації, коли запити йдуть один за одним, створюється черга за- просів, що виконуються асинхронно. Черга звільняється у міру виконання запро- сов. Такий приклад, названий AsyncRequest, наведено в стандартною постачання Java EE6 SDK.

Запитання для самоперевірки

1. Яка різниця між HTTP-сервером і Web-сервером?
2. Що таке сервер додатків?
3. Що таке сервлет?
4. Що таке контейнер сервлетів?
5. Що означає процедура установки сервлету?
6. Може чи сервлет Відправити клієнту не сторінку HTML, а інший документ?
7. Може чи сервлет обробляти паралельно кілька запитів?
8. Можуть чи сервлети, встановлені в один контейнер, обмінюватися інформацією?
9. Може чи сервлет встановити сеанс зв'язку з клієнтом?

ГЛАВА 27

Сторінки JSP

Як видно з наведених у попередньому розділі лістингів, більшу частину сервлету займають оператори висновку в вихідний потік тегів HTML, формують резуль- тат - сторінку HTML. Ці оператори майже без змін повторюються з сервлета в сервлет. Виникла ідея не записувати теги HTML в операторах Java, а, навпаки, запи- вати оператори Java у коді HTML за допомогою тегів спеціального вигляду. Потім обра- ботати отриману сторінку препроцесором, що розпізнає всі теги і перетворюю- щим їх в код сервлету Java.

Так вийшла мова розміток JSP (JavaServer Pages), що розширює мову HTML тегами виду `<% имя_тега атрибути %>` . За допомогою тегів можна не тільки записати описи, вити і оператори Java, але і вставити в сторінку файл з текстом або зображенням, викликати об'єкт Java, компонент JavaBean або компонент EJB.

Програміст може навіть розширити мова JSP своїми власними, як кажуть, *користувальницькими тегами* (custom tags), які зараз частіше називають *розширеннями тегів* (Tag extensions).

У лістингу 27.1 наведено приклад JSP-сторінки "Hello, World!" з поточною датою.

Листинг 27.1. Простейшая страница JSP

```
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page contentType="text/html;charset=windows-1251" %>
<%@ page import="java.util.*, java.text.*" %>
<html><head><title> Найпростіша сторінка JSP </title>
<META  http-equiv=Content-Type
      content="text/html; charset=windows-1251">
</head><body>
Hello, World!<p>
Сьогодні <%= getFormattedDate() %>
</body></html>
<%!
  String getFormattedDate(){
    SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy hh:mm");
    return sdf.format(new Date());
  }
%>
```

Контейнер сервлетів розширили препроцесором, що переводить запис, подібну тингу 27.1, в сервлет. У контейнері сервлетів Tomcat такий препроцесор називається Jasper. Препроцесор спрацьовує автоматично при першому зверненні до сторінки JSP. Отриманий в результаті його роботи сервлет відразу компілюється і виконує- ся. Відкомпільований сервлет потім зберігається в контейнері, так само як і все сервле- ти, та виконується за наступних викликах сторінки JSP.

Для сервлетів та сторінок JSP придумано загальну назву - *Web-компоненти* (Web Components). Контейнер сервлетів, розширений засобами роботи з JSP, називається *Web-контейнер* (Web Container або JSP Container). Додаток, складений з сервлетів, сторінок JSP, аплетів, документів HTML і XML, зображень та інших документів, що відносяться до додатку, називається *Web-додатком* (Web Application).

Весь статичний текст HTML, званий в документації JSP *шаблоном HTML* (template HTML), відразу прямує у вихідний потік. Вихідний потік сторінки бу- феризується. Буферизацію забезпечує клас `JspWriter` , що розширює клас `Writer` . Розмір буфера по замовчуванням 8 Кбайт, його можна, можливо змінити атрибутом `buffer` тега `<% @ page>` . Наявність буфера дозволяє заносити заголовки відповіді у вихідний потік упереміш . з текстом, що виводиться. У буфері заголовки будуть розміщені перед текстом.

Таким чином, програмісту достатньо написати сторінку JSP, записати її у файл з розширенням jsp і встановити файл у контейнер, подібно до сторінки HTML, не переймаючись про компіляції. При установці можна, можливо поставити початкові параметри сторінки JSP такж, як і початкові параметри сервлету.

У той час як створювалася і розвивалася мова JSP, широкого поширення набула мову XML, яку ми розглянемо у *розділі 28* . Для сумісності з ним усі теги JSP продубльовані елементами XML з іменами з простору імен **http://java.sun.com/JSP/Page** , який отримує, як правило, префікс `jsp:` . Теги JSP, запи- санні в формі XML, зараз називають *діями* (actions) JSP.

Наприклад, сторінку JSP, представлену в лістингу 27.1, можна написати у формі XML так як це зроблено в лістингу 27.2.

Листинг 27.2. Простейшая страница JSP в форме XML

```
<jsp:root xmlns:jsp=" http://java.sun.com/JSP/Page " version="2.0">
  <jsp:directive.page
     contentType="text/html;charset=windows-1251" />
  <jsp:directive.page
     import="java.util.Date, java.text.SimpleDateFormat" />

  <jsp:text>
    <![CDATA[
    <html><head><title> Найпростіша сторінка JSP </title>
    <META  http-equiv=Content-Type
          content="text/html; charset=windows-1251">
    </head><body>
       Hello, World!<p>
       Сьогодні ]]>
  </jsp:text>
```

```
  <jsp:expression>getFormattedDate()</jsp:expression>
  <jsp:text>
     <![CDATA[
     </body></html> ]]>
  </jsp:text>
  <jsp:declaration>
    String getFormattedDate(){
      SimpleDateFormat sdf = new SimpleDateFormat("dd-MMMM-yyyy hh:mm");
      return sdf.format(new Date());
    }
  </jsp:declaration>
</jsp:root>
```

Файл зі сторінкою JSP, записаною у формі XML, зазвичай отримує розширення jspx.

Деякі теги - `<jsp:forward>` , `<jsp:include>` , `<jsp:plugin>` , `<jsp:useBean>` , `<jsp:getProperty>` , `<jsp:setProperty>` - Завжди записувалися в формі XML.

Ви вже знаєте, що мова XML розрізняє регістр літер. Теги XML записуються як правило, малими літерами. Значення атрибутів тегів обов'язково записуються в лапках або апострофах. У більшості елементів XML є *відкриває тег* (start tag) з необов'язковими атрибутами, *тіло* (body) та *закриваючий тег* (end tag), вони мають вигляд:

```
<jsp: тег атрибути_тега >
   Тіло елемента або вкладені елементи XML
</jsp: тег >
```

Тіло елемента може бути порожнім, тоді елемент виглядає так:

```
<jsp: тег атрибути_тега ></jsp: тег >
```

> ***УВАГА !*** _
> Якщо між відкриваючим і що закриває тегом є хоча б один пробіл, то тіло еле-мента вже не порожньо.

Нарешті, тіло може бути відсутнім, тоді закриваючий тег не пишеться, а у відкритті ваючим тезі перед закриває кутовий дужкою ставиться похила характеристика:

```
<jsp: тег атрибути_тега />
```

Якщо XML-елемент має тіло, то його відкриваючий тег закінчується просто кутовий. дужкою, без похилої межі.

Специфікація "JavaServer Pages Specification" не рекомендує змішувати в одному документ теги вид JSP з елементами XML. Сторінку HTML, до якої вставлені теги виду `<%...%>` , вона офіційно називає *сторінкою JSP* (JSP Page), а документ XML з тегами виду `<jsp:.../>` називає *документом JSP* (JSP Document). Файл зі сторінкою JSP зазвичай отримує ім'я з розширенням jsp, а файл з документом JSP - ім'я з розширенням ренією jspx.

Документ JSP проходить ще одну стадію попередньої обробки, де він наводиться у повне відповідність із синтаксисом XML. Це приведення включає в себе запис кореневого елемента з простором імен **http://java.sun.com/JSP/Page** і

вставку елементів CDATA . Після приведення виходить документ XML, офіційно званий *поданням XML* (XML View).

Стандартні дії (теги) JSP

Набір стандартних тегів JSP досить простий. При їх написанні слід пам'ятати три правила:

- ❒ мова JSP розрізняє регістр літер, як і мова Java;
- ❒ при записі атрибутів, після знака рівності, що відокремлює ім'я атрибута від його зна- чення, не можна залишати прогалини;
- ❒ значення атрибутів можна заносити у лапки, а й у апострофи.

Записуватимемо теги і в старій формі JSP, і в новій формі елементів XML.

Коментар на сторінках JSP відзначається тегом

```
<%-- Коментар --%>
```

або тегом

```
<!-- Коментар -->
```

Коментар першого виду не передається клієнту. Все, що написано всередині нього, не обробляється препроцесором. Коментар другого виду переноситься у форму- мою HTML-сторінку. Всі JSP-теги, записані всередині такого коментаря, інтерпре- ні.

Оголошення полів та методів Java записуються у тезі

```
<%! Оголошення %>
<jsp:declaration> Оголошення </jsp:declaration>
```

Після обробки препроцесором вони будуть полями та методами сервлету.

Вираз Java записується в тезі

```
<%= Вираз %>
<jsp:expression> Вираз </jsp:expression>
```

Вираз обчислюється, результат підставляється місце тега. Врахуйте, що наприкінці вирази не треба ставити крапку з комою, оскільки вираз, що завершується точкою з комою, - Це вже оператор.

Фрагмент коду Java, званий в JSP *скриптлетом* (scriptlet), який може включати чати у собі не тільки оператори, але і визначення, записується в тезі

```
<% Скриптлет %>
<jsp:scriplet> Скриптлет </jsp:scriptlet>
```

Такий фрагмент після обробки препроцесором потрапить у метод _jspService () ного сервлету, що є оболонкою методу service() .

Увімкнення файлу во час компіляції Виготовляється тегом

```
<%@ include file=" URL файлу щодо контексту " %>
<jsp:directive.include file=" URL файлу щодо контексту " />
```

Загальні властивості сторінки JSP задаються тегом

```
<%@ page Атрибути %>
<jsp:directive.page Атрибути />
```

Усі атрибути тут необов'язкові. У лістингах 27.1 і 27.2 вже використані атрибу- ти `contentType` і `import` цього тега. Інші атрибути:

- `pageEncoding="` *Кодування* `"` - по замовчуванням `"ISO 8859-1"` ;
- `extends = "` *Повне ім'я розширюваного класу* `"` - суперклас того класу, в який компілюється сторінка JSP;
- `session="` *true або false* `"` - створювати чи сеанс зв'язку з клієнтом, по замовчуванням `"true"` ;
- `buffer="` *Nkb або none* `"` - розмір вихідного буфера, по замовчуванням `"8kb"` ;
- `autoFlush="` *true або false* `"` - автоматичне очищення буфера після його заповнення, замовчуванням `"true"` ; якщо значення цього атрибуту одно `"false"` , то при переповненнібуфера викидається виняток;
- `isThreadSafe="` *true або false* `"` — одночасний доступ до сторінки кількох кліентів, за замовчуванням `"true"` ; якщо цей атрибут дорівнює `"false"` , то отриманий після компіляції сервлет реалізує застарілий інтерфейс, що не використовується зараз `SingleThreadModel` ;
- `info="` *якийсь текст* `"` - повідомлення, яке можна, можливо буде прочитати методом `getServletInfo()` ;
- `errorPage="` *URL щодо контексту* `"` - адреса сторінки JSP, якої посилається виняток і яка показує повідомлення, посилаються винятком;
- `isErrorPage="` *true або false* `"` — чи може ця сторінка JSP використовувати об'єкт-виняток `exception` , або він буде пересланий інший сторінці, по замовчуванням `"False"` .

Два останні атрибути вимагають роз'яснення. Справа в тому, що якщо сторінка JSP не обробляє виняток, а викидає його далі, то Web-контейнер формує сторінку HTML з повідомленнями про виключення та надсилає її клієнту. Це заважає роботі клієнта. Атрибут `errorPage` дозволяє замість сторінки HTML із повідомленнями передати вбудований об'єкт-виключення `exception` для обробки сторінці JSP, кото- ню попередньо створив розробник. Атрибут `isErrorPage` сторінки-обробника виключення повинен дорівнювати `"true"` .
Нехай, наприклад, є сторінка JSP:

```
<html><body>
<%@ page errorPage="myerrpage.jsp" %>
<%
   String str = <%= request.getParameter("name") %>;
   int n = str.length();
%>
</body></html>
```

Посилання `str` може опинитися рівною `null` тоді метод `length()` викине виняток. Контейнер створить вбудований об'єкт `exception` та передасть його сторінці myerrpage.jsp. Вона може виглядати так:

```
<html><body>
   <%@ page isErrorPage="true" %>
   При виконанні сторінки JSP викинуто виняток:
   <%= exception %>
</body></html>
```

на цьому набір тегів виду `<%...%>` закінчується. Інші теги записуються тількив формі елементів XML.

Увімкнення файлу на етапі виконання або включення результату виконання серв- та, якщо цей результат представлений сторінкою JSP, виконується елементом

```
<jsp:include Атрибути />
```

У цьому тезі може бути один або два атрибут. Обов'язковий атрибут

```
page=" відносний URL або вираз JSP "
```

ставить адреса включається ресурсу. Тут "вираз JSP" записується в формі JSP `<%= вираз %>` або в формі XML

```
<jsp:expression> вираз </jsp:expression>
```

В результаті виразу має бути адреса URL. Найчастіше як виро- ження використовується звернення до методом `getParameter(String)` .

Другий, необов'язковий, атрибут

```
flush="true або false"
```

вказує, чи очищати вихідний буфер перед включенням ресурсу. Його значення по замовчуванням `"False"` .

Друга форма елемента, `include` , містить теги `<jsp:param>` в тілі елемента (Зверніть увага на відсутність похилої межі в кінці відкриває тега):

```
<jsp:include Атрибути >
   Тут записуються теги виду <jsp:param>
</jsp:include>
```

У тілі елемента можна, можливо записувати довільні параметри. Вони мають вигляд

```
<jsp:param name=" ім'я параметра "
           value=" значення параметра або вираз JSP " />
```

"Вираз JSP" записується так само, як і в заголовку тега. Параметри передаються включається ресурсу як його початкові параметри, та їх імена, зрозуміло, повинні збігатися з іменами початкових параметрів ресурсу.

JSP-файл може бути оформлений не повністю, а містити тільки від- слушний фрагмент коду JSP. У такому випадку його ім'я записують зазвичай з розширенням. їм jspf (JSP Fragment).

Наступний стандартний тег `<jsp:element>` дозволяє динамічно визначити який- елемент XML. Він має один обов'язковий атрибут `name` - ім'я створюваного елемента XML. У його тілі можна визначити атрибути створюваного елемента XML з за допомогою елемента `<jsp:attribute>` і тіло створюваного елемента XML за допомогою еле- мента `<jsp:body>` . Наприклад:

```
<jsp:element name="${myElem}"
            xmlns:jsp=" http://java.sun.com/JSP/Page ">
   <jsp:attribute name="lang">${content.lang}</jsp:attribute>
   <jsp:body>${content.body}</jsp:body>
</jsp:element>
```

Ще один простий стандартний тег `<jsp:text>` не містить атрибутів. Його тіло без усяких змін передається в вихідний потік. Наприклад:

```
<jsp:text>
   while(k < 10) {a[k]++; b[k++] = $1;}
</jsp:text>
```

Ви, напевно, помітили, що теги JSP не створюють ніякого коду ініціалізації серв- те, що зазвичай записується в метод `init()` сервлета. Такий код за потреби якось ініціалізувати отриманий після компіляції сервлет треба записати в метод `jspInit()` по наступною схемою:

```
<jsp:declaration>
   public void jspInit(){
      // Записуємо код ініціалізації
   }
</jsp:declaration>
```

Аналогічно, завершальні дії сервлет можна, можливо записати в метод `jspDestroy()` за такою схемою:

```
<jsp:declaration>
   public void jspDestroy(){
      // Записуємо код завершення
   }
</jsp:declaration>
```

Мова записи виразів EL

Хоча на сторінці JSP можна, можливо записати будь-яке вираз мови Java в тезі `<%=...%>` або в елементі XML `<jsp:expression>...</jsp:expression>` , JSP, починаючи з версії 2.0, вве- ден мова EL (Expression Language), призначений для записи виразів. Мова JSP EL з'явився в руслі тенденції до вигнання зі сторінок JSP чужорідного коду та за- міни його своїми "рідними" конструкціями.

У мові JSP EL вирази оточуються символами `${...}` , наприклад `${2 + 2}` . Це оточення дає сигнал до обчислення ув'язненого в них вирази. Вираз, запи-

санне без цих символів, не буде обчислюватися і сприйметься як простий набір символів.

У виразах можна використовувати дані типів `boolean` , `long` , `float` і рядкові кон- станти, укладені в лапки або апострофи. Значення `null` вважається окремим типом.

З даними цих типів можна виконувати арифметичні операції `+` , `-` , `*` , `/` , `%` , порівняно- ня `==` , `!=` , `<` , `>` , `<=` , `>=` , логічні операції `!` , `&&` , `||` та умовну операцію `?:` . Цікаво, що порівняння "рівно", "не рівно", "менше", "більше", "менше або одно", "більше" або одно" можна, можливо записати не тільки спеціальними символи, але і скороченнями `eq` , `ne` , `lt` , `gt` , `le` , `ge` слів "equal", "not equal", "less than", "greater", "less than or equal", "greater or equal". Це дозволяє залишити знаки "більше" і "менше" тільки для запису тегів XML. За аналогією операцію поділу можна записати словом `div` , операцію взяття залишку від поділу - словом `mod` _ а логічні операції - словами `not` , `and` і `or` .

У виразах можна звертатися до змінних, наприклад `${name}` , полів та методів об'єктів, наприклад `${pageContext.request.requestURI}` . У виразах мови JSP EL мож- але використовувати такі зумовлені об'єкти:

- `pageContext` - об'єкт типу `PageContext` ;
- `pageScope` - об'єкт типу `Map` , містить атрибути сторінки і їх значення;
- `requestScope` - об'єкт типу `Map` , містить атрибути запиту і їх значення;
- `sessionScope` - об'єкт типу `Map` , містить атрибути сеансу і їх значення;
- `applicationScope` - об'єкт типу `Map` , містить атрибути програми і їх зна-чення;
- `param` - об'єкт типу `Map` , містить параметри запиту, одержувані в сервлетахметодом `ServletRequest.getParameter(String name)` ;
- `paramValues` - об'єкт типу `Map` , містить параметри запиту, одержувані в серв-літах методом `ServletRequest.getParameterValues(String name)` ;
- `header` - об'єкт типу `Map` , містить заголовки запиту, одержувані в сервлетахметодом `ServletRequest.getHeader(String name)` ;
- `headerValues` - об'єкт типу `Map` , містить заголовки запиту, одержувані в серв-літах методом `ServletRequest.getHeaders(String name)` ;
- `initParam` - об'єкт типу `Map` , містить параметри ініціалізації контексту, по- випромінювані в сервлетах методом `ServletContext.getInitParameter(String name)` ;
- `cookie` - об'єкт типу `Map` , містить імена і об'єкти типу `Cookie` .

Зрештою, в виразах мови JSP EL можна, можливо записувати виклики функцій.

Вбудовані об'єкти JSP

Кожна сторінка JSP може містити у виразах та скриптлетах дев'ять готових вбудованих об'єктів, створюваних контейнером JSP при виконанні сервлета, напів- ченого після компіляції сторінки JSP. Ми вже використовували об'єкти `request` і

exception . У цих об'єктів задані певні імена і типи. У більшості випадківзадані не точні типи об'єктів, а їхні суперкласи і інтерфейси:

- request - об'єкт типу ServletRequest , частіше всього це об'єкт типу HttpServletRequest ;
- response - об'єкт типу ServletResponse , зазвичай це об'єкт типу HttpServletResponse ;
- config - об'єкт типу ServletConfig ;
- application - об'єкт типу ServletContext ;
- session - об'єкт типу HttpSession ;
- pageContext - об'єкт типу PageContext ;
- out - вихідний потік типу JspWriter ;
- page - довільний об'єкт класу Object ;
- exception - виняток класу Throwable .

У лістингу 27.3 наведено сторінка JSP, використовуюча скриптлети і вбудовані об'-екти для організації запиту до базі СДО, описаною у попередньому розділі.

Листинг 27.3. Страница JSP, использующая скриптлеты

```
<%@ page import="java.sql.*"
    contentType="text/html;charset=windows-1251" %>

<html><head><title>Запит до базі СДН</title></head>
<body>
<h2> Вітаю
   <%= (request.getRemoteUser() != null ? ", " +
        request.getRemoteUser() : "") %>!
</h2><hr><p>
<%
try{
   Connection conn =
     DriverManager.getConnection(
       (String)session.getValue("connStr"), "sdoadmin", "sdoadmin");

   Statement st = conn.createStatement ();

   ResultSet rs = st.executeQuery ("SELECT name, mark " +
      "FROM students ORDER BY name");

   if (rs.next()){
%>

<table border=1>
<tr>
  <th width = 200> <i>Учень</i> </th>
  <th width=100> <i>Оцінка</i> </th>
</tr>
<tr>
  <td> <%= rs.getString(1) %> </td>
```

```
  <td> <%= rs.getInt(2) %> </td>
</tr>

<%
  while (rs.next()){
%>

<tr>
  <td> <%= rs.getString(1) %> </td>
  <td> <%= rs.getInt(2) %> </td>
</tr>

<%
  }
%>

</table>

<%
  }else{
%>

<p> Вибачте, але відомостей ні! </p>

<%
   }
   rs.close();
   st.close();
} catch (SQLException e) {
   out.println("<p>Помилка під час виконання
   запиту:");out.println ("<pre>" + e + "</pre>");
}
%>
</body></html>
```

Звернення до компоненту JavaBean

З сторінки JSP можна, можливо звертатися до об'єктам Java, оформленим як компонентиJavaBeans. Це виконується тегом

```
<jsp:useBean id=" ім'я екземпляра компонента "
   [ scope=" page або request або session або application " ]
    Клас компонента
/>
```

"Ім'я екземпляра компонента" `id` визначає ім'я JavaBean, унікальне в заданій ат- рибу `scope` області. За замовчуванням приймається область `page` – поточна сторінка JSP та увімкнені в її сторінки.

Компонент зберігається як атрибут контексту вказаної атрибутом `scope` області та ви- ється методом `getAttribute()` відповідного контексту.

- Якщо атрибут `scope` дорівнює `"page"` , то компонент зберігається як один з атрибутів об'єму. екта класу `PageContext` .
- Якщо атрибут `scope` дорівнює `"request"` , то компонент зберігається як атрибут об'єкта типу
  `ServletRequest` .
- Якщо атрибут `scope` дорівнює `"session"` , то компонент буде атрибутом об'єкта типу
  `HttpSession` .
- Нарешті, якщо атрибут `scope` дорівнює `"application"` , то компонент стане атрибутом ти- па `ServletContext` .

Визначене в атрибуті `id` ім'я використовується при зверненні до властивостей та методів компонент JavaBean. Обов'язковий атрибут "клас компонента" описується однимз трьох способів:

- `class="` *повне ім'я класу* `" [ type="` *повне ім'я суперкласу* `" ]`
- `beanName="` *повне ім'я класу або вираз JSP* `"`
  `type="` *повне ім'я суперкласу* `"`
- `type="` *повне ім'я суперкласу* `"`

При зверненні до компоненту JavaBean в тілі елемента можна, можливо ставити і інші еле-менти.

Властивість (Property) вже викликаного тегом `<jsp:useBean>` компонента JavaBean з ім'ям `id="myBean"` встановлюється тегом

```
<jsp:setProperty name="myBean"
   property=" ім'я " value=" рядок або вираз JSP " />
```

або тегом

```
<jsp:setProperty name="myBean"
   property=" ім'я " param=" ім'я параметра запиту " />
```

У другому випадку властивості компонента JavaBean дається значення, певне параметром запиту, ім'я якого вказано атрибутом `param` .

Третя форма цього тегу

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

застосовується в тих випадках, коли імена всіх властивостей компонента JavaBean збігаютьсяз іменами параметрів запиту аж до збігу регістрів літер.

Для отримання властивостей вже викликаного компонента JavaBean з ім'ям `"myBean"` сущ-існує тег

```
<jsp:getProperty name="myBean" property=" ім'я властивості " />
```

У його атрибуті `property` вже не можна записувати зірочку.

Виконання аплету в браузері клієнта

Якщо в браузері клієнта встановлений Java Plug-in, то в ньому можна, можливо організувати викон-няння аплету або компонента з допомогою елемента

```
<jsp:plugin type=" bean або applet "
   [code=" ім'я класу аплету "]
   [codebase=" каталог аплету "]
   Інші параметри заголовка тега
>
   Тут записуються необов'язкові параметри
</jsp:plugin>
```

Як видно з цього опису, елемент `<jsp:plugin>` дуже схожий на тег `<applet>` мови HTML. Схожі і його атрибути `code` і `codebase` , тільки в імені класу аплету або ком- понента треба обов'язково вказувати його розширення `.class` . Якщо атрибут `codebase` не вказано, то за умовчанням розуміється каталог, у якому лежить сторінка JSP. Інші атрибути заголовка теж подібні параметрам тега `<applet>` :

- `name=" ім'я екземпляра "` ;
- `archive=" список адрес URL архівів аплету "` ;
- `align=" bottom або top, або middle, або left, або right "` ;
- `height=" висота в пікселах або вираз JSP "` ;
- `width=" ширина" в пікселах або вираз JSP "` ;
- `hspace=" горизонтальні поля в пікселах "` ;
- `vspace=" вертикальні поля в пікселах "` ;
- `jreversion=" версія JRE, по замовчуванням 1.2 "` ;
- `nspluginurl=" повний адреса URL, з якого можна, можливо завантажити Java Plug-in для NetscapeCommunicator "` ;
- `iepluginurl=" повний адреса URL, з якого можна, можливо завантажити Java Plug-in для InternetExplorer "` .

У тілі елемента можна, можливо помістити будь-які параметри виду

```
<jsp:params>
   <jsp:param name=" ім'я " value=" значення або вираз JSP " />
</jsp:params>
```

які будуть передані аплету чи компоненту. Крім того, в тілі елемента допускається тим вказувати повідомлення, яке з'явиться у вікні браузера, якщо аплет або компо- нент не вдалося завантажити. Для цього використовується елемент

```
<jsp:fallback> Текст повідомлення </jsp:fallback>
```

Передача управління

Сторінка JSP має можливість передати керування іншим ресурсом: сторінці JSP, сервлету або сторінці HTML. Це виконується тегом

```
<jsp:forward page=" адреса URL щодо контексту " />
```

містить адресу об'єкта, якому передається управління. Адреса може бути напів- чений як результат обчислення виразу JSP. Управління не повертається, та рядки, наступні за тегом `<jsp:forward>` , не будуть виконуватись.

Ресурсу можна, можливо передати один або кілька параметрів в тілі елемента:

```
<jsp:forward page=" адреса URL щодо контексту " >
   <jsp:param name=" ім'я " value=" значення або вираз JSP " />
</jsp:forward>
```

Користувальницькі теги

Розробник сторінок JSP може розширити набір стандартних дій (тегів) JSP, створивши свої власні, як кажуть, *користувацькі теги* (custom tags). Користувача- тельські теги організуються у вигляді цілої бібліотеки, навіть якщо до неї входить тільки один тег. Опис кожної бібліотеки зберігається в окремому XML-файлі з розширенням. ням tld, званим *описником бібліотеки тегів* TLD (Tag Library Descriptor). Цей файл зберігається в каталозі WEB-INF даного Web-програми або в його підкаталозі. Якщо Web-додаток упаковано в JAR-архів, то TLD-файли, описують біб- ліотеки користувацьких тегів цієї програми знаходяться в каталозі META-INF архів.

У лістингу 27.4 наведено приклад простого TLD-файлу, описуючого бібліотеку користувальницьких тегів з одним тегом `head` , реалізованим класом `HeadTag` . Повне об'- пояснення використаних в ньому елементів XML ви отримаєте у наступному розділі.

Листинг 27.4. Описатель TLD библиотеки тегов

```
<taglib xmlns=" http://java.sun.com/xml/ns/j2ee "
  xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance
  "xsi:schemaLocation=
    http://java.sun.com/xml/ns/j2ee _ web-jsptaglibrary_2_0.xsd"
  version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name></short-name>
  <uri>/sdo</uri>
  <tag>
    <name>head</name>
    <tag-class>sdotags.HeadTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
      <name>size</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

на сторінці JSP перед застосуванням користувальницьких тегів слід послатися на біб- ліотеку тегом

```
<%@ taglib uri=" адреса URI бібліотеки " prefix="
           префікс тегів бібліотеки _ %>
```

Наприклад, якщо на сторінці JSP написаний тег

```
<%@ taglib uri="/WEB-INF/sdotaglib.tld" prefix="sdo" %>
```

то на ній можна, можливо використовувати теги виду `<sdo:head />` .

Цей тег не має прямого XML-еквівалента. Префікс тегів бібліотеки та її адреса определяются у вигляді XML як простір імен атрибутом `xmlns` в елементі `<jsp:root>` . Наприклад, бібліотека з префіксом тегів `sdo` та адресою `/WEB-INF/sdotaglib.tld` описується так:

```
<jsp:root
   xmlns:jsp=" http://java.sun.com/JSP/Page
   "xmlns:sdo="/WEB-INF/sdotaglib.tld"
   version="2.0"
>
   Сторінка JSP
</jsp:root>
```

Необов'язковий елемент `<jsp:root>` , якщо він присутній, повинен бути кореневим елементом документ XML.

Як бачите, до кожної сторінки JSP завжди підключається бібліотека з префіксом тегів `jsp` . Стандартні (core) дії JSP входять в цю бібліотеку. Префікси `jsp` , `jspx` , `java` , `javax` , `servlet` , `sun` , `sunw` зарезервовані корпорацією Sun Microsystems, їх не можна упот- діти в бібліотеках користувацьких тегів.

У конфігураційному файлі web.xml можна створити псевдоніми адреси URI бібліотеці. кі з допомогою елемента `<taglib>` , наприклад:

```
<taglib>
   <taglib-uri>/sdo</taglib-uri>
   <taglib-location>/WEB-INF/sdotaglib.tld</taglib-location>
</taglib>
```

Після цього на сторінці JSP можна, можливо написати посилання на бібліотеку так:

```
<%@ taglib uri="/sdo" prefix="sdo" %>
```

Псевдонім може бути вказаний і в TLD-файлі, в елементі `<uri>` , як показано в листі- ге 27.4. У цьому випадку контейнер, виявивши на сторінці JSP тег `<%@taglib%>` з псевдонимом, переглядає TLD-файли у пошуках цього псевдоніма у їх елементах `<uri>` . Знайшовши псевдонім, контейнер зв'язує його шляхом до TLD-файлу, що містить псевдонім. Пошук TLD-файлів відбувається тільки в підкаталогах каталогу WEB-INF та в всіх JAR-архівах, а саме в каталогах META-INF, що знаходяться в JAR-архіви.

Кожен тег створюваної бібліотеки реалізується класом Java, званим у доку- ментації *обробником тега* (tag handler). Обробник тега повинен реалізувати ін- терфейс `Tag` , а якщо тег має тіло, яке треба виконати кілька разів, то потрібно реалізувати його розширення - інтерфейс `IterationTag` . Якщо ж тіло користувальницького тега вимагає попередньою обробки, то слід використовувати розширення інтерфейсу `IterationTag` - інтерфейс `BodyTag` . Ці інтерфейси зібрані в пакет `javax.servlet.jsp.tagext` . У ньому є і готові реалізації вказаних інтерфейсів. клас `TagSupport` , що реалізує інтерфейс `IterationTag` , і його розширення - клас `BodyTagSupport` , реалізуючий інтерфейс `BodyTag` .

Отже, для створення тега користувача без тіла або з тілом, але не вимагають попередньої обробки, зручно розширити клас `TagSupport` , а для створення тега з тілом, яке треба попередньо перетворити, - клас `BodyTagSupport` .

Класи, що реалізують бібліотеку тегів, зберігаються у каталозі WEB-INF/classes, а якщо вони упаковані в JAR-архів - то в каталозі WEB-INF/lib свого Web-додатки. Якщо бібліотека розділяється кількома Web-додатками, вона зберігається в ка- комусь загальному каталозі, наприклад common/lib. Відповідність між іменами тегів і класами, реалізуючими їх, вказується в TLD-файлі, що описує бібліотеку, у XML-елементі `<tag>` . У цьому елементі, у вкладеному XML-елементі `<attribute>` , описуються атрибути тега, що відкриває. Наприклад:

```
<tag>
   <name>reg</name>
   <tag-class>sdotags.RegTag</tag-class>
   <body-content>EMPTY</body-content>
   <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>java.lang.String</type>
   </attribute>
</tag>
```

Якщо після цього на сторінці JSP написати користувальницький тег `<sdo:reg>` , наприклад:

```
<sdo:reg name="%=request.getParameter(\"name\") %" />
```

або з допомогою мови JSP EL:

```
<sdo:reg name="${param.name}" />
```

то для обробки даного тега буде створено об'єкт класу `RegTag` з пакету `sdotags` .

Клас-обробник користувальницького тега

Як уже було сказано раніше, клас-обробник елемента, що не має тіла, або тіло якого не вимагає перетворень, має реалізувати інтерфейс `Tag` або розширити клас `TagSupport` . Якщо тіло елемента перед посилкою клієнту треба перетворити, то класу-оброблювачу слід реалізувати інтерфейс `BodyTag` або розширити клас `BodyTagSupport` . У випадку простих перетворень можна, можливо реалізувати інтерфейс `SimpleTag` або розширити клас `SimpleTagSupport` .

Основний метод інтерфейсу `Tag`

```
public int doStartTag();
```

виконує дії, запропоновані відкриваючим тегом. До нього сервлет звертається автоматично, починаючи обробку елемента. Метод повинен повернути одну з двох констант інтерфейсу `Tag` , вказівників на подальші дії:

- `EVAL_BODY_INCLUDE` - обробляти тіло елемента;
- `SKIP_BODY` - не обробляти тіло елемент.

Після завершення цього методу сервлет звертається до методом

```
public int doEndTag();
```

Котрий виконує дії, завершальні обробку користувальницького тега. Методповертає одну з двох констант інтерфейсу `Tag` :

- `EVAL_PAGE` - продовжувати обробку сторінки JSP;
- `SKIP_PAGE` - завершити на цьому обробку сторінки JSP.

Інтерфейс `IterationTag` додає метод

```
public int doAfterTag();
```

що дозволяє обробити повторно тіло користувальницького тега. Він виконуватиметься перед методом `doEndTag()` . Якщо метод `doAfterTag()` повертає константу `EVAL_BODY_AGAIN` інтерфейсу `IterationTag` , то тіло елемента буде оброблено ще раз, якщо константу `SKIP_BODY` - обробка тіла не стане повторюватися.

Інтерфейс `BodyTag` дозволяє буферизувати виконання тіла елемента. Буферизація проводиться, якщо метод `doStartTag()` повертає константу `EVAL_BODY_BUFFERED` інтерфейсу `BodyTag` . У такому випадку перед обробкою тіла тега контейнер звертається до методу

```
public void doInitBody();
```

Котрий може виконати різні попередні дії.

У цих методів ні аргументів, вони отримують інформацію з об'єкта класу `PageContext` , який завжди створюється Web-контейнером для виконання будь-якої сторінки. ниці JSP. При реалізації інтерфейсу `Tag` або `BodyTag` цей об'єкт можна отримати методом `getPageContext()` класу `JspFactory` , попередньо отримавши об'єкт класу `JspFactory` є його власним статичним методом `getDefaultFactory()` . Складність таких маніпуляцій ще один аргумент для того, щоб не реалізовувати інтерфейси, а розширювати клас `TagSupport` або клас `BodyTagSupport` .

Отже, для створення користувальницького тега без тіла або з тілом, що не вимагає обра- ботки, зручніше всього розширити клас `TagSupport` , а для створення користувальницького тега з обробкою тіла - розширити клас `BodyTagSupport` . При цьому розробник отримує в своє розпорядження об'єкт класу `PageContext` просто в вигляді захищеного поля `pageContext` .

Описані раніше методи реалізовані в класі `TagSupport` дуже просто:

```
public int doStartTag() throws JspException {
   return SKIP_BODY;
}
public int doEndTag() throws JspException {
   return EVAL_PAGE;
}
public int doAfterBody() throws JspException
   {return SKIP_BODY;
}
```

У підкласі `BodyTagSupport` реалізація методу `doStartTag()` трохи змінено:

```
public int doStartTag() throws JspException {
  return EVAL_BODY_BUFFERED;
}
```

Метод `doInitBody()` залишений порожній.

Отже, у найпростішому випадку достатньо розширити клас `TagSupport`, перевизначивши метод `doStartTag()`. Нехай, наприклад, визначено користувальницький тег без аргументів і без тіла, всього лише відправляючий клієнту повідомлення:

```
<sdo:info />
```

Реалізуючий його клас може виглядати так як показано в лістингу 27.5.

Листинг 27.5. Класс простейшего пользовательского тега

```
package sdotags;
import  javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class InfoTag extends TagSupport{
   public int doStartTag() throws JspException
   {
      pageContext.getOut().print("Бібліотека тегів СДО.");
      return SKIP_BODY;
   }
}
```

Початковий текст лістингу 27.5 треба відкомпілювати звичайним чином і встановити у контейнер так само, як встановлюється сервлет. Простежте за правильним відповідством пакетів Java та каталогів файлової системи: у каталозі WEB-INF/classes долдружин бути підкаталог sdotags з файлом InfoTag.class.

Ще простіше ці дії виконуються за допомогою методу `doTag()` інтерфейсу `SimpleTag`, реалізованого у класі `SimpleTagSupport`. У даного методу немає аргументів і повернення щатого значення, він об'єднує дії, зазвичай виконувані методами `doStartTag()` і `doEndTag()`.

Користувальницький тег з атрибутами

Для кожного атрибуту тега, що відкриває, треба визначити властивість JavaBean, тобто по- ле з ім'ям, що збігається з ім'ям атрибута, та методи доступу `get Xxx ()` та `set Xxx ()`. Наприклад, трохи раніше *(див. розд. "Користувацькі теги" даного розділу)* ми визначили користувальницький тег

```
<sdo:reg name="ім'я" />
```

з одним атрибутом `name`. Клас `RegTag`, міститься в лістингу 27.6, реалізує цей тег.

Листинг 27.6. Пользовательский тег с атрибутом

```
package sdotags;

import javax.servlet.jsp.*;import javax.servlet.jsp.tagext.*;

public class RegTag extends TagSupport{

private String name;

   public String getName(){
      return name;
}

   public void setName(String name) {
      this.name = name;
}

   public int doStartTag() throws JspException {
      if (name == null)
        name = pageContext.getRequest().getParameter("name");
      registrate(name);
      return SKIP_BODY;
   }
}
```

Користувальницький тег з тілом

Якщо у користувача тега є тіло, то при описі тега в TLD-файлі в елементі `<body-content>` замість слова `EMPTY` слід написати слово `JSP` або взагалі не писати цей елемент, оскільки його значення `JSP` приймається за замовчуванням.

У тіла елемента `<body-content>` можуть бути ще два значення. Значення `tagdependent` застосовується, якщо вміст тіла тега написано не мовою JSP, а якоюсь іншою- гом мовою, наприклад це запит на мовою SQL. Значення `scriptless` показує, що у тезі немає скриптлетів.

Якщо вміст тіла тега не потрібно обробляти, а треба тільки Відправити клієнту, то при створенні його обробника достатньо реалізувати інтерфейс `Tag` або розширити клас `TagSupport` . Якщо метод `doStartTag()` обробника поверне значення `EVAL_BODY_INCLUDE` , то все тіло тега буде автоматично відправлено в вихідний потік.

Нехай, наприклад, в файлі sdotaglib.tld визначено користувальницький тег `head` :

```
<tag>
   <name>head</name>
   <tag-class>sdotags.HeadTag</tag-class>
   <body-content>JSP</body-content>
   <attribute>
      <name>size</name>
      <required>false</required>
```

```
      <rtexprvalue>true</rtexprvalue>
   </attribute>
</tag>
```

Цей тег реалізований класом `HeadTag` , описаним в лістингу 27.7.

Листинг 27.7. Пользовательский тег с простым телом

```
package sdotags;

import  javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HeadTag extends TagSupport{

private String size = "4";

   public String getSize(){
      return size;
}

   public void setSize(String size) {
      this.size = size;
}

   public int doStartTag(){
      try{
         JspWriter out = pageContext.getOut();
         out.print("<font size=" + size + "">");
      } catch (Exception e) {
         System.err.println(e);
      }
      return EVAL_BODY_INCLUDE;
}

   public int doEndTag(){
      try{
         JspWriter out = pageContext.getOut();
         out.print("</font>");
      } catch (Exception e) {
         System.err.println(e);
      }
      return EVAL_PAGE;
   }
}
```

Після цього на сторінці JSP можна, можливо писати користувальницький тег

```
<sdo:head size = "2" >
   Сьогодні <jsp:expression>new java.util.Date()</jsp:expression>
</sdo:head>
```

Текст, написаний в його тілі, буде виведений у клієнта шрифтом зазначеного розміру.

Обробка тіла користувальницького тега

Якщо тіло користувальницького тега вимагає обробки, його клас-обробник повинен реалізувати інтерфейс `BodyTag` або розширити клас `BodyTagSupport` . Метод `doStartTag()` має повернути значення `EVAL_BODY_BUFFERED` . Після завершення методу `doStartTag()` , якщо тіло тега не порожнє, контейнер викличе метод `doInitBody()` , який може виконати нитка попередні дії перед обробкою вмісту користувальницького тіла тега. Далі контейнер звернеться до методу `doAfterBody()` , у якому треба проробити обробку тіла тега, оскільки до цьому моменту тіло тега буде прочитано і занесено в об'єкт класу `BodyContent` .

Клас `BodyContent` розширює клас `JspWriter` , отже, формально є вихідним потоком. Однак його зручніше розглядати як сховище інформації, отриманої з тіла тега.

Об'єкт класу `BodyContent` створюється після кожної ітерації методу `doAfterBody()` і все ці об'єкти зберігаються в стеку.

Посилання на об'єкт класу `BodyContent` можна, можливо отримати двома способами: методом

```
public BodyContent getBodyContent();
```

класу `BodyTagSupport` або, використовуючи об'єкт `pageContext` , наступним чином:

```
BodyContent bc = pageContext.pushBody();
```

Вміст тіла тега можна прочитати з об'єкта класу `BodyContent` теж двома способами. собами: або отримати посилання на символьний вхідний потік методом

```
public Reader getReader();
```

або уявити вміст об'єкта в вигляді рядки методом

```
public String getString();
```

Після обробки прочитаного вмісту його треба Відправити в вихідний потік `out` методом

```
public void writeOut(Writer out);
```

Вихідний потік `out` , аргумент цього методу, виводить інформацію у стек об'єктів класу `BodyContent` . Тому його можна, можливо отримати двома способами: методом

```
public JspWriter getPreviousOut();
```

класу `BodyTagSupport` або методом

```
public JspWriter getEnclosingWriter();
```

класу `BodyContent` .

Наведемо приклад. Нехай тег `query` , описаний в TLD-файлі sdotaglib.tld наступним проразом:

```
<tag>
   <name>query</name>
   <tag-class>sdotags.QueryTag</tag-class>
   <body-content>tagdependent</body-content>
   <attribute>
      <name>size</name>
```

```
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>
</tag>
```

містить в своєму тілі SQL-запити, наприклад:

```
<sdo:query>
   SELECT * FROM students
</sdo:query>
```

У лістингу 27.8 наведено фрагмент оброблювача цього тега.

Листинг 27.8. Пользовательский тег обработки SQL-запросов

```
package sdotags;

import java.sql.*;
import  javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class QueryTag extends BodyTagSupport{

   private Connection conn;
private ResultSet rs;

   public int doStartTag(){
      // . . .
      return EVAL_BODY_BUFFERED;
}

   public void doInitBody(){
   conn = DriverManager.getConnection(. . .);

       // Перевірка з'єднання
}

   public  int  doAfterBody(){
      BodyContent bc = getBodyContent();
      if (bc == null) return SKIP_BODY;

      String query = bc.getString();

      try{
         Statement st = conn.createStatement();
         rs = st.executeQuery(query);

         // Обробка результату запиту JspWriter

         out = bc.getEnclosingWriter();

         out.print("Висновок результатів");
      } catch (Exception e) {
         System.err.println(e);
      }
```

```
      return SKIP_BODY;
}

   public int doEndTag(){
      conn =  null;
      return EVAL_PAGE;
   }
}
```

Обробка взаємодіючих тегів

Часто користувацькі теги, розташовані на одній сторінці JSP, повинні взаємо- діяти один з одним. Наприклад, тіло тега може містити інші, вкладені, теги JSP. Вони можуть бути самостійними по відношенню до зовнішнього тегу або зави- мережа від нього. Так, наприклад, в мовою HTML тег `<tr>` може з'явитися тільки всередині тега `<table>` . При цьому атрибути тега `<table>` можуть використовуватися в тезі `<tr>` , а також можуть бути перевизначені всередині нього. Таких прикладів багато в мовою XML.

У мові JSP також можуть з'явитися теги, що залежать один від одного. Наприклад, ми можемо визначити тег

```
<sdo:connection source=" джерело даних " >
```

що встановлює з'єднання з базою даних, вказаною в атрибуті `source` . Усередині цього тега допустиме звернення до бази даних, наприклад:

```
<sdo:connection source="SDO" >
   <sdo:query >
      SELECT * FROM students
   </sdo:query>
   <sdo:insert>
      INSERT INTO students (name) VALUES ('Іванів')
  </sdo:insert>
</sdo:connection>
```

Звичайно, вкладені теги можна реалізувати вкладеними класами-обробниками чи розширеннями зовнішніх класів. Але в класах-оброблювачах користувальницьких тегів є свої кошти. Зовнішній та вкладений теги реалізуються окремими класами, що розширюють класи `TagSupport` або `BodyTagSupport` . Опис тегів у TLD-файлі ж не вкладені, вони записуються незалежно один від друга, наприклад:

```
<tag>
   <name>connection</name>
   <tag-class>sdotags.ConnectionTag</tag-class>
</tag>
<tag>
   <name>query</name>
   <tag-class>sdotags.QueryTag</tag-class>
</tag>
<tag>
   <name>insert</name>
   <tag-class>sdotags.InsertTag</tag-class>
</tag>
```

Для зв'язку обробників тегів використовуються їх методи. Спочатку обробник вкладений- ного тега ставить собі зовнішній, "батьківський" тег `parent` методом

```
public void setParent(Tag parent);
```

Потім обробник вкладеного тега може звернутися до оброблювача зовнішнього тега методі

```
public Tag getParent();
```

Більше загальний метод

```
public static final Tag findAncestorWithClass(Tag від, Class class);
```

дозволяє звернутися до оброблювача тега, не обов'язково безпосередньо навколишнього- го даний тег. Перший аргумент цього методу найчастіше просто `це` , а другий аргу- мент повинен реалізувати інтерфейс `Tag` або його розширення.

Отже, всі вкладені теги можуть звернутися до полів і методів зовнішніх тегів та взаємодії. модіювати з їх допомогою.

Припустимо, клас `ConnectionTag` , реалізуючий користувальницький тег `connection` , втомившись-ливає з'єднання з джерелом даних, як показано в лістингу 27.9.

Листинг 27.9. Реализация тега соединения с базой данных

```
public class ConnectionTag extends TagSupport{

  private Connection conn;

      public Connection getConnection(){
          return conn;
  }

      public int doStartTag(){
    Connection conn = DriverManager.getConnection(url, user, password);

        if (conn == null) return SKIP_BODY;
        return EVAL_BODY_INCLUDE;
  }

      public int doEndTag(){
          conn = null;
          return EVAL_BODY;
      }
}
```

Клас `QueryTag` , реалізуючий тег `query` , може скористатися об'єктом `conn` , як Бувай-зано в лістингу 27.10.

Листинг 27.10. Реализация связанного тега

```
public class QueryTag extends BodyTagSupport{

  private ConnectionTag parent;
```

```
  public int doStartTag(){

          parent = (ConnectionTag)findAncestorWithClass(this,
        ConnectionTag.class);

          if (parent == null) return SKIP_BODY;
          return EVAL_BODY_INCLUDE;
  }

      public int doAfterBody(){
    Connection conn = parent.getConnection();

    // Інші дії

        return SKIP_BODY;
      }
}
```

Інший спосіб зробити об'єкт `obj` доступним для кількох тегів - вказати його в ві-де атрибута якогось контексту. Це виконується методом

```
public void setAttribute(String name, Object obj);
```

класу `PageContext` . Контекст цього атрибуту - сторінка, на якої він визначено. Якщо область дії атрибуту треба розширити, то використовується метод

```
public void setAttribute(String name, Object obj, int scope);
```

того ж класу. Третій аргумент даного методу ставить область дії атрибуту -це одна з констант: `PAGE_SCOPE` , `APPLICATION_SCOPE` , `REQUEST_SCOPE` , `SESSION_SCOPE` .

Значення атрибуту завжди можна, можливо отримати методами

```
public Object getAttribute(String name, int scope);
public Object getAttribute(String name);
```

класу `PageContext` .

Для зручності роботи з атрибутом обробник тега може створити змінну. ну в області дії атрибуту і доступну для інших тегів в цією області (Scripting variable). Вона міститиме посилання на створений атрибут. Змінну можна визначити не тільки за атрибутом контексту, а й за атрибутом відкриває тега. Для визначення змінною є два методу.

Перший спосіб - вказати в TLD-файлі елемент `<variable>` , описуючий змінну. У цей елемент вкладається хоча би один із двох елементів:

- `<name-given>` - постійне ім'я змінної;
- `<name-from-attribute>` - ім'я атрибуту, значення якого буде ім'ям пере-змінної.

Інші вкладені елементи необов'язкові:

- `<variable-class>` - клас змінної, по замовчуванням `String` ;
- `<declare>` - створювати чи новий об'єкт при зверненні до змінної, по замовчуванням `true` ;

- `<Scope>` - область дії змінної, одна з трьох констант:
  - `NESTED` - між відкриваючим і що закриває тегами, т. е. в методах `doStartTag()` , `doInitBody()` , `doEndBody()` ; приймається по замовчуванням;
  - `AT_BEGIN` - від відкриває тега до кінця сторінки;
  - `AT_END` - після закриваючого тега до кінця сторінки.

Наприклад:

```
<tag>
   <name>tagname</name>
   <tag-class>sdotags.SomeTag</tag-class>
   <variable>
      <name-given>varname</name-given>
      <variable-class>java.lang.String</variable-class>
      <declare>true</declare>
      <scope>AT_END</scope>
   </variable>
</tag>
```

Другий спосіб - визначити об'єкт, містить ту ж саму інформацію, що іелемент `<Variable>` . Даний об'єкт створюється як розширення класу `TagExtraInfo` .

Ось як виглядає це розширення для прикладу, наведеного раніше:

```
public class MyTei extends TagExtraInfo{
  public VariableInfo[] getVariableInfo(TagData data){
    return new VariableInfo[]{
      new VariableInfo(data.getAttributeString("id"),
                       "java.lang.String", true, VariableInfo.AT_END)
    };
  }
}
```

Клас `MyTei` описується в TLD-файлі в елементі `<tei-class>` наступним чином:

```
<tag>
   <name>tagname</name>
   <tag-class>sdotags.SomeTag</tag-class>
   <tei-class>sdotags.MyTei</tei-class>
   <bodycontent>JSP</bodycontent>
</tag>
```

Цей спосіб має більше можливостей, ніж елемент `<variable>` . Клас `TagExtraInfo` має у своєму розпорядженні екземпляр класу `TagData` , що містить відомості про тег, зібрані з TLD-файл. Для зручності використання цього екземпляра в класі `TagExtraInfo` є логічний метод

```
public boolean isValid(TagData info);
```

Його можна, можливо застосовувати для перевірки атрибутів тега на етапі компіляції сторінки JSP.

Обробка винятків в користувальницьких тегах

Раніше вже говорилося про те, що обробку виключення, що виник на сторінці JSP, можна, можливо перенести на іншу сторінку, зазначену атрибутом `errorPage` тега `<%@ page %>` . У тегах користувача можна полегшити обробку виключення, реалізувавши інтер- фейс `TryCatchFinally` .

Інтерфейс `TryCatchFinally` описує всього два методу:

```
public void doCatch(Throwable thr);
public void doFinally();
```

Метод `doCatch()` викликається автоматично контейнером при виникненні виключе- ня в одному з методів `doStartTag()` , `doInitBody()` , `doAfterTag()` , `doEndTag()` обробника, реалізує інтерфейс `TryCatchFinally` . Методу `doCatch()` передається створений об'єм - ект-виключення. Метод `doCatch()` сам може викинути виняток.

Метод `doFinally()` виконується у всіх випадках після методу `doEndTag()` . Він уже не може викидати винятки.

Наприклад, при реалізації тега `допускається` використання методів інтерфейсу `TryCatchFinally` для відкату транзакції наступним чином:

```
public class ConnectionTag extends TagSupport implements TryCatchFinally{

  private Connection conn;

  // Інші поля і методи класу

      public void doCatch(Throwable t) throws Throwable {
         conn.rollback();
         throw t;
  }

      public void doFinally(){
         conn.close();
      }
}
```

Обробка тегів засобами JSP

Вже згадувана тенденція до вигнання зі сторінок JSP будь-якого чужорідного коду, навіть коду Java, призвела до того, що для обробки тега користувача ви можете замість класу Java написати сторінку JSP, описавши на ній дії тега. Ім'я файлу з такою сторінкою-обробником JSP отримує розширення tag, а якщо вона оформлена як документ XML, то розширення tagx. Файл, що містить окремий фрагмент стра- ниці-обробника, отримує ім'я з розширенням tagf.

Наприклад, тег `<sdo:info />` , дія якого ми показали в лістингу 27.5, можна пропрацювати наступною, дуже простою, сторінкою JSP, записаною в tag-файл з ім'ям info.tag:

```
<jsp:root
     xmlns:jsp=" http://java.sun.com/JSP/Page "
```

```
      version="2.0" >
   Бібліотека тегів З
   ДО.
</jsp:root>
```

Оскільки елемент `<jsp:root>` необов'язковий, весь файл info.tag може перебувати тількиз одного рядка:

```
Бібліотека тегів З ДО.
```

На сторінці-обробнику тегів користувача можна записувати будь-які теги JSP, крім директиви `page` . Замість її вказується спеціально введена для цього цілі у мову JSP директива `tag` . Наприклад:

```
<%@ tag body-content="scriptless">
```

Друга відмінність полягає в тому, що в директиві `<%@taglib%>` слід записувати не атрибут `uri` , а атрибут `tagdir` , в якому вказується каталог з tag-файлами, наприклад:

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="sdo" %>
```

Атрибути оброблюваного користувальницького тега описуються ще однією спеціальне введеною директивою `attribute` , аналогічною елементу `<attribute>` файлу TLD. Нехай, наприклад, на сторінці JSP застосовується тег користувача з двома атрибу- тамі:

```
<sdo:reg name="Іванів" age="25"/>
```

У файлі reg.tag, оброблювальному цей тег, можна, можливо написати

```
<%@ attribute name="name"
   required="true"
   fragment="false"
   rtexprvalue="false"
%>
<%@ attribute name="age"
   type="java.lang.Integer"
%>
```

Якщо тип `type` атрибуту не вказано, то по замовчуванням він вважається рядком класу
`java.lang.String` .

Типом атрибуту може бути фрагмент JSP, т. е. об'єкт класу `javax.servlet.jsp.tagext.JspFragment` . У такому разі атрибути `type` і `rtexпервину` не вказуються, а атрибут `fragment="true"` .

Після цього описи значення атрибутів, введені в тег, що обробляється, в прикладі вони рівні `"Іванів"` і `"25"` , можна, можливо використовувати в файлі reg.tag як `${name}` та `${age}` .

Третя, спеціально введена директива `variable` _ аналогічна елементу `<variable>` TLD-файлу, дозволяє описати змінну на сторінці-обробнику. Наприклад:

```
<%@ variable name-given="x"
   variable-class="java.lang.Integer"
   scope="NESTED"
   declare="true"
%>
```

Область дії змінної визначається атрибутом `scope` , таким самим, як описаний у *розд. "Обробка взаємодіючих тегів" цього розділу* однойменний елемент TLD-файл.

Значення атрибутів, що є фрагментами сторінок JSP, і користувальницьке тіло тега не обчислюються Web-контейнером, а передаються tag-файлу.

Для того, щоб змусити Web-контейнер виконати значення атрибута, що є фрагментом сторінки JSP, в tag-файлі потрібно використовувати стандартний тег `<jsp:invoke>` . Нехай, наприклад, атрибут `price` описаний наступним чином:

```
<%@ attribute name="price"
              fragment="true"
%>
```

Для обчислення його значення Web-контейнером у tag-файлі слід написати:

```
<jsp:invoke fragment="price"/>
```

Якщо ж треба змусити Web-контейнер виконати не фрагмент, а все тіло користувательського тега, то в tag-файлі записується стандартний тег `<jsp:doBody/>` .

Tag-файли не вимагають ніякого TLD-описи, якщо файл, містить сторінку- обробник тега користувача, зберігається в каталозі WEB-INF/tags/ або в його під- каталогах. Інше місце зберігання tag-файлів має бути описане в TLD-файлі еле- ментом `<tag-file>` . Наприклад:

```
<taglib>
   <tag-file>
      <name>info</name>
      <path>/WEB-INF/sdo/tags/info.tag</path>
   </tag-file>
</taglib>
```

Якщо Web-додаток упаковано в JAR-архів, то tag-файли повинні зберігатися в каталозі META_INF/tags/ архів або в його підкаталогах.

Стандартні бібліотеки тегів JSTL

Незважаючи на недовгу історію мови JSP, її можливість створення користувача тегів була використана багатьма фірмами та окремими розробниками. Вже створено але безліч бібліотек тегів користувача. Їх огляд можна подивитися, наприклад заходів, на сайті **http://www.servletsuite.com/jsp.htm** . У рамках уже не раз згадуючого ся проекту Jakarta створена потужна і широко застосовувана бібліотека ських тегів Struts, яку можна скопіювати з сайту **http://struts.apache.org/** . До жаль, рамки нашої книги не дозволяють дати її опис.

Корпорація Sun Microsystems створила стандартні бібліотеки користувацьких те- гів, які мають загальну назву JSTL (JSP Standard Tag Library). Їх реалізації складають- ють пакет `javax.servlet.jsp.jstl` та його підпакети. Довідку про те, де знайти найпо- слідку реалізацію можна отримати за адресою **http://www.oracle.com/technetwork/ java/index-jsp-135995.html** . Втім, бібліотеки JSTL входять в стандартну постачання

Java EE SDK. Всю бібліотеку JSTL складають два JAR-архіву: `jstl-1.2.jar` і `standard.jar` . Номер версії JSTL, звичайно, може бути іншим.

У пакет JSTL входять п'ять бібліотек користувальницьких тегів: core, xml, fmt, sql і fn.

Бібліотека core

Бібліотека core описується на сторінці JSP тегом

```
<%@ taglib uri=" http://java.sun.com/jsp/jstl/core " prefix="c" %>
```

До неї входять теги, що створюють подібність мови програмування. Вони покликані замі- нитка оператори Java, тим самим усуваючи скриптлети зі сторінок JSP.

Тег `<c:out>` виводить значення свого обов'язкового атрибуту `value` у вихідний потік `out` . Якщо воно порожнє, то буде виведено значення атрибута `default` , якщо, звичайно, цей атрибут написаний.

Наприклад:

```
Здрастуйте, шановний <c:out value="${name}"
   default="пан" /> !
```

Тег `<c:set>` встановлює значення змінної, ім'я якої задається атрибутом `var` , а область дії - атрибутом `scope` . Наприклад:

```
<c:set var="name" scope="session" value="${param.name}"/>
```

або, щось ж саме:

```
<c:set var="name" scope="session">
   ${param.name}
</c:set>
```

Тег `<c:remove>` видаляє раніше певні змінні, наприклад:

```
<c:remove var="name" scope="session"/>
```

Тег `<c:url>` також встановлює змінну, але її значення має бути адресою, записаною в формі URL. Наприклад:

```
<c:url var="bhv" value=" http://www.bhv.ru "/>
```

або, щось ж саме:

```
<c:url var="name">
   http://www.bhv.ru
</c:url>
```

Після цього змінну `bhv` можна використовувати для створення сеансу зв'язку з пользова- телем, як це робилось у попередньому розділі, або записувати у гіперпосиланнях HTML:

```
<a href="${bhv}">Видавництво</a>
```

До створюваної URL-адреси можна додати параметри вкладеним тегом `<c:param>` , наприклад:

```
 <c:url var="bhv" value=" http://www.bhv.ru
                                        ">
   <c:param name="id" value="${book.id}"/>
```

```
   <c:param name="author" value="${book.author}"/>
</c:url>
```

Тег `<c:if>` перевіряє умова, записане в його атрибуті `test` логічним виразом. Якщо умова істинно, то виконується тіло тег. Наприклад:

```
<c:if test="${age <= 0}">
   Шановний ${name}! Вкажіть, будь ласка, свій вік.
</c:if>
```

Як бачите, цей тег не реалізує повністю умова "if-then-else". Таке розгалуженняможна, можливо організувати тегом `<c:choose>` .

Тег `<c:choose>` з вкладеними в нього елементами `<c:when>` і `<c:otherwise>` реалізує ви-бір одного з варіантів.

Наприклад:

```
<c:choose>
   <c:when test="${balance < 0}" >
        на вашому рахунку негативний залишок.
   </c:when>
   <c:when test="${balance == 0}" >
        на вашому рахунку нульовий залишок.
   </c:when>
   <c:when test="${balance > 0}" >
        на вашому рахунку позитивний залишок.
   </c:when>
   <c:otherwise>
        Ні відомостей про вашому залишку.
   </c:otherwise>
</c:choose>
```

У прикладі лістингу 27.11 показано, як можна організувати розгалуження за допомогою тега `<c:choose>` . У ньому перевіряється, чи відомо відомості про клієнта. Якщо вони з-відомі, клієнту посилається привітання, якщо ні - форма для введення імені та пароля.

Листинг 27.11. Ответ, сформированный с помощью JSTL

```
<%@ taglib uri=" http://java.sun.com/jsp/jstl/core " prefix="c" %>
<html>
  <head><title>Перевірка введення імені</title></head>
  <body>
    <c:choose>
      <c:when test="${user != null}">
        Вітаю, ${user.name}!
      </c:when>
      <c:otherwise>
        <form method="POST" action="/sdo/jsp/user.jsp">
          Ім'я:  <input type="text" name="name">
          Пароль: <input type="password" name="pswd">
          <input type="submit" name="ok" value="Надіслати">
        </form>
```

```
      </c:otherwise>
    </c:choose>
  </body>
</html>
```

Тег `<c:forEach>` має дві форми. Перша створює цикл, що пробігає колекцію типу `Collection` або `Map` , вказану атрибутом `items` . Посилання на поточний елемент колекції заноситься до змінної, визначеної атрибутом `var` . Її можна використовувати у тілі тега.

Наприклад:

```
<c:forEach var="item" items="${sessionScope.cart.items}">
   <td>
      ${item.quantity}
   </td>
</c:forEach>
```

Друга форма тега `<c:forEach>` створює цикл з перерахуванням, в якому змінна циклу, що визначається атрибутом `var` , пробігає від початкового значення, що задається значенням атрибуту `begin` , до кінцевого значення, задається значенням атрибуту `end` ,з кроком - значенням атрибуту `step` . Наприклад:

```
<c:forEach var="k" begin="0" step="1" end="${n}" >
   <td>
       ${k}-й стовпець
   </td>
</c:forEach>
```

Тег `<c:forTokens>` розбиває рядок символів, заданий атрибутом `items` , на слова по-подібно до класу `StringTokenizer` , розглянутому в *розділі 5* . Розділювачі слів перераховуючи- ються в атрибуті `delims` . Поточне слово заноситься в змінну, певну атрибу-том `var` . Наприклад:

```
<c:forTokens var="word" items="${text}" delims=" \n\r\t:;,.?!">
   <c:out value="${word}"/>
</c:forTokens>
```

Тег `<c:import>` включає на сторінку JSP ресурси по їх адресою URL. Наприклад:

```
<c:import url="/html/intro.html"
          var="intro"
          scope="session"
          charEncoding="windows-1251"
/>
```

Змінну, визначену атрибутом `var` , можна використовувати у своїй області дей- ності, визначеної атрибутом `scope` (за замовчуванням, `page`). Атрибут `charEncoding` пока-кликає кодування символів включається ресурсу. за замовчуванням це кодування ISO 8859-1, яка погано підходить для кирилиці.

Тег `<c:redirect>` припиняє обробку сторінки і посилає HTTP-відповідь `redirect` клі- енту з вказівкою адреси, записаного в атрибуті `url` . Браузер клієнта зробить новий

запит на цю адресу. У відповідному сервлет метод `doEndTag()` повертає кон- станту `SKIP_PAGE` . Наприклад:

```
<c:redirect url="/books/list.html" context="/lib" />
```

Необов'язковий атрибут `context` встановлює контекст для нового запиту.

У тег `<c:redirect>` , як і тег `<c:url>` , можна вкласти теги `<c:param>` , що задають пара-метри нового запиту.

Бібліотека xml

Бібліотека xml, описувана тегом

```
<%@ taglib uri=" http://java.sun.com/jsp/jstl/xml " prefix="x" %>
```

містить теги `<x:out>` , `<x:set>` , `<x:forEach>` , `<x:if>` , `<x:choose>` , `<x:when>` , `<x:otherwise>` , ана- логічні відповідні теги бібліотеки core, але вибирають потрібний елемент XML з інтерпретованого документа атрибутом `select` , а також теги `<x:parse>` і `<x:transform>` , інтерпретуючі і перетворюючі документ XML.

Робота з бібліотекою xml заснована на адресації елементів документа XML засоби- ми мови XPath, що виходить за рамки нашої книги.

Бібліотека fmt

Бібліотека fmt містить теги, які допомагають інтернаціоналізації сторінок JSP. Вона описується так:

```
<%@ taglib uri=" http://java.sun.com/jsp/jstl/fmt " prefix="fmt" %>
```

У її входять теги `<fmt:setLocale>` , `<fmt:timeZone>` , `<fmt:formatDate>` , `<fmt:parseDate>` , `<fmt:formatNumber>` , `<fmt:parseNumber>` та інші теги, які роблять локальні установки.

Наприклад, тег

```
<fmt:formatNumber var="formattedAmount" pattern="0.00" value="${amount}" />
```

запише у змінну `Форматованому` типу типу `String` кількість `об'єму` з двома цифрами в дробовий частини, а тег

```
<fmt:formatDate var="ruDate" pattern="dd.MM.yyyy" value="${today}" />
```

запише в змінну `ruDate` типу `String` дату `today` в вигляді 08.12.2007.Теги

```
<fmt:parseNumber var="n" pattern="0.00" value="${amount}" />
<fmt:formatDate var="d" pattern="dd.MM.yyyy" value="${today}" />
```

виконують зворотне перетворення рядки символів, заданою атрибутом `value` , в об'єкти типу `Number` і `Date` відповідно, записані в змінні `n` і `d` по шабло- ну `pattern` .

Ці теги реалізовані класами `DecimalFormat` та `SimpleDateFormat` з пакету `java.text` та перетворюють дані за правилами цих класів, які можна подивитися в документ- тації Java SE.

Бібліотека sql

Четверта бібліотека, sql, описувана тегом

```
<%@ taglib uri=" http://java.sun.com/jsp/jstl/sql " prefix="sql" %>
```

містить теги зв'язку і роботи з базами даних: `<sql:setDataSource>` , `<sql:query>` , `<sql:update>` та `<sql:transaction>` . Їхня дія побудована на JDBC і робота з ними дуже схожа на роботу з базами даних через JDBC, що можна зрозуміти з наступного приміра:

```
<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
   url="jdbc:oracle:thin:@localhost:1521:ORCL"
   user="scott"
   password="tiger"
/>
<sql:query var="depts" >
   select * from DEPT
</sql:query>
<sql:transaction>
  <sql:update>
      insert into DEPT values(50, 'XXX', 'YYY')
  </sql:update>
</sql:transaction>
```

Бібліотека fn

П'ята бібліотека, fn, містить функції для обробки рядків. Вона описується тегом

```
<%@ taglib uri=" http://java.sun.com/jsp/jstl/functions " prefix="fn" %>
```

і містить теги `<fn:substring>` , `<fn:substringBefore>` , `<fn:substringAfter>` , `<fn:indexOf>` , `<fn:toUpperCase>` `<fn:toLowerCase>` , `<fn:startsWith>` , `<fn:endsWith>` , `<fn:contains>` , `<fn:containsIgnoreCase>` , `<fn:trim>` , `<fn:replace>` , `<fn:split>` , `<fn:join>` , `<fn:escapeXml>` . Ви, безсумнівно, дізнаєтеся в назвах цих тегів звичні методи обробки рядків, кото- рі ми вивчили в *главі 5* .

Крім тегів обробки рядків, бібліотека fn містить функцію `<fn:length>` , обчислюю- щую як довжину рядки, так і довжину колекції, переданою їй в якості аргументу. Наприклад:

```
<c:if test="${fn:length(param.username) > 0}" >
   <%@include file="response.jsp" %>
</c:if>
```

Frameworks

Приступаючи до розробки Web-програми, кожна команда вирішує питання про його архі- тектурі. Частіше всього відповідь знаходиться в схемою MVC (Model-View-Controller) *(Див. розділ 3)* . Один або кілька сервлетів, що приймають та обробляють запро- си клієнта, складають *Контролер* . Підготовка відповіді, зв'язок з базою даних, відбір

інформації, все те, що називається бізнес-логікою або бізнес-процесами, виконує ється класами Java, що утворюють *Модель* . Сторінки HTML і JSP, заповнені ін- формацією, отриманої від *Моделі* , складають *Вид* .

У ході обговорення та реалізації проекту кожна з трьох частин схеми MVC конкретизується, описується інтерфейсами та абстрактними класами. Іноді на цьому етапі виходить наполовину реалізована конструктивна схема, придатна для виконання ня цілого класу типових проектів. Для завершення проекту Web-додатки ос- тається реалізувати кілька інтерфейсів, дописати при необхідності свої власні ні класи та визначити під свій проект параметри додатка, записавши їх у конфігураційні файли.

Найбільш вдалі з таких шаблонів готової програми стають надбанням всієї спільноти програмістів, активно обговорюються, покращуються, модернізують- ся і набувають широкого поширення. Такі загальновизнані шаблони, або, по- іншому, каркаси програмного продукту отримали назва *Frameworks* .

У технології Java вже є десятки таких каркасів, в їх числі JSF, Seam, Struts, Facelets, Tales, Shale, Spring, Velocity, Tapestry, Jena, Stripes, Trails, RIFE, WebWork та безліч інших. До складу Java EE SDK входить каркас JSF. Познайомимося з ним під- дрібніше.

JavaServer Faces

Framework під назвою JSF (JavaServer Faces) виріс із простої бібліотеки тегів, розширює можливості тегів HTML. Він входить у стандартне постачання Java EE, хоча всіма можливостями JSF можна скористатися і не маючи на комп'ютері Java EE. Є багато інших реалізацій інтерфейсів JSF.

Еталонну реалізацію, яку надає проект GlassFish, можна, можливо завантажити з сайту **http://javaserverfaces.dev.java.net/** . З інших реалізацій слід виділити дуже популярний продукт Apache MyFaces, **http://myfaces.apache.org/** .

Все необхідне для роботи з JSF зібрано в одному архівному файлі. Достатньо розпакувати його в якийсь каталог, і JSF готовий до роботи. У цьому каталозі ви знайдете підкаталоги з документацією та підкаталог з докладними прикладами. Основу реалізації- цієї JSF складають два JAR-файли `jsf-api.jar` і `jsf-impl.jar` з підкаталогу lib, кото- рі треба скопіювати у вашу Web-додаток або у ваш сервер додатків або запи- сати шляхи до ним в змінну `CLASSPATH` . Для роботи JSF знадобиться бібліотека JSTL, простежте за тим, щоб у вашої Web-додатки був доступ до її JAR-архіву, на- приклад файлу `jstl-1.2.jar` .

Каркас JSF побудований по схемою MVC (Model-View-Controller), яку ми обговорювалив *Розділ 3* .

Роль Контролера в JSF грає сервлет під назвою `FacesServlet` . Як і Усе сервлети, він має бути описаний у конфігураційному файлі web.xml. Цей опис виглядає так:

```
<web-app>
 . . . . .
   <servlet>
```

```
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    </servlet>
     . . . . .
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
 . . . . .
</web-app>
```

Після цього опису ми можемо звертатися до JSF із запитами виду

```
http://localhost:8080/MyAppl/index.faces ,
http://localhost:8080/MyAppl/login.faces
```

Часто елемент `<servlet-mapping>` записують по іншому:

```
<servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
 </servlet-mapping>
```

Після такого опису запити до JSF повинні виглядати так:

```
http://localhost:8080/MyAppl/faces/index.jsp ,
http://localhost:8080/MyAppl/faces/login.jsp
```

І в першому, і в другому випадку запитуються файли index.jsp та login.jsp. Розширення імені файлу faces в запитах першого виду зроблено тільки для JSF, на сервері файлів з такими іменами ні. Побачивши в запиті ім'я index.faces, JSF буде Шукати файл index.jsp.

Для організації Вида у JSF розробляються бібліотеки тегів. До складу JSF зараз входять дві бібліотеки: core та html. Вони зазвичай описуються на сторінці JSP дирек- тивами

```
<%@ taglib uri=" http://java.sun.com/jsf/html " prefix="h" %>
<%@ taglib uri=" http://java.sun.com/jsf/core " prefix="f" %>
```

Усі теги JSF вкладаються в елемент `<f:view>` . Компілятор JSF переглядає і ком- пилює лише теги, вкладені в `<f:view>` . Теги файлу повинні бути вкладені в елемент `<f:subview>` . Це можна, можливо зробити так, як показано в лістингу 27.12.

Найчастіше на сторінці JSP формується інтерфейс користувача, який буде відображено браузером клієнта. У ньому розміщуються форми з полями введення, списками вибору, кнопками та іншими компонентами, текст з гіперпосиланнями, панелі, вкладки, таблиці.

Створення класичної сторінки HTML не задовольняє ні розробників, ні клієнтів. Набір тегів HTML невеликий і фіксований, їх можливості дуже обмежені. ченни. Вже давно вигадуються різні способи пожвавлення сторінок HTML: ци стилів CSS, динамічний HTML, аплети. Бібліотека тегів html в першу оче- редь покликана посилити можливості тегів HTML.

Для кожного тега HTML в JSF є відповідний тег, що має додатковий ними можливостями. У лістингу 27.12 показано форму з полями введення імені та пароля і кнопкою типу Submit.

Листинг 27.12. Страница JSP с тегами библиотеки JSF

```
<html><head>
<%@ taglib uri=" http://java.sun.com/jsf/html " prefix="h" %>
<%@ taglib uri=" http://java.sun.com/jsf/core " prefix="f" %>
</head><body>
<f:view>

   <f:subview id="myNestedPage">
       <jsp:include page="theNestedPage.jsp" />
   </f:subview>

      <h:form>
      <h:inputText id="name" size="50"
                  value="#{cashier.name}"
                  required="true">
          <f:valueChangeListener type="listeners.NameChanged" />
      </h:inputText>
      <h:inputSecret id="pswd" size="50"
                  value="#{cashier.pswd}"
                  required="true">
          <f:valueChangeListener type="listeners.NameChanged" />
      </h:inputSecret>

      <h:commandButton type="submit" value="Надіслати"
                      action="#{cashier.submit}"/>

   </h:form>

</f:view>
</body></html>
```

У бібліотеці html є більше двох десятків тегів, що відповідають графічним компонентам інтерфейсу користувача. Крім того, розробник може легко створити свої, *користувальницькі* компоненти (custom components), розширивши класи JSF. Вже створено багато бібліотек тегів для JSF, що вільно розповсюджуються або комерційноських.

Головна особливість бібліотеки html полягає в тому, що її складають не просто теги, а цілі компоненти. Компоненти JSF реагують на події миші та клавіатури, які можуть бути оброблені звичайними засобами JavaScript або спеціальними засобами JSF. Для цього введені теги-обробники подій. Один такий тег,
`<v:valueChangeListener>` , показаний у лістингу 27.12. Треба сказати, що розробники тегів бібліотеки html свідомо орієнтувалися на компоненти Swing, намагаючись принаймні можливості, наділити теги такими самими властивостями. Зокрема, компоненти JSF мож- але розміщувати на формі, користуючись підходящим IDE. Це "вміє" робити, наприклад,Java Studio Creator.

Форма, записана в лістингу 27.12, посилає на сервер ім'я та пароль, пов'язані з по- лями `cashier.name` та `cashier.pswd` атрибутом `value` . Що це за поля та якому об'єкту `cashier` вони належать?

Дані, отримані від HTML-форми, зберігатимуться в об'єкті, клас якого дол- дружин написати розробник Web-додатки. Цей клас оформляється як JavaBean, щоб JSF міг заповнювати та читати його поля методами доступу. Він буде частиною *моделі* в схемою MVC. Клас для зберігання імені і пароля, отриманих від форми листин- га 27.12, показаний у лістингу 27.13.

Листинг 27.13. Класс с данными HTML-формы

```
package myjsf;

import javax.faces.bean.*;

@ManagedBean(name="cashier")
@RequestScoped
public class Cashier{
    private String name;
    private String pswd;
    public String getName(){ return name; }
    public void setName(String name) { this.name = name;
    }public String getPswd(){ return pswd; }
    public void setPswd(String pswd) { this.pswd = pswd;
    }public String submit(){
       if ("Cashier".equalsIgnoreCase(name) && "rT34?x_D".equals(pswd))
          return "success";
       else
          return "failure";
    }
}
```

Як бачите, в цьому ж класі записаний метод обробки клацання за кнопкою **Відправити** .

Тепер треба якимось чином вказати JSF цей клас. Відомості про нього записуються в анотаціях або в конфігураційному файлі XML faces-config.xml, який повинен зберігатися у каталозі WEB-INF разом із файлом web.xml. Цей файл для нашого приміра записаний в лістингу 27.14.

Листинг 27.14. Конфигурационный файл JSF

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config xmlns=" http://java.sun.com/xml/ns/javaee " xmlns:xsi="
              http://www.w3.org/2001/XMLSchema-instance "
              xsi:schemaLocation=" http://java.sun.com/xml/ns/javaee
              http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd
              "version="1.2">
    <managed-bean>
        <description>
```

```
            Клас-обробник реєстрації касира
        </description>
        <managed-bean-name>cashier</managed-bean-name>
        <managed-bean-class>myjsf.Cashier</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
    <navigation-rule>
        <from-view-id>/index.jsp</from-view-id>
        <navigation-case>
            <description>
                Після вдалою перевірки перехід на сторінку welcome.jsp
            </description>
            <from-outcome>success</from-outcome>
            <to-view-id>/welcome.jsp</to-view-id>
     </navigation-case>
     <navigation-case>
         <description>
                Після невдалою перевірки повернення на сторінку index.jsp
            </description>
            <from-outcome>failure</from-outcome>
            <to-view-id>/index.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Елемент `<managed-bean>` пов'язує ім'я класу `cashier` , використовуване в лістингу 27.12, з повним ім'ям класу `myjsf.Cashier` . Елемент `<navigation-rule>` показує, звідки (`<from-view-id>`) і куди (`<to-view-id>`) слід перейти після обробки даних методом `submit()` об'єкта `cashier` , а також за якої події (`<from-outcome>`) зробити той чи інакший перехід.

Для повноти лишилося написати файл welcome.jsp. Він може виглядати так, як показале в лістингу 27.15.

Листинг 27.15. Страница приветствия с тегами библиотеки JSF

```
<html><head>
<%@ taglib uri=" http://java.sun.com/jsf/html " prefix="h" %>
<%@ taglib uri=" http://java.sun.com/jsf/core " prefix="f" %>
</head><body>
<f:view>

     Вітаю, ${cashier.name}!

</f:view>
</body></html>
```

Огляд усіх можливостей JSF виходить за межі нашої книги. Ви можете ознайомитись з ними на сайті розробників JSF **http://jsfcentral.com/** . Безліч статей та навчально- ков по JSF зібрано на сайті **http://www.jsftutorials.net/** . Там ви можете знайти дальнайніші посилання.

Запитання для самоперевірки

1. Для чого придуманий мова JSP?
2. Можна, можливо чи змішувати код JSP та код HTML на однієї сторінці?
3. Можна, можливо чи записувати код Java на сторінках JSP?
4. Можна, можливо чи включати в сторінку JSP інші файли?
5. Можна, можливо чи передавати управління з сторінки JSP іншим ресурсів?
6. Можна, можливо чи розширити набір стандартних тегів JSP?
7. Можна, можливо чи написати кілька класів Java, по різному обробних один і той ж користувальницький тег?
8. Можна, можливо чи обробити користувальницький тег не класом Java, а сторінкою JSP?
9. Як підключити бібліотеку користувальницьких тегів до сторінці JSP?

ГЛАВА 28

Зв'язок Java з технологією XML

У розвитку Web-технології величезну роль відіграла мова HTML (HyperText Markup Language) - мова розмітки гіпертексту. Будь-яка людина, зовсім не знайома з програмуванням, міг за півгодини зрозуміти принцип розмітки тексту і за пару днів вичитати HTML теги. Користуючись найпростішим текстовим редактором, він міг написати свою сторінку HTML, відразу подивитися її у своєму браузері, випробувати почуття глибокого задоволення і гордо виставити в Інтернеті свій шедевр.

Чудово! Не треба місяцями вивчати заплутані мови програмування, очевидно призначені тільки для яйцеголових "ботаніків", освоювати складні алгоритми, возитися з компіляторами та відладчиками, розмножувати свій витвір на дисках. Дуже Незабаром з'явилися текстові редактори, що позначають типовий "плоский" текст тегами HTML. Розробнику залишалося тільки поправляти готову сторінку HTML, створенутаким редактором.

Простота мови HTML призвела до вибухового зростання кількості сайтів, користувачів Інтер- немає та авторів численних Web-сторінок. Звичайні користувачі комп'ютерів відчули себе творцями, отримали можливість заявити про себе, висловити власні думки і почуття, знайти в Інтернет однодумців.

Обмежені можливості мови HTML швидко перестали задовольняти поднато-розробників, що відчули себе "профі". Набір тегів мови HTML стро- го визначено і має однаково розумітися всіма браузерами. Не можна ввести доповнення ні теги або вказати браузеру, як слід відображати на екрані вміст того чи іншого тегу. Введення *таблиць стилів* CSS (Cascading Style Sheet) та *включень на стороні сервера* SSI (Server Side Include) лише ненадовго зменшило невдоволення розробників. Професіоналу завжди не вистачає засобів розробки, він постійно ви- намагається потреба додати до них якийсь свій засіб, що дозволяє реалізовувати. вати все його фантазії.

Така можливість є. Ще в 1986 році стала стандартом мова створення мов раз- мітки SGML (Standard Generalized Markup Language), за допомогою якого і був створений дано мову HTML. Основна особливість мови SGML полягає в тому, що вона дозволена. ляє сформувати нову мову розмітки, визначивши її набір тегів. Кожен конкрет- ний набір тегів, створений по правилам SGML, постачається описом DTD (Document Type Definition) - *визначення типу документа* , що роз'яснює зв'язок ті- гів між собою і правила їх застосування. Спеціальна програма - драйвер прин-

тера або SGML-браузер — керується цим описом для друку або відображення. ня документа на екрані дисплея.

У цей же час виявилася ще одна, найважливіша сфера застосування мов ки — пошук та вибірка інформації. В даний час переважна більшість інформації зберігається у реляційних базах даних. Вони зручні для зберігання та використання. ка відомостей, представлених у вигляді таблиць: анкет, відомостей, списків тощо, але не- зручні для зберігання різних документів, планів, звітів, статей, книг, не надаються ним у вигляді таблиці. Тегами мови розмітки можна задати структурну, а не візу- альну розмітку документа, розбити документ на глави, параграфи та абзаци або на якісь інші елементи виділити важливі для пошуку ділянки документа. Легко на- писати програму, аналізуючу розмічений такими тегами документ та витяг- каючу з нього потрібну інформацію.

Мова SGML виявилася надто складною, що вимагала ретельного і об'ємного опису. вання елементів створюваної з його допомогою мови. Він застосовується тільки у великих проектах, наприклад створення єдиної системи документообігу великої фірми. Скажімо, man-сторінки Solaris Operational Environment написані на спеціально зроблелланною реалізацію мови SGML.

Золотою серединою між мовами SGML та HTML стала мова XML (eXtensible Markup Language) - мова розмітки, що розширюється. Це підмножина мови SGML, позбавлений- ное від зайвою складності, але що дозволяє розробнику Web-сторінок створювати власні теги. Мова XML досить широка, щоб можна було створити все потрібне. ні теги, і достатньо простий, щоб можна було швидко їх описати.

Організовуючи опис документа мовою XML, треба передусім продумати структуру документа. Наведемо приклад. Нехай ми вирішили, нарешті, упорядкувати записну книжку з адресами та телефонами. У ній записані прізвища, імена та по батькові род. вінників, товаришів по службі та знайомих, дні їх народження, адреси, що складаються з поштової індексу, міста, вулиці, будинки та квартири, та телефони, якщо вони є: робітники та будинки- ня. Ми вигадуємо теги для виділення кожного із цих елементів, продумуємо вкладеність елементів і отримуємо структуру, показану в лістингу 28.1.

Листинг 28.1. Пример XML-документа

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook SYSTEM "ntb.dtd">

<notebook>

  <person>

    <name>
      <first-name>Іван</first-name>
      <second-name>Петрович</second-name>
      <surname>Сидорів</surname>
    </name>

    <birthday>25.03.1977</birthday>

    <address>
      <street>Садова, 23-15</street>
```

```
      <city>Урюпінськ</city>
      <zip>123456</zip>
    </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
    </phone-list>

 </person>

 <person>

    <name>
      <first-name>Марія</first-name>
      <second-name>Петрівна</second-name>
      <surname>Сидорова</surname>
    </name>

    <birthday>17.05.1969</birthday>

    <address>
      <street>Ягідна, 17</street>
      <city>Жмеринка</city>
      <zip>234561</zip>
    </address>

    <phone-list>
      <home-phone>2334455</home-phone>
    </phone-list>

 </person>

</notebook>
```

Документ XML починається з необов'язкового прологу, що складається з двох частин.

У першій частині прологу - *оголошення XML* (XML declaration), - записаної в першій рядку лістингу 28.1, вказується версія мови XML, необов'язкове кодування доку- мента і зазначається, залежить чи цей документ від інших документів XML (`standalone="yes"/"no"`). за замовчуванням приймається кодування UTF-8.

Усі елементи документа XML обов'язково повинні бути у *кореневому елементі* (root element), в лістингу 28.1 це елемент `<notebook>` . Ім'я кореневого елемента вважає- ся ім'ям всього документа і вказується в другій частині прологу, званої *оголошення ленням типу документа* (document type declaration). (Не плутайте з визначенням типу документа DTD!) Ім'я документа записується після слова `DOCTYPE` . Оголошення типу документа записано у другому рядку лістингу 28.1. У цій частині прологу після слова `DOCTYPE` і імені документа в квадратних дужках йде опис DTD:

```
<!DOCYPE notebook [ Сюди заноситься опис DTD ]>
```

Дуже часто опис DTD складається відразу для кількох документів XML. В такому випадку його зручно записати окремо від документа. Якщо опис DTD відді-

лено від документа, то в другій частині прологу замість квадратних дужок вказується одне із слів: `SYSTEM` або `PUBLIC` . За словом `SYSTEM` йде URI файл з описом DTD, а за словом `PUBLIC` , крім того, можна, можливо записати додаткову інформацію.

Документ XML складається з *елементів* . Елемент починається *відкриваючим тегом* , так- ліє йде необов'язкове *тіло елемента* , потім - *Закриваючий тег* :

```
< Відкриваючий тег >Тіло елемента</ Закриваючий тег >
```

Закриваючий тег містить похилу рису, після якої повторюється ім'я ного тега.

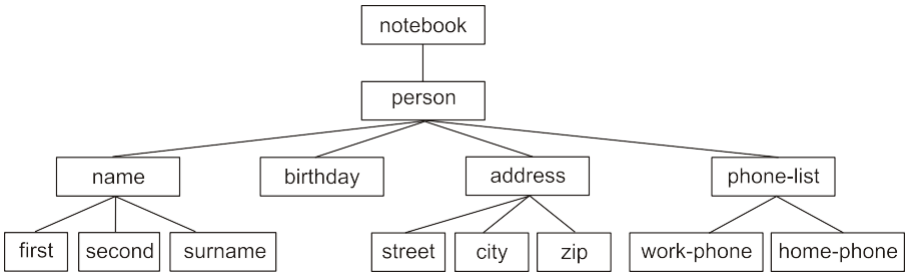
Мова XML, на відміну мови HTML, вимагає обов'язково записувати закривають теги. Якщо елемент не має тіла і закриває тега (empty — *порожній елемент*), то його відкриває тег повинен закінчуватися символами " `/>` ", наприклад:

```
<br> />
```

УВАГА ! _

Відразу треба сказати, що мова XML, в відмінність від HTML, розрізняє регістри літер.

З лістингу 28.1 видно, що елементи документа XML можуть бути вкладені один в іншому га. Треба стежити, щоб елементи не перетиналися, а повністю вкладалися один в одного. Як уже говорилося раніше, всі елементи, що становлять документ, вкладаються в ні в кореневий елемент цього документа. Тим самим документ наділяється структурою дерева вкладених елементів. На рис. 28.1 показано структуру адресної книжки, опи- санною в лістингу 28.1.



Мал. 28.1. Дерево елементів документа XML

У тегів XML , що відкривають , можуть бути *атрибути* . Наприклад, ім'я, по батькові та фамі- лію можна, можливо записати як атрибути , _ `second` і `surname` тега `<name>` :

```
<name first="Іван" second="Петрович" surname="Сидорів" />
```

На відміну від мови HTML у мові XML значення атрибутів обов'язково треба заключати в лапки або в апострофи.

Атрибути є зручними для опису простих значень. У кожного громадянина Росії, уваКого паспортний режим, обов'язково є одне ім'я, одне по батькові і одна фамі- лія. Їх краще записувати атрибутами. Але у громадянина Росії може бути кілька телефонів, тому їх номери зручніше оформити як елементи `<work-phone>` та `<home- phone>` , вкладені в елемент `<phone-list>` , а не атрибути відкриває тега `<phone-`

list> . Зауважте, що елемент <name> з атрибутами порожній, у нього ні тіла, отже, не потрібен тег, що закриває. Тому тег <name> з атрибутами завершується символами " /> ". У лістингу 28.2 наведено змінену адресна книга.

Листинг 28.2. Пример XML-документа с атрибутами в открывающем теге

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook SYSTEM "ntb.dtd">

<notebook>

 <person>

    <name first="Іван" second="Петрович" surname="Сидорів" />

    <birthday>25.03.1977</birthday>

    <address>
      <street>Садова, 23-15</street>
      <city>Урюпінськ</city>
      <zip>123456</zip>
    </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
    </phone-list>

 </person>

 <person>

    <name first="Марія" second="Петрівна" surname="Сидорова" />

    <birthday>17.05.1969</birthday>

    <address>
      <street>Ягідна, 17</street>
      <city>Жмеринка</city>
      <zip>234561</zip>
    </address>

    <phone-list>
      <home-phone>2334455</home-phone>
    </phone-list>

 </person>

</notebook>
```

Атрибути відкриває тега зручні і для вказівки типу елемент. Наприклад, ми не уточнюємо, в місті живе наш родич, в селищі або в селі. Можна, можливо ввести

в тег <city> атрибут type , що приймає одне з значень: місто , селище , село . Наприклад:

```
<City type="місто">Москва</city>
```

Для описи адресною книжки нам знадобилися відкривають теги <notebook> , <person> , <name> , <address> , <street> , <city> , <zip> , <phone-list> , <work-phone> , <home-phone> та відповідні їм закривають теги, позначені похилою межею. Тепер необ- ходимо дати їх опис. В описі вказуються тільки найзагальніші ознаки ло- гічною взаємозв'язку елементів і їх тип.

- Елемент <notebook> може утримувати ні одного, один або більше одного елемента <person> і більше нічого.
- Елемент <person> містить рівно один елемент <name> , ні одного, один або кілька- до елементів <address> і ні одного або тільки один елемент <phone-list> .
- Елемент <name> порожній.
- У відкриває тезі <name> три атрибуту: , _ second _ surname , значення яких -рядки символів.
- Елемент <address> містить по одному елементу <street> , <City> і <zip> .
- Елементи <street> і <City> мають по однієї текстовий рядку.
- Елемент <zip> містить одне ціле число.
- У відкриває тега <City> є один необов'язковий атрибут type , приймаючийодне з трьох значень: Місто , селище або село . Значення по замовчуванням - Місто .
- Необов'язковий елемент <phone-list> містить ні одного, один або кілька еле-ментів <work-phone> і <home-phone> .
- Елементи <work-phone> і <home-phone> містять по однієї рядку, що складається тільки з цифр.

Цей словесний опис, званий *схемою* документа XML, формалізується кілько- ними способами. Найбільш поширені два способи: можна зробити опис DTD, що прийшов у XML із SGML, або описати схему на мові XSD (XML Schema Definition Language).

Опис DTD

Опис DTD нашої адресною книжки записано в лістингу 28.3.

Листинг 28.3. Описание DTD документа XML

```
<!ELEMENT notebook (person) *>
<!ELEMENT person (name, birthday?, address*, phone-list?)>
<!ELEMENT name EMPTY>
<!ATTLIST name
   first  CDATA #IMPLIED
   second CDATA #IMPLIED
   surname CDATA #REQUIRED>
<!ELEMENT birthday (#PCDATA)>
```

```
<!ELEMENT address (street, city, zip)?>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ATTLIST city
     type (Місто | селище | село) "місто">
<!ELEMENT zip (#PCDATA)>
<!ELEMENT phone-list (work-phone*, home-phone*)>
<!ELEMENT work-phone (#PCDATA)>
<!ELEMENT home-phone (#PCDATA)>
```

Як бачите, опис DTD майже очевидний. Воно повторює наведене раніше словес- ний опис. Перше слово `ELEMENT` означає, що елемент може містити тіло з вкладеними елементами. Вкладені елементи перераховуються у круглих дужках. Поря- док перерахування вкладених елементів у дужках повинен відповідати порядку їх появи у документі. Слово `EMPTY` у третьому рядку лістингу 28.3 означає порожній елемент.

Слово `ATTLIST` починає опис списку атрибутів елемента. Для кожного атрибуту вказується ім'я, тип та обов'язковість присутності атрибута. Типів атрибуту всього дев'ять, але найчастіше використовується тип `CDATA` (Character DATA), що означає довільний рядок символів Unicode, чи перераховуються значення типу. Так зроблено в опис атрибута `type` тега `<city>` , що приймає одне з трьох значень: `місто` , `селище` або `село` . У лапках показано значення по замовчуванням - `Місто` .

Обов'язковість вказівки атрибуту відзначається одним з трьох слів:

- `#REQUIRED` - атрибут обов'язковий;
- `#IMPLIED` - атрибут необов'язковий;
- `#FIXED` - значення атрибуту фіксовано, воно задається в DTD.

Першим словом можуть бути, крім слів `ELEMENT` або `ATTLIST` , слова `ANY` , `MIXED` або `ENTITY` . Слова `ANY` і `MIXED` означають, що елемент може містити прості дані та/або володіти ні елементи. Слово `ENTITY` служить для позначення або адреси даних, наведеної в описі DTD, так званою *сутності* .

Після імені елемента в дужках записуються вкладені елементи або тип даних, що містяться в тілі елемента. Тип `PCDATA` (Parsed Character DATA) означає рядок символів Unicode, яку треба інтерпретувати. Тип `CDATA` - рядок символів Unicode, яку не слід інтерпретувати.

Зірочка, записана після імені елемента, означає "нуль або більше входжень" елемента, після якого вона стоїть, а плюс - "одне або більше входжень". Запитання- ний знак означає "нуль або один раз". Якщо ці символи відносяться до всіх вла- ним елементам, їх можна вказати після круглої дужки, що закриває список вкладених елементів.

Опис DTD можна занести в окремий файл, наприклад ntb.dtd, вказавши його ім'я другої частини прологу, як показано у другому рядку лістингів 28.1 та 28.2. Можна, можливо включити опис у другу частину прологу XML-файлу, уклавши його у квадратні дужки:

```
<!DOCTYPE notebook [ Опис DTD ]>
```

Після того, як створено опис DTD нашої реалізації XML і написано документ, Розмічений тегами цієї реалізації, слід перевірити правильність їхнього написання. Для цього є спеціальні програми - *перевіряючі аналізатори* (validating parsers). Всі фірми, що розробляють засоби для роботи з XML, випускають без- платні або комерційні аналізатори, що перевіряють.

Перевіряючий аналізатор корпорації Sun Microsystems міститься у пакеті класів JAXP (Java API for XML Processing), що входить до складу Java SE. Крім того, цей пакет можна, можливо завантажити окремо з адреси **http://jaxp.java.net/** .

Корпорація Microsoft поставляє аналізатор MSXML (Microsoft XML). Parser), доступний за адресою **http://msdn.microsoft.com/xml/** .

Є ще безліч аналізаторів, що перевіряють, але лідером серед них є, по- мабуть, Apache Xerces2, що входить у багато засобів обробки документів XML, ви- пускаються іншими фірмами. Він вільно доступний по адресою **http://xerces.apache.org/xerces2-j/** .

Обмежені засоби DTD не дозволяють повністю описати структуру документа XML. Зокрема, опис DTD не вказує точну кількість повторень вкладених них елементів, воно не задає точний тип тіла елемента. Наприклад, у лістингу 28.3 з Опису DTD не видно, що в елементі `<birthday>` міститься дата народження. Ці не- статки DTD призвели до появи інших схем опису документів XML. Найбільш розвинений опис дає мову XSD. Ми будемо називати опис цією мовою просто *схемою XML* (XML Schema).

Подивимося, як створюються схеми XML, але спочатку познайомимося ще з одним поняттям. їм XML - Простором імен.

Простору імен XML

Оскільки в різних мовах розміток – реалізаціях XML – можуть зустрітися одні та ті ж імена тегів та їх атрибутів, що мають зовсім різний зміст, а в документі XML їх часто доводиться змішувати, аналізатору треба дати можливість їх якось розрізняти. У мові Java ми імена полів класу уточнюємо ім'ям класу, а імена клас- сов уточнюємо вказівкою пакета, що дозволяє назвати поле просто ім'ям `a` , застосовуючи при необхідності повне ім'я, щось на зразок `com.mydomain.myhost.mypack.MyClass.a` . При цьому рекомендується пакет для унікальності називати доменним ім'ям сайту розробника, записуючи його праворуч наліво.

У мові XML прийнято таку схему: локальне ім'я уточнюється ідентифікатором. Ідентифікатор має бути унікальним у всьому Інтернеті, тому ним має бути рядок символів, які мають форму адреси URI. Адреса може відповідати чи не відповідати якомусь реальному сайту Інтернету це не важливо. Важливо, щоб він не повторювався в Інтернеті, можна записати, наприклад, рядок `http://some.firm.com/ 2008/ntbml` . Усе імена з одним і тим ж ідентифікатором утворюють одне *простірімен* (Namespace).

Оскільки ідентифікатор простору імен виходить дуже довгим, було б дуже незручно записувати його перед локальним ім'ям. У мові Java для спів- покращення записи імен ми використовуємо оператор `import` . У XML прийнятий інший підхід.

У кожному документі ідентифікатор простору імен пов'язується з деяким ко- ротким префіксом, що відокремлюється від локального імені двокрапкою. Префікс визначається атрибутом `xmlns` наступним чином:

```
<ntb:notebook xmlns:ntb = " http://some.firm.com/2008/ntbml ">
```

Як бачите, префікс `ntb` тільки що визначено, але його вже можна, можливо використовувати в імені
`ntb:notebook` .

Після такого опису імена тегів та атрибутів, які ми хочемо віднести до про- подорожі імен `http://some.firm.com/2008/ntbml` , постачаються префіксом `ntb` , наприклад:

```
<ntb:city ntb:type="селище">Горелове</ntb:city>
```

Ім'я разом з префіксом, наприклад `ntb:city` , називається *розширеним* або *уточненим ім'ям* (Qualified Name, QName).

Хоча ідентифікатор простору імен записується у формі адреси URI, такої як `http://some.firm.com/2008/ntbml` , аналізатор документа XML та інші програми, якими користуються документом, не звертатимуться за цією адресою. Там навіть може не бути ніякий Web-сторінки взагалі. Просто ідентифікатор простору імен повинен бути унікальним у всьому Інтернеті, та розробники рекомендації щодо застосування простору імен, яку можна переглянути за адресою **http://www.w3.org/TR/REC- xml-names** справедливо вирішили, що буде зручно використовувати для нього DNS-ім'я сайту, де розміщено визначення простору імен. Дивіться на URI просто як на рядок символів, що ідентифікує простір імен. Зазвичай вказується URL фірми, що створила цю реалізацію XML, або ім'я файлу з описом схеми XML.

Оскільки ідентифікатор - це рядок символів, то і порівнюються вони як рядки, з урахуванням регістру символів. Наприклад, ідентифікатор `http://some.firm.com/2008/Ntbml` буде вважатися XML-аналізатором, відмінним від ідентифікатора `http://some.firm.com/ 2008/ntbml` , введеного нами раніше, і буде визначати інше простір імен.

За правилами SGML і XML двокрапка може застосовуватися в іменах як звичайний сімвіл, тому ім'я з префіксом це просто фокус, аналізатор розглядає його як просте ім'я. Звідси випливає, що у описі DTD не можна опускати префікси імен. Деяким аналізаторам треба спеціально вказати на необхідність урахування простору імен. Наприклад, при роботі з аналізатором Xerces слід застосувати метод `setNamespaceAware(true)` .

Атрибут `xmlns` , визначальний префікс імен, може з'явитися в будь-кому елементі XML, а не лише у кореневому елементі. Певний ним префікс можна застосовувати в тому елементі, в якому записано атрибут `xmlns` , і у всіх вкладених у нього елементах. Більше того, в одному елементі можна, можливо визначити кілька просторів імен:

```
<ntb:notebook xmlns:ntb = " http://some.firm.com/2008/ntbml "
              xmlns:bk  = " http://some.firm.com/2008/bookml ">
```

Поява імені тега без префікса в документі, який використовує простір імен, означає, що ім'я належить *простору стандартних імен* (default namespace). Наприклад, мова XHTML допускає застосування тегів HTML і XML в одному документі. ті. Допустимо, ми визначили тег з ім'ям `title` . Щоб аналізатор не прийняв його за один з тегів HTML, чинимо наступним чином:

```
<html xmlns=" http://www.w3.org/1999/xhtml "
      xmlns:ntb = " http://some.firm.com/2002/ntbml
      ">

   <head>
      <title>Моя бібліотека</title>
   </head>

   <body>
      <ntb:book>
          <ntb:title>Створення Java Web Services</ntb:title>
      </ntb:book>
   </body>
</html>
```

У цьому прикладі простором імен по замовчуванням стає простір імен XHTML, що має загальновідомий ідентифікатор `http://www.w3.org/1999/xhtml` , та теги, що відносяться до цього простору імен записуються без префіксу.

Атрибути ніколи не входять у простір стандартних імен. Якщо ім'я атрибута записано без префікса, то це означає, що атрибут не відноситься до жодного про- подорожі імен.

Префікс імені не відноситься до ідентифікатора простору імен і може бути раз- ним у різних документах. Наприклад, у якомусь іншому документі ми можемо на- писати кореневий елемент

```
<nb:notebook xmlns:nb = " http://some.firm.com/2008/ntbml ">
```

і записувати елементи з префіксом `nb` :

```
<nb:city nb:type="селище">Горелове</nb:city>
```

Більш того, можна пов'язати кілька префіксів з одним і тим самим ідентифікатором простору імен навіть в одному документі, але це може призвести до плутанини, по- му застосовується рідко.

Тепер, після того як ми запровадили поняття простору імен, можна звернутися до схеми XML.

Схема XML

У травні 2001 року консорціум W3C рекомендував описувати структуру документів XML мовою опису схем XSD (XML Schema Definition Language). цією мовою складаються *схеми XML* (XML Schema), описують елементи документів XML.

Схема XML сама записується як документ XML. Його елементи називають *компонен-тами* (components), щоб відрізнити їхню відмінність від елементів описуваного документа XML. Кореневий компонент схеми носить ім'я `<Schema>` . Компоненти схеми описують елемен- ти XML і визначають різні типи елементів. Рекомендація схеми XML, яку можна знайти за посиланнями, записаними за адресою **http://www.w3.org/XML/Schema.html** , перераховує 13 типів компонентів, але найважливіші компоненти, що визначають прості і складні типи елементів, самі елементи та їхні атрибути.

Мова XSD розрізняє прості та складні елементи XML. *Простими* (simple) елемен- тами описуваного документа XML вважаються елементи, не містять атрибутів і

вкладених елементів. Відповідно, *складні* (complex) елементи містять атрибу- ти та/або вкладені елементи. Схема XML описує *прості типи* - типи простих елементів, та *складні типи* - Типи складних елементів.

Мова описи схем містить багато вбудованих простих типів. Вони перераховані в наступному розділ.

Вбудовані прості типи XSD

Вбудовані типи мови опису схем XSD дозволяють записувати двійкові та десятки. цільні числа, речові числа, дату і час, рядки символів, логічні значення, адреси URI. Розглянемо їх по порядку.

Речові числа

Речові числа в мовою XSD розділені на три типу: `decimal` , `float` і `double` .

Тип `decimal` становлять речові числа, записані з фіксованою точкою: 123.45, -0.1234567689345 і т. д. Фактично зберігаються два цілих числа: мантіс і по- рядок. Специфікація мови XSD не обмежує кількість цифр у мантисі, але тре- буде, щоб можна було записати щонайменше 18 цифр. Цей тип легко реалізується клас- сом `java.math.BigDecimal` , описаним в *главі 4* .

Типи `float` та `double` підходять стандарту IEEE754-85 і однойменних типів Java. Вони записуються з фіксованою або з плаваючою десятковий точкою.

Цілі числа

Основний цілий тип `integer` розуміється як підтип типу `decimal` , містить числа з нульовим порядком. Це цілі числа з будь-якою кількістю десяткових цифр: –34567, 123456789012345 і т. д. Даний тип легко реалізується класом `java.math.BigInteger` , описаним в *главі 4* .

Типи `long` , `int` , `short` та `byte` повністю відповідають однойменним типам Java. Вони розуміються як підтипи типу `integer` , типи більш коротких чисел вважаються підтипу- ми більше довгих чисел: тип `byte` - це підтип типу `short` , обидва вони підтипи типу `int` і т.д.

Типи `nonPositiveInteger` і `negativeInteger` - підтипи типу `integer` - складені з не- позитивних і негативних чисел відповідно з будь-яким кількістю цифр.

Типи `nonNegativeInteger` і `positiveInteger` - підтипи типу `integer` - складені з не- негативних і позитивних чисел відповідно з будь-яким кількістю цифр.

У типу `nonNegativeInteger` є підтипи беззнакових цілих чисел `unsignedLong` , `unsignedInt` , `unsignedShort` і `unsignedByte` .

Рядки символів

Основний символьний тип `string` визначає довільний рядок символів Unicode. Його можна, можливо реалізувати класом `java.lang.String` .

Тип `normalizedString` - підтип типу `string` - це рядки, що не містять символів пе- ревода рядки `'\n'` , повернення каретки `'\r'` і символи горизонтальною табуляції `'\t'` .

У рядках типу token - підтипу типу normalizedString - немає, крім того, початкових і завершальних прогалин і ні кількох поспіль ідучих пробілів.

У типі token виділено три підтипу. Підтип language визначено для записи назви мови згідно з RFC 1766, наприклад: ru , en , de , fr . Підтип NMTOKEN використовується тільки в атрибутах для записи їх перерахованих значень. Підтип name становлять *імена XML* - послідовності букв, цифр, дефісів, точок, двокрапок, знаків підкреслення. вання, що починаються з літери (крім зарезервованої послідовності літер X , x , M , m , L , l у будь-якому поєднанні регістрів) або знака підкреслення. Двокрапка в значенні- ях типу name використовується для виділення префікса простору імен.

З типу name виділено підтип NCName (Non-Colonized Name) імен, що не містять двох- точки, у якому, своєю чергою, визначено три підтипу: ID , ENTITY , IDREF , описи- ваючі ідентифікатори XML, сутності та перехресні посилання.

Дата та час

Тип duration описує проміжок часу, наприклад запис P1Y2M3DT10H30M45S чає один рік (1Y), два місяця (2M), три дня (3D), десять годин (10H), тридцять хвилин (30M) і сорок п'ять секунд (45S). Запис може бути скороченим, наприклад P120M означає 120 місяців, а T120M - 120 хвилин.

Тип dateTime містить дату та час у форматі CCYY-MM-DDThh:mm:ss , наприклад 2008-04-25T09:30:05 . Інші типи виділяють якусь частина дати або часу.

Тип time містить час в звичайному форматі hh:mm:ss .

Тип date містить дату в форматі CCYY-MM-DD .

Тип gYearMonth виділяє рік і місяць в форматі CCYY-MM .

Тип gMonthDay містить місяць і день місяця в форматі -MM-DD .

Тип gYear означає рік в форматі CCYY , тип gMonth - місяць в форматі -MM- , тип gDay - день місяця у форматі -DD .

Двійкові типи

Двійкові цілі числа записуються або в шістнадцятковій формі без будь-яких до-додаткових символів: 0B2F , 356C0A і т. д., це тип hexBinary , або кодування Base64, це тип base64Binary .

Інші вбудовані прості типи

Ще три вбудованих простих типу описують значення, часто використовувані в документах XML.

Адреси URI відносяться до типу anyURI .

Розширене ім'я тега або атрибута (qualified name), тобто ім'я разом з префіксом, від-діленим від імені двокрапкою, - це тип QName .

Елемент NOTATION опису DTD виділено як окремий тип схеми XML. Його використовують для запису математичних, хімічних та інших символів, нот, абетки Брайля та інших позначень.

Визначення простих типів

У схемах XML за допомогою вбудованих типів можна трьома способами визначити нові типи простих елементів. Вони вводяться як *звуження* (restriction) вбудованого або раніше певного простого типу, *перелік* (list) або *об'єднання* (union) простих типів.

Простий тип визначається компонентом схеми `<simpleType>` , мають вигляд:

```
<xsd:simpleType name=" ім'я типу "> Визначення типу </xsd:simpleType>
```

Звуження

Звуження простого типу визначається компонентом `<restriction>` , в якому атрибут `base` вказує простий тип, що звужується, а в тілі задаються обмеження, що виділяють визначуваний простий тип. Наприклад, поштовий індекс `zip` можна встановити як шість арабських цифр наступним чином:

```
<xsd:simpleType name="zip">

  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{6}" />
 </xsd:restriction>

</xsd:simpleType>
```

Можна дати інше визначення простого типу `zip` як цілого позитивного числа, що знаходиться в діапазоні від 100 000 до 999 999:

```
<xsd:simpleType name="zip">

  <xsd:restriction base="xsd:positiveInteger">
    <xsd:minInclusive value="100000" />
    <xsd:maxInclusive value="999999" />
  </xsd:restriction>

</xsd:simpleType>
```

Теги `<pattern>` , `<maxInclusive>` та інші теги, задають обмеження, називаються *фа-сітками* (Facets). Ось їх список:

- `<maxExclusive>` - найбільше значення, воно вже не входить в визначається тип;
- `<maxInclusive>` - найбільше значення визначається типу;
- `<minExclusive>` - найменше значення, вже не вхідне в визначається тип;
- `<minInclusive>` - найменше значення визначається типу;
- `<totalDigits>` - загальне кількість цифр в що визначається числовому типі - звуженнятипу `decimal` ;
- `<fractionDigits>` - кількість цифр в дробовий частини числа;
- `<length>` - довжина значень визначається типу;
- `<maxLength>` - найбільша довжина значень визначається типу;

- `<minLength>` - найменша довжина значень визначається типу;
- `<enumeration>` - одне з перерахованих значень;
- `<pattern>` - регулярне вираз [8];
- `<whiteSpace>` - застосовується при звуженні типу `string` і визначає спосіб перетворення зування пробельних символів `'\n'` , `'r'` , `'t'` . Атрибут `value` цього тега приймає одне з трьох значень:
  - `preserve` - не прибирати пробільні символи;
  - `replace` - замінити пробільні символи пробілами;
  - `collapse` _ після заміни пробільних символів пробілами прибрати початкові і кінцеві прогалини, а з кількох поспіль пробілів, що йдуть, залишити тільки один.

У тегах-фасетках можна записувати такі атрибути, які називаються *базовими фасітками* (Fundamental facets):

- `ordered` - задає впорядкованість визначеного типу, приймає одне з трьох значень:
  - `false` - тип невпорядкований;
  - `partial` - тип частково упорядкований;
  - `total` - тип повністю упорядкований;
- `bounded` - ставить обмеженість або необмеженість типу значеннями `true` або `false` ;
- `cardinality` - ставить кінцівка або нескінченність типу значеннями `finite` або `countably infinite` ;
- `numeric` - показує, числовий цей тип або ні, значеннями `true` або `false` .

Як видно з наведених раніше і далі прикладів, в одному звуженні може бути не- скільки обмежень-фасеток. При цьому фасетки `<pattern>` і `<enumeration>` задають неза- висимо один від одного обмеження, їх можна подумки об'єднати союзом "або". Інші фасетки задають загальні обмеження, що спільно накладаються, їх можна подумки об'єднати союзом "і".

перелік

Простий тип-список - це тип елементів, в тілі яких записується, через пробіл, кілька значень одного й того самого простого типу. Наприклад, у документі XML мо- жет зустрітися такий елемент, містить список цілих чисел:

```
<days>21 34 55 46</days>
```

перелік визначається компонентом `<list>` , в якому атрибутом `itemType` вказується тип елементів визначуваного списку. Тип елементів списку можна вказати і в тілі елемента `<list>` . Наприклад, показаний раніше елемент документа XML `<days>` можна визначити в схемою так:

```
<xsd:element name="days" type="listOfInteger" />
```

а використаний при його визначенні тип `listOfInteger` задати як перелік не більшечим з п'яти цілих чисел наступним чином:

```
<xsd:simpleType name="listOfInteger">

<xsd:restriction>

    <xsd:simpleType>
      <xsd:list itemType="xsd:integer" />
 </xsd:simpleType>

 <xsd:maxLength value="5" />

</xsd:restriction>

</xsd:simpleType>
```

При визначенні списку можна, можливо застосовувати фасетки `<length>` , `<minLength>` , `<maxLength>` ,
`<enumeration>` , `<pattern>` . У наведеному прикладі перелік - тіло елемента `<days>` - не може утримувати понад п'ять чисел.

Об'єднання

Простий тип-об'єднання визначається компонентом `<union>` , в якому атрибутом `memberTypes` можна, можливо вказати імена об'єднаних типів. Наприклад:

```
<xsd:union memberTypes="xsd:string xsd:integer listOfInteger" />
```

Інший спосіб - записати в тілі компонента `<union>` визначення простих типів, вхо-діючих у об'єднання. Наприклад:

```
<xsd:attribute name="size">

 <xsd:simpleType>
   <xsd:union>

      <xsd:simpleType>

            <xsd:restriction base="xsd:positiveInteger">
              <xsd:minInclusive value="8"/>
              <xsd:maxInclusive value="72"/>
       </xsd:restriction>

      </xsd:simpleType>

      <xsd:simpleType>

            <xsd:restriction base="xsd:NMTOKEN">
              <xsd:enumeration value="small"/>
              <xsd:enumeration value="medium"/>
              <xsd:enumeration value="large"/>
      </xsd:restriction>

      </xsd:simpleType>
```

```
   </xsd:union>
 </xsd:simpleType>

</xsd:attribute>
```

Після цього атрибут `size` можна, можливо використовувати, наприклад, так:

```
<font size='large'>Глава 28</font>
<font size='12'>Простий текст</font>
```

Опис елементів і їх атрибутів

Елементи, які будуть застосовуватися в документі XML, описуються в схемою компонентом `<element>` :

```
<xsd:element name=" ім'я елемента " type=" тип елемента "
  minOccurs=" найменше число появ елемента в документі "
  maxOccurs=" найбільше число появ " />
```

Значення за умовчанням необов'язкових атрибутів `minOccurs` та `maxOccurs` дорівнює 1. Це означає, що якщо ці атрибути відсутні, то елемент повинен з'явитися в документі XML рівно один разів. Визначення типу елемента можна, можливо винести в тіло елемента
`<element>` :

```
<xsd:element name=" ім'я елементу " >
   Визначення типу елемента
</xsd:element>
```

Опис атрибуту елемента теж не складно:

```
<xsd:attribute name=" ім'я атрибута " type=" тип атрибута " use="
  обов'язковість атрибуту " default=" значення по замовчуванням _ />
```

Необов'язковий атрибут `use` приймає три значення:

- `optional` - описуваний атрибут необов'язковий (це значення по замовчування);
- `required` - описуваний атрибут обов'язковий;
- `prohibited` - описуваний атрибут не застосовується. Це значення корисно при визначалінування підтипу, щоб скасувати деякі атрибути базового типу.

Якщо описуваний атрибут необов'язковий, то атрибутом `default` можна, можливо поставити його зна-чення за замовчуванням.

Визначення типу атрибуту - а це повинен бути простий тип - можна винести в тіло елемента `<attribute>` :

```
<xsd:attribute name=" ім'я атрибута
   ">Тип атрибуту
</xsd:attribute>
```

Визначення складних типів

Нагадаємо, що тип елемента називається складним, якщо в елемент вкладені інші елементи та/або в відкриває тезі елемента є атрибути.

Складний тип визначається компонентом <complexType> , мають вигляд:

```
<xsd:complexType name=" ім'я типу " > Визначення типу </xsd:complexType>
```

Необов'язковий атрибут name визначає ім'я типу, а в тілі компонента <complexType> описуються елементи, що входять до складний тип, та/або атрибути відкриває тега.

Визначення складного типу можна розділити визначення типу порожнього елемента, елемента з простим тілом та елемента, що містить вкладені елементи. Розглянемо ці визначення Детальніше.

Визначення типу порожній елемент

Найпростіше визначається тип порожнього елемента - елемента, що не має тіла, а сотримає тільки атрибути в відкриває тег. Такий, наприклад, елемент <name> із лістингу 28.2. Кожен атрибут описується одним компонентом <attribute> , на- приклад:

```
<xsd:complexType name="imageType">
  <xsd:attribute name="href" type="xsd:anyURI" />
</xsd:complexType>
```

Після цього визначення можна, можливо в схемою описати елемент <image> типу imageType :

```
<xsd:element name="image" type="imageType" />
```

а в документі XML використовувати це опис:

```
<image href=" http://some.com/images/myface.gif " />
```

Визначення типу елемента з простим тілом

Трохи складніший опис елемента, що містить тіло простого типу та атрибути в відкриває тег. Цей тип відрізняється від простого типу лише наявністю атрибутів та визначається компонентом <simpleContent> . У тілі даного компонента має бути або компонент <restriction> , або компонент <extension> , атрибутом base, що задає тип (простий) тіла описуваного елемента.

У компоненті <extension> описуються атрибути тега, що відкриває. елемент. Усе разом виглядає так як в наступному приклад:

```
<xsd:complexType name="calcResultType">

   <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
         <xsd:attribute name="unit" type="xsd:string" />
         <xsd:attribute name="precision"
                        type="xsd:nonNegativeInteger" />
      </xsd:extension>
</xsd:simpleContent>

</xsd:complexType>
```

Цю конструкцію можна, можливо описати словами так: "Визначається тип calcResultType еле- мента, тіло якого містить значення вбудованого простого типу xsd: decimal . Простій тип розширюється тим, що до ньому додаються атрибути unit і precision ".

Якщо в схемою описати елемент <result> цього типу наступним чином:

```
<xsd:element name="result" type="calcResultType" />
```

то у документі XML можна, можливо написати

```
<result unit="см" precision="2">123.25</result>
```

У компоненті <restriction> крім атрибутів описується простий тип тіла елемента та/або фасетки, обмежують тип, заданий атрибутом base . Наприклад:

```
<xsd:complexType name="calcResultType">

 <xsd:simpleContent>
     <xsd:restriction base="xsd:decimal">
        <xsd:totalDigits value="8" />
        <xsd:attribute name="unit" type="xsd:string" />
        <xsd:attribute name="precision"
                       type="xsd:nonNegativeInteger" />
     </xsd:restriction>
</xsd:simpleContent>

</xsd:complexType>
```

Визначення типу вкладених елементів

Якщо значеннями складного типу, що визначається, будуть елементи, що містять вкладений- ні елементи, як, наприклад, елементи <address> , <phone-list> лістингу 28.2, то перед переліком опису вкладених елементів слід вибрати *модель групи* (model group) вкладених елементів. Справа в тому, що вкладені елементи, що становлять оп- тип, що виділяється, можуть з'являтися або в певному порядку, або в довільному Крім того, можна вибирати тільки один з перерахованих елементів. Ця можливість називається *моделлю групи* елементів. Вона визначається одним із трьох компонентів: <sequence> , <all> або <choice> .

Компонент <sequence> застосовується в том випадку, коли перераховані елементи должні записуватись у документі в конкретному порядку. Нехай, наприклад, ми описуємо книгу. Спочатку визначаємо тип:

```
<xsd:complexType name="bookType">

  <xsd:sequence maxOccurs="unbounded">
    <xsd:element name="author" type="xsd:normalizedString"
                 minOccurs="0" />
    <xsd:element name="title" type="xsd:normalizedString" />
    <xsd:element name="pages" type="xsd:positiveInteger"
                 minOccurs="0" />
    <xsd:element name="publisher" type="xsd:normalizedString"
                 minOccurs="0" />
  </xsd:sequence>

</xsd:complexType>
```

Потім описуємо елемент:

```
<xsd:element name="book" type="bookType" />
```

Елементи <author> , <title> , <pages> і <publisher> повинні входити в елемент <book> саме в такому порядку. У документі XML треба писати:

```
<book>
  <author>I. Ільф, е. Петров</author>
  <title>Золотий теля</title>
  <publisher>Художня література</publisher>
</book>
```

Якщо ж замість компонента <xsd:sequence> записати компонент <xsd:all> , то елементи <author> , <title> , <pages> і <publisher> можна, можливо перераховувати в будь-кому порядку.

Компонент <choice> застосовується у разі, коли треба вибрати одне із кількох елементів. Наприклад, при описі журналу замість видавництва, що описується еле- ментом <publisher> , слід записати назва журналу. Це можна, можливо визначити так:

```
<xsd:complexType name="bookType">

<xsd:sequence maxOccurs="unbounded">

    <xsd:element name="author" type="xsd:normalizedString"
                 minOccurs="0" />
    <xsd:element name="title" type="xsd:normalizedString" />
    <xsd:element name="pages" type="xsd:positiveInteger"
                 minOccurs="0" />

    <xsd:choice>
          <xsd:element name="publisher" type="xsd:normalizedString"
                       minOccurs="0" />
          <xsd:element name="magazine" type="xsd:normalizedString"
                       minOccurs="0" />
  </xsd:choice>

</xsd:sequence>

</xsd:complexType>
```

Як видно з наведеного прикладу, компонент <choice> можна, можливо вкласти в компонент <sequence> . Можна, навпаки, вкласти компонент <sequence> у компонент <choice> . Та- кі вкладення допустимо робити скільки завгодно разів. Крім того, кожна група у цих моделях може з'явитися скільки завгодно разів, тобто у компоненті <choice> теж дозволено записати атрибут maxOccurs="unbounded" .

Модель групи <all> відрізняється в цьому від моделей <sequence> і <choice> . У компоненті <all> не допускається застосування компонентів <sequence> та <choice> . Назад, в компо- Нентах <sequence> та <choice> не можна застосовувати компонент <all> . Кожен елемент, вхо- що дає в групу моделі <all> , може з'явитися не більше одного разу, тобто атрибут maxOccurs цього елемента може дорівнювати тільки одиниці.

Визначення типу зі складним тілом

При визначенні складного типу можна скористатися вже певним, *базовим* , складним типом, розширивши його додатковими елементами або, навпаки, вилучивши з нього деякі елементи. Для цього необхідно застосувати компонент `<complexContent>` . В цьому компоненті, так ж як і в компоненті `<simpleContent>` , записується або компонент `<extension>` , якщо потрібно розширити базовий тип, або компонент `<restriction>` , якщо необхідно звузити базовий тип. Базовий тип вказується атрибутом `base` , так само як і при записи компонента `<simpleContent>` , але тепер це повинен бути складний, а не про-стій тип!

Розширимо, наприклад, певний тип `bookType` , додавши рік видання — еле- мент `<year>` :

```
<xsd:complexType name="newBookType">
  <xsd:complexContent>
    <xsd:extension base="bookType">
      <xsd:sequence>
         <xsd:element name="year" type="xsd:gYear">
    </xsd:sequence>
  </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

При звуженні базового типу компонентом `<restriction>` треба перерахувати ті елементи, які залишаться після звуження. Наприклад, залишимо у типі `newbookType` тільки автора і назва книги з типу `bookType` :

```
<xsd:complexType name="newBookType">
<xsd:complexContent>
      <xsd:restriction base="bookType">
      <xsd:sequence>
         <xsd:element name="author" type="xsd:normalizedString"
                     minOccurs="0" />
         <xsd:element name="title" type="xsd:normalizedString" />
      </xsd:sequence>
  </xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
```

Цей опис виглядає дивним. Чому треба заново описувати всі елементи, залишаю- щіся після звуження? Не простіше чи визначити новий тип?

Справа в тому, що в мову XSD внесені елементи об'єктно-орієнтованого програмного забезпечення. вання, яких ми не стосуватимемося. Розширений і звужений типи пов'язані з своїм базовим типом ставленням успадкування, і до них можна застосувати операцію підстановки. У всіх типів мови XSD є загальний предок - базовий тип `anyType` . Від

йому успадковуються усі складні типи. Це подібно до того, як у всіх класів Java є загальний предок - клас `Object` , а всі масиви успадковуються від нього. Від базового типу `anyType` успадковується і тип `anySimpleType` - загальний предок всіх простих типів.

Отже, складні типи визначаються як звуження типу `anyType` . Якщо суворо підходити до визначенню складного типу, то визначення типу `bookType` , зроблене на початку попереднього розділу, треба записати так:

```
<xsd:complexType name="bookType">

<xsd:complexContent>

  <xsd:restriction base="xsd:anyType">

        <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="author" type="xsd:normalizedString"
                     minOccurs="0" />
        <xsd:element name="title" type="xsd:normalizedString" />
        <xsd:element name="pages" type="xsd:positiveInteger"
                     minOccurs="0" />
        <xsd:element name="publisher" type="xsd:normalizedString"
                     minOccurs="0" />
      </xsd:sequence>

   </xsd:restriction>

</xsd:complexContent>

</xsd:complexType>
```

Рекомендація мови XSD дозволяє скоротити цей запис, що ми і зробили в попередні ному розділі. Це подібно до того, як у Java ми опускаємо слова " `extends Object` " в заголівці описи класу.

Закінчимо на цьому опис мови XSD і перейдемо до прикладів.

приклад: схема адресною книги

У лістингу 28.4 записана схема документа, наведеного у лістингу 28.2.

Листинг 28.4. Схема документа XML

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd=" http://www.w3.org/2001/XMLSchema "
            xmlns:ntb=" http://some.firm.com/2011/ntbNames
            "
       targetNamespace=" http://some.firm.com/2011/ntbNames ">

<xsd:element name="notebook" type="ntb:notebookType" />

   <xsd:complexType name="notebookType">
       <xsd:element name="person" type="ntb:personType"
                    minOccurs="0" maxOccurs="unbounded" />
</xsd:complexType>
```

```
  <xsd:complexType name="personType">
  <xsd:sequence>

      <xsd:element name="name">
      <xsd:complexType>
        <xsd:attribute name="first"   type="xsd:string" use="optional" />
        <xsd:attribute name="second"  type="xsd:string" use="optional" />
        <xsd:attribute name="surname" type="xsd:string" use="required" />
      </xsd:complexType>
  </xsd:element>

  <xsd:element name="birthday" type="ntb:ruDate" minOccurs="0" />

     <xsd:element name="address" type="ntb:addressType"
              minOccurs="0" maxOccurs="unbounded" />

     <xsd:element name="phone-list" type="ntb:phone-listType"
              minOccurs="0" />

   </xsd:sequence>
</xsd:complexType>

  <xsd:complexType name="addressType" >
  <xsd:sequence>
     <xsd:element name="street" type="xsd:string" />
     <xsd:element name="city"   type="ntb:cityType" />
     <xsd:element name="zip"    type="xsd:positiveInteger" />
  </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name='cityType'>
    <xsd:simpleContent>
      <xsd:extension base='xsd:string' >
       <xsd:attribute name='type' type='ntb:placeType' default='місто' />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:simpleType name="placeType">
    <xsd:restriction base = "xsd:string">
       <xsd:enumeration value="місто" />
        <xsd:enumeration value="селище" />
          <xsd:enumeration value="село" />
    </xsd:restriction>
</xsd:simpleType>

  <xsd:complexType name="phone-listType">
    <xsd:element name="work-phone" type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="home-phone" type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded" />
  </xsd:complexType>
```

```
  <xsd:simpleType name="ruDate">
    <xsd:restriction base="xsd:string">
       <xsd:pattern value="[0-9]{2}.[0-9]{2}.[0-9]{4}" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Лістинг 28.4, як звичайний документ XML, починається з прологу, що показує вер- цю XML і визначає стандартний простір імен схеми XML з ідентифікації тором `http://www.w3.org/2001/XMLSchema` . Цьому ідентифікатору дано префікс `xsd` . Кінеч- але, префікс може бути іншим, часто пишуть префікс `xs` .

`targetNamespace`, що ще не зустрічався нам, визначає ідентифікатор про- подорожі імен, в яке потраплять імена типів, що визначаються в цьому документі, еле- ментів та атрибутів, так зване *цільове простір імен* (target namespace). Цей ідентифікатор відразу ж зв'язується з префіксом `ntb` , який відразу використовує ється для уточнення тільки що певних імен в посилання на них.

Всі описи схеми нашої адресної книжки укладено в одному третьому рядку, в ко- торой зазначено, що адресна книга складається з одного елемента з ім'ям `notebook` , має складний тип `notebookType` . Цей елемент повинен з'явитися в документі рів- але один раз. Залишок лістингу 28.4 присвячений опису типу цього елемента та інших типів.

Опис складного типу `notebookType` нескладно (вибачте за каламбур). Воно займає три рядки лістингу, не рахуючи відкриває і закриває тега, і просто каже про тому, що даний тип складають кілька елементів `person` типу `personType` .

Опис типу `personType` небагатьом складніше. Воно каже, що цей тип складають чотири елемента: `name` , `birthday` , `address` і `phone-list` . Для елемента `name` відразу ж вказано необов'язкові атрибути `first` та `second` простого типу `string` _ певного в про- подорожі імен `xsd` . Тип обов'язкового атрибуту `surname` - теж `string` .

Далі в лістингу 28.4 визначаються типи, що залишилися: `addressType` , `phone-listType` і `ruDate` . Необхідність визначення простого типу `ruDate` виникає тому, що встро- енний у схему XML тип `date` наказує задавати дату у вигляді 2004-10-22, а в Росії прийнято формат 22.10.2004. Тип `ruDate` визначається як *звуження* (Restriction) типу `string` за допомогою шаблону. *Шаблон* (pattern) для запису дати у вигляді `ДД.ММ.РРРР` задається регулярним виразом.

Безіменні типи

Усі описані у лістингу 28.4 типи використовуються лише один раз. Тому необов'яз- тельно давати типу ім'я. Схема XML, як говорилося раніше, дозволяє визначати без- зимові типи. Таке визначення дається всередині описи елемент. Саме так в лістингу 28.4 описані атрибути елемента `name` . У лістингу 28.5 показано спрощене опис схеми адресної книги.

Листинг 28.5. Схема документа XML с безымянными типами

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd=' http://www.w3.org/2001/XMLSchema' targetNamespace='
            http://some.firm.com/2011/ntbNames '>
```

```
<xsd:element name='notebook'>
  <xsd:complexType>
 <xsd:sequence>

      <xsd:element name='person' maxOccurs='unbounded'>
        <xsd:complexType>
      <xsd:sequence>

            <xsd:element name='name'>
               <xsd:complexType>
    <xsd:attribute name='first'   type='xsd:string' use='optional' />
    <xsd:attribute name='second'  type='xsd:string' use='optional' />
    <xsd:attribute name='surname' type='xsd:string' use='required' />
               </xsd:complexType>
       </xsd:element>

            <xsd:element name='birthday'>
               <xsd:simpleType>
     <xsd:restriction base='xsd:string'>
        <xsd:pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
     </xsd:restriction>
               </xsd:simpleType>
         </xsd:element>

            <xsd:element name='address' maxOccurs='unbounded'>
               <xsd:complexType>
  <xsd:sequence>
     <xsd:element name='street' type='xsd:string' />
     <xsd:element name='city'>
       <xsd:complexType>
      <xsd:simpleContent>

           <xsd:extension base='xsd:string'>
              <xsd:attribute name='type' type='xsd:string'
                             use='optional' default='gorod' />
        </xsd:extension>

        </xsd:simpleContent>
      </xsd:complexType>
   </xsd:element>

     <xsd:element name='zip' type='xsd:positiveInteger' />
  </xsd:sequence>
               </xsd:complexType>
        </xsd:element>

            <xsd:element name='phone-list'>
              <xsd:complexType>
  <xsd:sequence>
    <xsd:element name='work-phone' type='xsd:string'
                 minOccurs='0' maxOccurs='unbounded' />
```

```
      <xsd:element name='home-phone' type='xsd:string'
                   minOccurs='0' maxOccurs='unbounded' />
    </xsd:sequence>
                 </xsd:complexType>
           </xsd:element>

            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

        </xsd:sequence>
  </xsd:complexType>

</xsd:element>

</xsd:schema>
```

Ще одне спрощення можна зробити, використовуючи простір стандартних імен. Подивимося, які простори імен застосовуються в схемах XML.

Простору імен мови XSD

Імена елементів і атрибутів, що використовуються при записі схем, визначені в просторі. ствi імен з ідентифікатором `http://www.w3.org/2001/XMLSchema` . Префікс імен, відносячи- тих до цього простору, часто називають `xs` або `xsd` , як у лістингах 28.4 і 28.5. Кожен аналізатор "знає" цей простір імен і "розуміє" імена з цього про- подорожі.

Можна зробити цей простір імен простором за промовчанням, але тоді треба обов'язково визначити префікс ідентифікатора цільового простору імен для оп-виділених в схемою типів та елементів.

У лістингу 28.6 для спрощення запису стандартне простір імен схеми XML з ідентифікатором `http://www.w3.org/2001/XMLSchema` зроблено простором імен по замовчуванням. Імена, що належать до цільового простору імен, мають префікс. `ntb` , щоб вони не потрапили в простір імен по замовчуванням.

Листинг 28.6. Схема документа XML с целевым пространством имен

```
<?xml version='1.0'?>
<schema xmlns=' http://www.w3.org/2001/XMLSchema'
   targetNamespace='
   http://some.firm.com/2011/ntbNames'xmlns:ntb='
   http://some.firm.com/2011/ntbNames' >

<element name='notebook'>

    <complexType>
    <sequence>

          <element name='person' maxOccurs='unbounded'>
             <complexType>
          <sequence>
```

```
            <element name='name'>
          <complexType>
             <attribute name='first'   type='string' use='optional' />
             <attribute name='second'  type='string' use='optional' />
             <attribute name='surname' type='string' use='required' />
           </complexType>
           </element>

            <element name='birthday'>
           <simpleType>
             <restriction base='string'>
               <pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
             </restriction>
           </simpleType>
           </element>

            <element name='address' maxOccurs='unbounded'>
          <complexType>
             <sequence>
               <element name='street' type='string' />
               <element name='city'   type='string' />
               <element name='zip'    type='positiveInteger' />
             </sequence>
          </complexType>
           </element>

            <element name='phone-list'>
           <complexType>
             <sequence>
               <element name='work-phone' type='string'
                        minOccurs='0' maxOccurs='unbounded'/>
               <element name='home-phone' type='string'
                        minOccurs='0' maxOccurs='unbounded'/>
             </sequence>
           </complexType>
           </element>

       </sequence>
     </complexType>
   </element>

         </sequence>
 </complexType>

</element>

</schema>
```

Оскільки в лістингу 28.6 простором імен по замовчуванням зроблено простір `http://www.w3.org/2001/XMLSchema` , префікс `xsd` не потрібний.

Слід помітити, що в цільове простір імен потрапляють тільки *глобальні має-на* , чиї описи безпосередньо вкладені в елемент `<Schema>` . Це природно, пото-

му що лише глобальними іменами можна скористатися далі в цій чи іншій схемою. У лістингу 28.6 лише одне глобальне ім'я - `<notebook>` . Вкладені імена `name` , `address` і інші всього-навсього *асоційовані* з глобальними іменами.

У схемах та документах XML часто застосовується ще один стандартний простір імен. Рекомендація мови XSD визначає кілька атрибутів: `type` , `nil` , `schemaLocation` , `noNamespaceSchemaLocation` , які застосовуються не тільки в схемах, але і безпосередньо в описуваних цими схемами документах XML, званих *екземплярами схем* (XML schema instance). Імена цих атрибутів відносяться до простору імен `http://www.w3.org/2001/XMLSchema-instance` . Даному простору імен найчастіше приписують префікс `xsi` , наприклад:

```
<xsd:schema xmlns:xsd=" http://www.w3.org/2001/XMLSchema "
            xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance
            ">
```

Увімкнення файлів схеми в іншу схему

До створюваної схеми можна включити файли, що містять інші схеми. Для цього є два елемента схеми: `<include>` і `<import>` . Наприклад:

```
<xsd:include xsi:schemaLocation="names.xsd" />
```

Файл, що вмикається, задається атрибутом `xsi:schemaLocation` . У прикладі він використаний для того, щоб включити до створюваної схеми вміст файлу names.xsd. Файл повинен містити схему з описами та визначеннями з того ж простору імен, що і в схемі, що створюється, або без простору імен, тобто в ньому не використаний атри- бут `targetNamespace` . Це зручно, якщо ми хочемо додати до створюваної схеми визначення схеми names.xsd або просто хочемо розбити велику схему на два файли. Можна, можливо уявити результат включення так, ніби вміст файлу names.xsd про- сто записано на місці елемента `<include>` .

Перед увімкненням файлу можна змінити деякі визначення, наведені в ньому. Для цього використовується елемент `<redefine>` , наприклад:

```
<xsd:redefine schemaLocation="names.xsd">

  <xsd:simpleType name="nameType">
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="40"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:redefine>
```

Якщо ж файл, що включається, містить імена з іншого простору імен, то треба вос- скористатися елементом схеми `<import>` . Наприклад, нехай файл A.xsd починається зі наступних визначень:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd=" http://www.w3.org/2001/XMLSchema "targetNamespace="
  http://some.firm.com/someNames ">
```

а файл B.xsd починається з визначень

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd=" http://www.w3.org/2001/XMLSchema "targetNamespace="
  http://some.firm.com/anotherNames ">
```

Ми вирішили включити ці файли в новий файл C.xsd. Це робиться так:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd=" http://www.w3.org/2001/XMLSchema "
    targetNamespace=" http://some.firm.com/yetAnotherNames
    "xmlns:pr1=" http://some.firm.com/someNames "
    xmlns:pr2=" http://some.firm.com/anotherNames ">

<xsd:import namespace=" http://some.firm.com/someNames "
    xsi:schemaLocation="A.xsd" />

<xsd:import namespace=" http://some.firm.com/anotherNames "
    xsi:schemaLocation="B.xsd" />
```

Після цього в файлі C.xsd можна, можливо використовувати імена, певні в файлах A.xsd іB.xsd, постачаючи їх префіксами `pr1` і `pr2` відповідно.

Елементи `<include>` і `<import>` слід розташовувати перед усіма визначеннями схеми.

Значення атрибуту `xsi:schemaLocation` - рядок URI, тому файл з включається схе-мій може розташовуватися в будь-кому місці Інтернет.

Зв'язок документа XML зі своєю схемою

Програмі-аналізатору, що перевіряє відповідність документа XML його схемою, треба якось вказати файли (один чи кілька), які містять схему документа. Це можна зробити у різний спосіб. По-перше, можна подати необхідні файли на вхід аналізатора. Так робить, наприклад, перевіряючий аналізатор XSV (XML Schema Validator) - **ftp://ftp.cogsci.ed.ac.uk/pub/XSV/** :

```
$ xsv ntb.xml ntb1.xsd ntb2.xsd
```

По-друге, можна задати файли зі схемою як властивість аналізатора, що встановлюється методом `setProperty()` або значення змінної оточення аналізатора. Так робить, наприклад, перевіряючий аналізатор Xerces.

Ці способи зручні, коли документ у різних випадках потрібно пов'язати з різними схемами. Якщо ж схема документа фіксована, то її зручніше вказати прямо в документе XML. Це робиться одним з двох способів.

- Якщо елементи документа не належать ніякому простору імен і запису- ні без префікса, то в кореневому елементі документа записується атрибут `noNamespaceSchemaLocation` , вказівник розташування файлу зі схемою у фор- ме URI:

```
<notebook xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance "
    xsi:noNamespaceSchemaLocation="ntb.xsd">
```

І тут у схемі має бути цільового простору імен, т. е. не слід використовувати атрибут `targetNamespace` .

- Якщо ж елементи документа відносяться до деякого простору імен, то прикладняется атрибут `schemaLocation` , в якому через прогалину парами перераховуються про- подорожі імен та розташування файлу зі схемою, що описує цей простір імен. Продовжуючи приклад попереднього розділу, можна написати:

```
<notebook xmlns=" http://some.firm.com/2003/ntbNames "
    xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance
    "xsi:schemaLocation=
       http://some.firm.com/someNames A.xsd _
        http://some.firm.com/anotherNames B.xsd"
    xmlns:pr1=" http://some.firm.com/someNames
    " xmlns:pr2="
    http://some.firm.com/anotherNames ">
```

Після цього в документі можна використовувати імена, визначені у схемах A.xsd та B.xsd, постачаючи їх префіксами `pr1` і `pr2` відповідно.

Інші мови описи схем

Навіть із наведеного раніше короткого опису мови XSD видно, що він вийшов дуже складним та заплутаним. Є вже кілька книг, повністю присвячених цьому мови. Їх обсяг анітрохи не менше обсягу цією книги.

Існують інші, більш прості мови опису схеми документа XML. Най- більше поширення отримали такі мови:

- Schematron - **http://www.ascc.net/xml/resource/schematron/** ;
- RELAX NG (Regular Language Description for XML, New Generation), ця мова нік як злиття мов Relax і TREX - **http://www.oasis-open.org/committees/ relax-ng/** ;
- Relax - **http://www.xml.gr.jp/relax/** ;
- TREX (Tree Regular Expressions for XML) - **http://www.thaiopensource.com/trex/** ;
- DDML (Document Definition Markup Language), відомий ще як XSchema - **http://www.w3.org/TR/NOTE-ddml** .

Менш поширені мови DCD (Document Content Description), SOX (One's Schemafor Object-Oriented XML), XDR (XML-Data Reduced).

Усі перелічені мови дозволяють більше або менше повно описувати схему документ-та. Можливо, вони витіснять мова XSD, можливо, будуть існувати спільно.

Інструкції по обробці

Згадаємо ще одну конструкцію мови XML – *інструкції з обробки* (processing) інструкції). Вона дозволяє передати аналізатору чи іншій програмі-обробнику документа додаткові відомості для обробки. Інструкція з обробки вигляду- дит так:

```
<? відомості для аналізатора ?>
```

Перша частина прологу документа XML - перший рядок XML-файлу - це саме інструкція з обробки. Вона передає аналізатору документа версію мови XML і кодірування символів, якими записаний документ.

Першу частину роботи закінчено. Документи XML та їх схеми написані. Тепер треба подумати про те, як вони відображатимуться на екрані дисплея, на аркуші паперу, на екрані мобільного телефону, тобто потрібно подумати про візуалізацію документа XML.

Насамперед всього, документ слід розібрати, проаналізувати (parse) його структуру.

Аналіз документа XML

на першим етапі розбору проводиться *лексичний аналіз* (lexical parsing) документа XML. Документ розбивається на окремі *неподільні елементи* (tokens), якими є теги, службові слова, роздільники, текстові константи. Проводиться перевірка напів- чених елементів та їх зв'язків між собою. Лексичний аналіз виконують спеціаль- ні програми - *сканери* (Scanners). Найпростіші сканери - це класи `java.util.StringTokenizer` та `java.io.StreamTokenizer` із стандартної поставки Java SE JDK, які ми розглядали в попередніх розділах.

Потім здійснюється *граматичний аналіз* (grammar parsing). При цьому аналізую- ється логічна структура документа, складаються вирази, вирази об'єднується. ються в блоки, блоки - в модулі, якими можуть бути абзаци, параграфи, пункти, глави. Граматичний аналіз проводять програми-аналізатори, так звані *пар- сірки* (parsers).

Створення сканерів і парсерів - улюблена розвага програмістів. За недовгу історію XML написані десятки, а то й сотні XML-парсерів. Багато з них написа- ні мовою Java. Усі парсери можна розділити на три групи.

У першу групу входять парсери, які проводять аналіз, ґрунтуючись на структурі деревини. ва, що відбиває вкладеність елементів документа (tree-based parsing). Дерево доку- мента будується в оперативній пам'яті перед переглядом. Такі парсери простіше реалі- зувати, але створення дерева вимагає великого обсягу оперативної пам'яті, адже розмір документів XML не обмежено. Необхідність частого перегляду вузлів дерева сповільнюється. ляє роботу парсеру.

У другу групу входять парсери, провідні аналіз, грунтуючись на подіях (event-based parsing). Ці парсери переглядають документ один раз, відзначаючи події перегляду. Подією вважається поява чергового елемента XML: відкриває або тега, що закриває, тексту, що міститься в тілі елемента. У разі виникнення події викликається відповідний метод його обробки: `startElement()` , `endElement()` , `characters()` і т. д. Такі парсери складніші в реалізації, зате вони не стро- ють дерево в оперативній пам'яті і можуть аналізувати не весь документ, а його окремо. ні елементи разом із вкладеними в них елементами. Фактичним стандартом тут став вільно поширюваний набір класів та інтерфейсів SAX (Simple API for XML), створений Давидом Меггінсон (David Megginson). Основний сайт даного проекту - **http://www.saxproject.org/** . Зараз застосовується друге покоління цього набору, зване SAX2. Набір SAX2 входить во багато парсери, наприклад Xerces2.

Третю групу утворюють потокові парсери (stream parsers), які так само, як і парсери другої групи переглядають документ, переходячи від елемента до елемента. При кожному переході парсер надає програмі методи `getName()` , `getText()` , `getAttributeName()` , `getAttributeValue()` і т. д., що дозволяють обробити поточний елемент.

У стандартне постачання Java і Standard Edition і Enterprise Edition входить набір інтерфейсів і класів для створення парсерів і перетворення документів XML, JAXP (Java API for XML Processing). За допомогою однієї з частин цього набору, званої DOM API (Document Object Model API), можна створювати парсери першого тип формує дерево об'єктів. За допомогою другої частини набору JAXP, називає- мій SAX API, можна створювати SAX-парсери. Третя частина JAXP, призначена для створення потокових парсерів, називається StAX (Streaming API для XML).

Аналіз документів XMLз допомогою SAX2

Інтерфейси та класи SAX2 зібрані в пакети `org.xml.sax` , `org.xml.sax.ext` ,`org.xml.sax.helpers` , `javax.xml.parsers` . Розглянемо їх Детальніше.

Основу SAX2 складає інтерфейс `org.xml.sax.ContentHandler` , що описує методи обробки подій: початку документа, появи тега, поява тіла елемента, поява закриває тега, закінчення документа. При виникненні такої події SAX2 звертається до методу-оброблювача події, передаючи йому аргу- менти, що містять інформацію про подію. Справа розробника реалізувати ці методи, забезпечивши правильний аналіз документа.

У початку обробки документа викликається метод

```
public void startDocument();
```

У ньому можна, можливо поставити початкові дії по обробці документа.

При появі символу " < ", початківця відкриває тег, викликається метод

```
public void startElement(String uri, String name,
                         String qname, Attributes attrs);
```

У метод передаються три імені, два з яких пов'язані з простором імен: іден- тифікатор простору імен `uri` , локальне ім'я тега без префікса `name` та розширене ім'я з префіксом `qname` , а також атрибути елемента `attrs` , що відкриває тега , якщо вони є. Якщо простір імен не визначено, то значення першого та другого аргументів рівні `null` . Якщо ні атрибутів, то передається посилання на порожній об'єкт `attrs` .

При появі символів " </ ", початківців закриває тег, викликається метод

```
public void endElement(String uri, String name, String qname);
```

При появі рядки символів викликається метод

```
public void characters(char[] ch, int start, int length);
```

У нього передається масив символів `ch` , індекс початку рядка символів `start` у цьому масиві і кількість символів `length` .

При появі у тексті документа інструкції по обробці викликається метод

```
public void processingInstruction(String target, String data);
```

У метод передається ім'я програми-оброблювача `target` і додаткові відомості `data` .

При появі пробельних символів, які мають бути пропущені, викликається метод

```
public void ignorableWhitespace(char[] ch, int start, int length);
```

У нього передається масив `ch` пробілових символів, що йдуть поспіль , індекс початку сім- волів в масиві `start` і кількість символів `length` .

Інтерфейс `org.xml.sax.ContentHandler` реалізований класом `org.xml.sax.helpers.DefaultHandler` . У ньому зроблено пуста реалізація всіх методів. Розробнику залишається реалізувати тільки ті методи, які йому потрібні.

Застосуємо методи SAX2 для обробки нашої адресною книги. Запис документа на мовою XML зручна для виявлення структури документа, але незручна для роботи з документом в об'єктно-орієнтованій середовище. Тому найчастіше вміст документ XML представляється у вигляді одного або декількох об'єктів, званих *об'єктами даних* JDO (Java Data Objects). Ця операція називається *зв'язуванням даних* (data binding) з об'єктами JDO.

Зв'яжемо вміст нашої адресною книжки з об'єктами Java. Для цього спочатку опишемо класи Java (листинги 28.7 та 28.8), які представлять вміст адресної книги.

Листинг 28.7. Класс, описывающий адрес

```
public class Address {

  private String street, city, zip, type = "Місто";

  public Address(){}

  public String getStreet(){ return street; }
  public void setStreet(String street) { this.street = street; }

  public String getCity(){ return city; }
  public void setCity(String city) { this.city = city; }

  public String getZip(){ return zip; }
  public void setZip(String zip) { this.zip = zip; }

  public String getType(){ return type; }
  public void setType(String type) { this.type = type; }

  public String toString(){
    return "Address: " + street + " " +  city + " " + zip;
  }
}
```

Листинг 28.8. Класс, описывающий запись адресной книжки

```
public class Person{

  private String firstName, secondName, surname, birthday;
  private Vector<Address> address;
  private Vector<Integer> workPhone;
  private Vector<Integer> homePhone;

  public Person(){}

  public Person(String firstName, String secondName, String surname) {
    this.firstName = firstName;
    this.secondName = secondName;
    this.surname = surname;
  }

  public String getFirstName(){ return firstName; }
  public void setFirstName(String firstName) {
     this.firstName = firstName;
  }

  public String getSecondName(){ return secondName; }
  public void setSecondName(String secondName) {
     this.secondName = secondName;
  }

  public String getSurname(){ return surname; }
  public void setSurname(String surname) {
      this.surname = surname;
  }

  public String getBirthday(){ return birthday; }
  public void setBirthday(String birthday) {
      this.birthday = birthday;
  }

  public void addAddress(Address addr) {
    if (address == null) address = new Vector();
    address.add(addr);
  }
  public Vector<Address> getAddress(){ return address; }

  public void removeAddress(Address addr) {
    if (address != null) address.remove(addr);
  }

  public void addWorkPhone(String phone) {
    if (workPhone == null) workPhone = new Vector();
    workPhone.add(new Integer(phone));
  }
  public Vector<Integer> getWorkPhone(){ return workPhone; }
```

```
  public void removeWorkPhone(String phone) {
    if (workPhone != null) workPhone.remove(new Integer(phone));
  }

  public void addHomePhone(String phone) {
    if (homePhone == null) homePhone = new Vector();
    homePhone.add(new Integer(phone));
  }
  public Vector<Integer> getHomePhone(){ return homePhone; }

  public void removeHomePhone(String phone) {
    if (homePhone != null) homePhone.remove(new Integer(phone));
  }

  public String toString(){
    return "Person: " + surname;
  }
}
```

Після визначення класів Java, до екземплярів яких буде занесено вміст адресною книжки, напишемо програму, читаючу адресну книжку і зв'язуючуїї з об'єктами Java.

У лістингу 28.9 наведено приклад класу-обробника `NotebookHandler` для адресної книжки, описаної у лістингу 28.2. Методи класу `NotebookHandler` аналізують вміст жиму адресної книжки та поміщають його у вектор, складений з об'єктів класу `Person` , описаного в лістингу 28.8.

Листинг 28.9. Класс-обработчик документа XML средствами SAX2

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;
import java.util.*;
import java.io.*;

public class NotebookHandler extends DefaultHandler{

   static final String JAXP_SCHEMA_LANGUAGE =
     " http://java.sun.com/xml/jaxp/properties/schemaLanguage ";

   static final String W3C_XML_SCHEMA =
     " http://www.w3.org/2001/XMLSchema
     ";

   private Person person;
   private Address address;
   private static Vector<Person> pers = new Vector<>();
   boolean inBirthday, inStreet, inCity, inZip,
           inWorkPhone, вHomePhone;

  public void startElement(String uri, String name,
                           String qname, Attributes attrs)
```

```
        throws SAXException{
  switch (qname) {
    case "name":
        person = new Person(attrs.getValue("first"),
           attrs.getValue("second"), attrs.getValue("surname"));
     break;
    case "birthday":
        inBirthday = true; break;
    case "address":
       address = new Address(); break;
    case "street":
       inStreet = true; break;
  case "City":
       inCity = true;
       if (attrs != null) address.setType(attrs.getValue("type"));
     break;
  case "zip":
       inZip = true; break;
  case "work-phone":
       inWorkPhone = true; break;
  case "home-phone":
     inHomePhone = true;
}

public void characters(char[] buf, int offset, int len)
       throws SAXException{
  String s = new String(buf, offset, len);

  if (inBirthday) {
     person.setBirthday(s);
     inBirthday = false;
  }else if (inStreet){
     address.setStreet(s);
     inStreet = false;
  }else if (inCity){
     address.setCity(s);
     inCity = false;
  }else if (inZip){
     address.setZip(s);
     inZip = false;
  }else if (inWorkPhone){
     person.addWorkPhone(s);
     inWorkPhone = false;
  }else if (inHomePhone){
     person.addHomePhone(s);
     inHomePhone = false;
  }
}

public void endElement(String uri, String name, String qname)
       throws SAXException{
```

```
    if (qname.equals("address")){
        person.addAddress(address);
        address = null;
    }else if (qname.equals("person")){
        pers.add(person);
        person = null;
    }
  }

  public static void main(String[] args) {
    if (args.length < 1) {
       System.err.println("Usage: java NotebookHandler ntb.xml");
       System.exit(1);
    }
    try{
      NotebookHandler handler = New NotebookHandler();

      SAXParserFactory fact = SAXParserFactory.newInstance();

          fact.setNamespaceAware(true);
          fact.setValidating(true);

      SAXParser saxParser = fact.newSAXParser();

      saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);

      File f = new File(args[0]);

      saxParser.parse(f, handler);

          for (int k = 0; k < pers.size(); k++)
        System.out.println((pers.get(k)).getSurname());

}catch(SAXNotRecognizedException x) {

         System.err.println("Невідоме властивість: " +
                            JAXP_SCHEMA_LANGUAGE);
      System.exit(1);

} catch (Exception ex) {

      System.err.println(ex);
    }
  }

  public void warning(SAXParseException ex){
    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }

  public void error(SAXParseException ex){
    System.err.println("Error: " + ex);
```

```
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }

  public void fatalError(SAXParseException ex){
    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }
}
```

Після того, як клас-обробник написано, проаналізувати документ дуже легко. Стандартні дії наведено в методі `main()` програми лістингу 28.9.

Оскільки реалізація парсера залежить від його програмного оточення, SAX- парсер - об'єкт класу `SAXParser` - створюється не конструктором, а фабричним мето- будинок `newSAXParser()` .

Об'єкт-фабрика, своєю чергою, формується методом `newInstance()` . Далі можна методом

```
void setFeature(String name, boolean value);
```

встановити властивості парсерів, створюваних цією фабрикою. Наприклад, після

```
fact.setFeature(" http://xml.org/sax/features/namespace-prefixes ", true);
```

парсери, створювані фабрикою `fact` , враховуватимуть префікси імен тегів та атрибутів.

Список таких властивостей можна переглянути в документації Java API в описі пакета `org.xml.sax` або на сайті проекту SAX **http://www.saxproject.org/** . Слід враховувати, що не всі парсери повністю виконують ці властивості.

Якщо до об'єкту-фабриці застосувати метод

```
void setValidating(true);
```

як це зроблено в лістингу 28.9, то вона буде виробляти парсери, перевіряючі структуру документа. Якщо застосувати метод

```
void setNamespaceAware(true);
```

то об'єкт-фабрика буде виробляти парсери, враховують простору імен.

Після того як об'єкт-парсер створено, залишається тільки застосувати метод `parse()` , передавшийому ім'я аналізованого файлу та екземпляр класу-обробника подій.

У класі `javax.xml.parsers.SAXParser` є десяток методів `parse()` . Крім методу `parse(File, DefaultHandler)` , використаного в лістингу 28.9, існують ще методи, що дозволяють витягти документ із вхідного потоку класу `InputStream` , об'єкта класу `InputSource` , адреси URI або з спеціально створеного джерела класу `InputSource` .

Методом `setProperty()` можна встановити різні властивості парсера. У лістингу 28.9 цей метод використаний для того, щоб парсер перевіряв правильність документа з поміщю схеми XSD. Якщо парсер виконує перевірки, тобто. застосований метод `setValidating(true)` , має сенс зробити розгорнуті повідомлення про помилки. Це передбачено інтерфейсом `ErrorHandler` . Він розрізняє попередження, помилки і

фатальні помилки і описує три методи, які автоматично виконуються при появі помилки відповідного виду:

```
public void warning(SAXParserException ex);
public void error(SAXParserException ex);
public void fatalError(SAXParserException ex);
```

Клас `DefaultHandler` робить порожню реалізацію цього інтерфейсу. При розширенні даного класу можна, можливо зробити реалізацію одного або всіх методів інтерфейсу `ErrorHandler` . приклад такий реалізації наведено в лістингу 28.9. Клас `SAXParserException` зберігає номер рядка і стовпця документа, що перевіряється, в якому помічено помилку. Їх можна отримати методами `getLineNumber()` та `getColumnNumber()` , як зроблено в лістингу 28.9.

Аналіз документів XML з допомогою StAX

Інтерфейси та класи StAX зібрані в пакети `javax.xml.stream` , `javax.xml.stream.events` , `javax.xml.stream.util` , але на практиці достатньо застосування тільки кількох інтерфейсів і класів.

Основні методи розбору документа описані в інтерфейсі `XMLStreamReader` . Розбір полягає в тому, що парсер переглядає документ, переходячи від елемента до елемента і від однієї частини елемента до іншої методом `next()` . Метод `next()` повертає одну з цілих констант, описаних в інтерфейсі `XMLStreamConstants` , що показують тип частини документа, в якій знаходиться парсер: `START_DOCUMENT` , `START_ELEMENT` , `END_ELEMENT` , `CHARACTERS` і т.д. Залежно від цього інтерфейс `XMLStreamReader` надає ті чи інші методи опрацювання документа. Так, якщо парсер знаходиться в відкриває тегу елемента, `START_ELEMENT` , то ми можемо отримати коротке ім'я еле- мента методом `getLocalName()` , число атрибутів методом `getAttributeCount()` , ім'я і зна- чення `k` -го атрибута методами `getAttributeName(k)` та `getAttributeValue(k)` . Якщо парсер знаходиться в тілі елемента, `CHARACTERS` , можна отримати вміст тіла методом `getText()` .

Програма наступного лістингу 28.10 виконує засобами StAX ту ж роботу, що й програма, записана в лістингу 28.9.

Листинг 28.10. Класс-обработчик документа XML средствами StAX

```
import javax.xml.stream.*;
import  java.util.*;
import java.io.*;

public class NotebookHandlerStAX implements XMLStreamConstants{
 private Person person;
 private Address address;
 private static Vector<Person> pers = new Vector<>();
 boolean inBirthday, inStreet, inCity, inZip,
     inWorkPhone, вHomePhone;
```

```
 private void processElement(XMLStreamReader element)
     throws XMLStreamException{
  switch (element.getLocalName()){
   case "name": person = new Person(element.getAttributeValue(0),
                element.getAttributeValue(1),
                element.getAttributeValue(2)); break;
   case "birthday": inBirthday = true; break;
   case "address": address = new Address(); break;
   case "street": inStreet = true; break;
   case "City": inCity = true; break;
   case "zip": inZip = true; break;
   case "work-phone": inWorkPhone = true; break;
   case "home-phone": inHomePhone = true; break;
  }
}
private void processText(String text) {
 if (inBirthday) {
   person.setBirthday(text);
   inBirthday = false;
 }else if (inStreet){
   address.setStreet(text);
   inStreet = false;
 }else if (inCity){
   address.setCity(text);
   inCity = false;
 }else if (inZip){
   address.setZip(text);
   inZip = false;
 }else if (inWorkPhone){
   person.addWorkPhone(text);
   inWorkPhone = false;
 }else if (inHomePhone){
   person.addHomePhone(text);
   inHomePhone = false;
 }
}
private void finishElement(String name) {
 switch (name) {
  case "address": person.addAddress(address);
   address = null; break;
  case "person": pers.add(person);
   person = null; break;
 }
}

public static void main(String[] args)
 {if (args.length < 1) {
```

```
   System.err.println("Usage: java NotebookHandlerStAX ntb.xml");
   System.exit(1);
 }
 NotebookHandlerStAX handler = new NotebookHandlerStAX();
 try{
  FileInputStream inStream = new FileInputStream(args[0]);
  XMLStreamReader xmlReader =
  XMLInputFactory.newInstance().
     createXMLStreamReader(inStream);
  int event;
  while (xmlReader.hasNext()) {
    event = xmlReader.next();
    switch (event) {
   case START_ELEMENT: handler.processElement(xmlReader); break;
   case CHARACTERS: handler.processText(xmlReader.getText()); break;
   case END_ELEMENT: handler.finishElement(xmlReader.getLocalName());
    }
  }
  xmlReader.close();
 } catch (Exception ex) {
  ex.printStackTrace();
 }
}
}
```

Зв'язування даних XML з об'єктами Java

У наведеному прикладі ми самі створили класи `Address` і `Person` , які представляють до-документ XML. Оскільки структура документа XML чітко визначена, можна розробити. ти стандартні правила зв'язування даних XML з об'єктами Java і створити про- грамні засоби для їх реалізації.

Корпорація Sun Microsystems розробила пакет інтерфейсів та класів JAXB (Java Architecture for XML Binding), що полегшують зв'язування даних. Він входить у стандартну постачання Java SE, а також може бути скопійовано з сайту **http:// jaxb.dev.java.net/** . Для роботи з пакетом JAXB аналізований документ XML обя-тельно має бути забезпечений описом мовою XSD.

До складу пакету JAXB входить компілятор `xjc` (XML-Java Compiler). Він переглядає схему XSD і будує по ньому об'єкти Java в оперативній пам'яті, а також створює ви-хідні файли об'єктів Java. Наприклад, після виконання команди

```
$ xjc -roots notebook ntb.xsd -d sources
```

в якій ntb.xsd — файл листингу 28.4 — у каталозі sources (за замовчуванням у поточному каталозі) будуть створені файли Address.java, Name.java, Notebook.java, Person.java, PhoneList.java з описами об'єктів Java.

Прапор `-roots` показує один або кілька кореневих елементів, розділених комами.

Створені компілятором `xjc` вихідні файли звичайним чином, за допомогою компілятора `javac` , компілюються у класи Java.

Отримавши об'єкти даних, можна перенести в них вміст документа XML мето- будинок `unmarshal()` , який створює дерево об'єктів, або, навпаки, записати об'єкти Java в документи XML методом `marshal()` . Ці методи вже записані в створений компілятором `xjc` клас кореневого елемента, в прикладі це клас `Notebook` .

Об'єкти даних JDO

Завдання зв'язування даних природно узагальнити - пов'язувати об'єкти Java не тільки з документами XML, але і з текстовими файлами, реляційними або об'єктними ба- зами даних, іншими сховищами даних.

Корпорація Sun Microsystems опублікувала специфікацію JDO, зараз вже вір- ці 3.0, і розробила інтерфейси для роботи з JDO. Специфікація JDO розглядає- е більш широке завдання зв'язування даних, отриманих не тільки з документа XML, але і з будь-якого джерела даних, званого *інформаційної системою підприємства* (Enterprise Information System, EIS). Специфікація описує два набори класів та інтерфейсів:

- JDO SPI (JDO Service Provider Interface) - допоміжні класи та інтерфейси, які слід реалізувати на сервері додатків для звернення до джерела даних, створення об'єктів, забезпечення їх збереження, виконання транзакцій, перевірки прав доступу до об'єктам; ці класи та інтерфейси складають пакет `javax.jdo.spi` ;
- JDO API (JDO Application Programming Interface) - інтерфейси, що надаються користувачеві для доступу до об'єктів, управління транзакціями, створення та видалення ня об'єктів; ці інтерфейси зібрані в пакет `javax.jdo` .

Є багато комерційних і вільно поширюваних реалізацій інтерфейсів JDO.

Спільнота Apache Software Foundition назвало свою реалізацію Apache JDO. Цюрозробку можна, можливо подивитись за адресою **http://db.apache.org/jdo/index.html** .

Фірма DataNucleus, **http://www.datanucleus.org/** , випускає продукт Access Platform, колишній JPOX.

Компанія SolarMetric, **http://www.solarmetric.com/** , випускає свою реалізацію специ- Цифікації JDO під назвою Kodo JDO. Її можна вбудувати у сервери додатків WebLogic, WebSphere, JBoss.

Деякі фірми розробили свої варіанти JDO, більш менш відповідні специфікації. Найбільш відома розробка, що вільно розповсюджується, названа Кастор. Її можна, можливо подивитись за адресою **http://www.castor.org/** .

За допомогою Castor можна спростити зв'язування даних. Наприклад, створю- ня об'єкта Java з простого документа XML, якщо відволіктися від перевірок та обробки виняткових ситуацій, виконується одним дією:

```
Person person = (Person)Unmarshaller.unmarshal(
                    Person.class, new FileReader("person.xml"));
```

Назад, збереження об'єкта Java у вигляді документа XML у найпростішому випадку видивиться так:

```
Marshaller.marshall(person, new FileWriter("person.xml"));
```

У складніших випадках треба написати файл XML, аналогічний схемою XSD, з указаніями по *зв'язування даних* (mapping file).

Аналіз документів XML з допомогою DOM API

Як видно з попередніх розділів, SAX-парсер читає документ лише один раз. помічаючи теги, що з'являються по ходу читання, відкривають теги, вміст елементів і за- криві теги. Цього достатньо для зв'язування даних, але незручно для редагування. рування документа.

Консорціум W3C розробив специфікації і набір інтерфейсів DOM (Document Object Model), які можна переглянути на сайті цього проекту **http://www.w3.org/ DOM/** . Методами цих інтерфейсів документ XML можна завантажити в оперативну пам'ять у вигляді об'єктів дерева. Це дозволяє не тільки аналізувати документ ана- лізаторами, заснованими на структурі дерева, але і змінювати дерево, додаючи або уда- ляючи об'єкти з дерева. Крім того, можна звертатися безпосередньо до кожного об'єкту в дереві і не тільки читати, але і змінювати інформацію, що зберігається в ньому. Але все це вимагає великого обсягу оперативної пам'яті для завантаження великого деревіння.

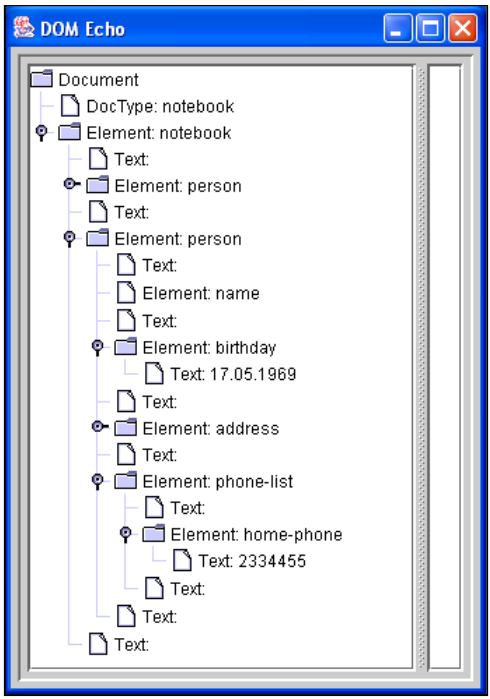
Корпорація Sun Microsystems реалізувала інтерфейси DOM в пакетах `javax.xml.parsers` і `org.w3c.dom` , вхідних в склад пакету JAXP. Скористатися цією реалізацією дужелегко:

```
DocumentBuilderFactory fact =
        DocumentBuilderFactory.newInstance();
DocumentBuilder builder = fact.newDocumentBuilder();
Document doc = builder.parse("ntb.xml");
```

Метод `parse()` будує дерево об'єктів та повертає посилання на нього у вигляді об'єкта типу `Document` . У класі `DocumentBuilder` є кілька методів `parse()` , що дозволяють загруфайл з адреси URL, з вхідного потоку, як об'єкт класу `File` або з джерела класу `InputSource` .

Інтерфейс `Document` , що розширює інтерфейс `Node` , описує дерево об'єктів документу загалом. Інтерфейс `Node` – основний інтерфейс в описі структури дерева . описує вузол дерева. У нього є ще один спадкоємець - інтерфейс `Element` . лист дерева, що відповідає елементу документа XML. Як видно з струк- тури успадкування цих інтерфейсів, і саме дерево, і його лист вважаються вузлами де- ревіння. Кожен атрибут елемента дерева описується інтерфейсом `Attr` . Ще кілька інтерфейсів - `CDATASection` , `Comment` , `Text` , `Entity` , `EntityReference` , `ProcessingInstruction` , `Notation` - описують різні типи елементів XML.

На рис. 28.2 показано початок дерева об'єктів, побудованого за документом, наведеному у лістингу 28.2. Зверніть увагу на те, що текст, що знаходиться в тілі еле- та, що зберігається в окремому вузлі дерева — нащадку вузла елемента. Для кожного атрибуту відкриває тега теж створюється окремий вузол.



Мал. 28.2. Дерево об'єктів документа XML

Інтерфейс *Node*

Інтерфейс `Node` описує тип вузла однієї з наступних констант:

- `ATTRIBUTE_NODE` - вузол типу `Attr` , містить атрибут елемента;
- `CDATA_SECTION_NODE` - вузол типу `CDADASection` , містить дані типу `CDATA` ;
- `COMMENT_NODE` - вузол типу `Comment` , містить коментар;
- `DOCUMENT_FRAGMENT_NODE` - в вузлі типу `DocumentFragment` знаходиться фрагмент документа;
- `DOCUMENT_NODE` - кореневий вузол типу `Document` ;
- `DOCUMENT_TYPE_NODE` - вузол типу `Document` ;
- `ELEMENT_NODE` - вузол є листом дерева типу `Element` ;
- `ENTITY_NODE` - в вузлі типу `Entity` зберігається сутність `ENTITY` ;
- `ENTITY_REFERENCE_NODE` - в вузлі типу `EntityReference` зберігається посилання на сутність;
- `NOTATION_NODE` - в вузлі зберігається нотація типу `Notation` ;
- `PROCESSING_INSTRUCTION_NODE` - вузол типу `ProcessingInstruction` , містить інструкціюпо обробці;
- `TEXT_NODE` - в вузлі типу `Text` зберігається текст.

Методи інтерфейсу `Node` описують дії з вузлом дерева:

- `public short getNodeType()` - повертає тип вузла;
- `public String getNodeName()` - повертає ім'я вузла;

- `public String getNodeValue()` - повертає значення, що зберігається в вузлі;
- `public boolean hasAttributes( )` виконує перевірку існування атрибутів у елемента XML, що зберігається у вузлі як об'єкт типу `NamedNodeMap` , якщо це вузол типу `Element` ;
- `public NamedNodeMap getAttributes()` - повертає атрибути; метод повертає `null` якщо у елемента немає атрибутів;
- `public boolean hasChildNodes()` - перевіряє, є чи у даного вузла вузли-нащадки;
- `public NodeList getChildNodes()` - повертає перелік вузлів-нащадків в вигляді об'єкта типу `NodeList` ;
- `public Node getFirstChild()` - повертає перший вузол в списку вузлів-нащадків;
- `public Node getLastChild()` - повертає останній вузол в списку вузлів-нащадків;
- `public Node getParentNode()` - повертає батьківський вузол;
- `public Node getPreviousSibling()` - повертає попередній вузол, має того жпредка, що і даний вузол;
- `public Node getNextSibling()` - повертає наступний вузол, що має того ж перед-ка, що і даний вузол;
- `public Document getOwnerDocument()` - повертає посилання на весь документ.

Наступні методи дозволяють змінити дерево об'єктів:

- `public Node appendChild(Node NewChild)` - додає новий вузол-нащадок `NewChild` ;
- `public Node insertBefore(Node NewChild, Node refChild)` - вставляє новий вузол-нащадок `newChild` перед існуючим нащадком `refChild` ;
- `public Node replaceChild(Node NewChild, Node oldChild)` - замінює один вузол-нащадок `oldChild` новим вузлом `NewChild` ;
- `public Node removeChild(Node child)` - видаляє вузол-нащадок.

Інтерфейс *Document*

Інтерфейс `Document` додає до методам свого предка `Node` методи роботи з документ-том в цілому:

- `public DocumentType getDocType()` - повертає загальні відомості про документ у вигляді об'єкта типу `DocumentType` ;
- `getName()` , `getEntitied()` , `getNotations()` і інші методи інтерфейсу `DocumentType` возобертають конкретні відомості про документ;
- `public Element getDocumentElement()` - повертає кореневий елемент дерева об'єктів;
- `public NodeList getElementsByTagName(String name)` ;

  `public NodeList getElementsByTagNameNS(String uri, String qname)`

  `;public Element getElementById(String id)`

  повертають Усе елементи з зазначеним ім'ям `tag` без префікса або з префіксом, а також елемент, визначається значенням атрибуту з ім'ям `ID` .

Декілька методів дозволяють змінити структуру і вміст дерева об'єктів:

- `public Element createElement(String name)` - створює новий порожній елемент по йогоімені;
- `public Element createElementNS(String uri, String name)` - створює новий порожній еле-мент по імені із префіксом;
- `public CDATASection createCDATASection(String name)` - створює вузол типу `CDATA_SECTION_NODE` ;
- `public EntityReference createEntityReference(String name )` створює вузол типу `ENTITY_ REFERENCE_NODE` ;
- `public ProcessingInstruction createProcessingInstruction(String name)` - створює вузол типу `PROCESSING_INSTRUCTION_NODE` ;
- `public TextNode createTextNode(String name)` - створює вузол типу `TEXT_NODE` ;
- `public Attr createAttribute(String name)` - створює вузол-атрибут з ім'ям `name` ;
- `public Attr createAttributeNS(String uri, String name)` - аналогічно;
- `public Comment createComment(String comment)` - створює вузол-коментар;
- `public DocumentFragment createDocumentFragment()` — створює порожній документ — фраг- мент даного документа з метою його подальшого заповнення;
- `public Node importNode(Node importedNode, boolean deep)` - вставляє створений вузол, отже, і все його піддерево, у дерево документа. Цим методом можна поєднати два дерева об'єктів. Якщо другий аргумент дорівнює `true` , то рекурсивно вставляється все поддерево.

Інтерфейс *Element*

Інтерфейс `Element` додає до методів свого предка `Node` методи роботи з атрибутами елемента XML, що відкриває тега, і методи, що дозволяють звернутися до вкладених елементів. Тільки один метод

```
public String getTagName();
```

дає відомості про самому елемент, а саме ім'я елемент.

Перш ніж отримати значення атрибуту з ім'ям `name` , потрібно перевірити його наявність методами

```
public boolean hasAttribute(String name);
public boolean hasAttributeNS(String uri, String name);
```

Другий із цих методів враховує простір імен з ім'ям `uri` , записаним у виглядірядки URI; ім'я `name` повинно бути повним, з префіксом.

Отримати атрибут в вигляді об'єкта типу `Attr` або його значення в вигляді рядки по імені
`name` з обліком префікса або без нього можна, можливо методами

```
public Attr getAttributeNode(String name);
public Attr getAttributeNodeNS(String uri, String name);
public String getAttribute(String name);
public String getAttributeNS(String uri, String name);
```

видалити атрибут можна, можливо методами

```
public Attr removeAttributeNode(Attr name);
public void removeAttribute(String name);
public void removeAttributeNS(String uri, String name);
```

Встановити значення атрибуту можна, можливо методами

```
public void setAttribute(String name, String value);
public void setAttributeNS(String uri, String name, String value);
```

Додати атрибут в якості нащадка можна, можливо методами

```
public Attr setAttributeNode(String name);
public Attr setAttributeNodeNS(Attr name);
```

Два методу дозволяють отримати перелік вузлів-нащадків:

```
public NodeList getElrmentsByTagName(String name);
public NodeList getElrmentsByTagNameNS(String uri, String name);
```

Отже, методи перерахованих інтерфейсів дозволяють переміщатися по дереву, змінювати структуру дерева, переглядати інформацію, що зберігається у вузлах та листі дерева, тазмінювати її.

Наведемо приклад роботи з деревом об'єктів, побудованим за документом XML. Добавим до адресної книжки лістингу 28.2 новий робочий або домашній телефон Сидоровиття. Ця дія записано в лістингу 28.11.

Листинг 28.11. Анализ адресной книжки с помощью DOM API

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
class ErrHand implements ErrorHandler{
  public void warning(SAXParseException ex)
  {
    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }

  public void error(SAXParseException ex){
    System.err.println("Error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }

  public void fatalError(SAXParseException ex){
    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
                       "  col = " + ex.getColumnNumber());
  }
}
```

```
public class TreeProcessDOM {

  static final String JAXP_SCHEMA_LANGUAGE = "
      http://java.sun.com/xml/jaxp/properties/schemaLanguage ";

  static final String W3C_XML_SCHEMA = "
      http://www.w3.org/2001/XMLSchema ";

  public static void main(String[] args) throws Exception{
    if (args.length != 3) {
      System.err.println("Usage: java TreeProcessDOM " +
            "<file-name>.xml {work|home} <phone>");
      System.exit(-1);
    }

 DocumentBuilderFactory fact =    DocumentBuilderFactory.newInstance();

 fact.setNamespaceAware(true);

 fact.setValidating(true);

    try{
       fact.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
    } catch (IllegalArgumentException x) {
       System.err.println("Невідоме властивість: " +
                          JAXP_SCHEMA_LANGUAGE);
       System.exit(-1);
 }

 DocumentBuilder builder = fact.newDocumentBuilder();

 builder.setErrorHandler(new ErrHand());

 Document doc = builder.parse(args[0]);

 NodeList list = doc.getElementsByTagName("notebook");

 int n = list.getLength();

    if (n == 0) {
      System.err.println("Документ
      порожній");System.exit(-1);
 }

 Node thisNode = null;

    for (int k = 0; k < n; k++){
      thisNode = list.item(k);
      String elemName = null;
      if (thisNode.getFirstChild() instanceof Element){
         elemName = (thisNode.getFirstChild()).getNodeName();
      if (elemName.equals("name")){
         if (!thisNode.hasAttributes()){
           System.err.println("Атрибути відсутні " + elemName);
```

```
            System.exit(1);
          }
          NamedNodeMap attrs = thisNode.getAttributes();
          Node attr = attrs.getNamedItem("surname");
          if (attr instanceof Attr)
             if (((Attr)attr).getValue().equals("Сидорова")) break;
       }
      }
}

   NodeList topics = ((Element)thisNode)
              .getElementsByTagName("phone-list");
Node NewNode;

   if (args[1].equals("work"))
        newNode = doc.createElement("work-phone");
else newNode = doc.createElement("home-phone");

Text textNode = doc.createTextNode(args[2]);

newNode.appendChild(textNode);

   thisNode.appendChild(newNode);
  }
}
```

Дерево об'єктів можна, можливо вивести на екран дисплея, наприклад, як об'єкт класу `JTree` – одного з компонентів графічної бібліотеки Swing. Саме так зроблено на Мал. 28.2. Для виведення застосовано програму DomEcho з електронного підручника "Web Services Tutorial". Вихідний текст програми занадто великий, щоб наводити його тут, але його можна, можливо подивитися по адресою **http://java.sun.com/webservices/tutorial.html** . До складу парсера Xerces як приклад аналізу документа розділ samples/ui/ входить програма TreeView, яка теж показує дерево об'єктів у вигляді дерева `JTree` бібліотеки Swing.

Інші DOM-парсери

Модель дерева об'єктів DOM була спочатку розроблена групою OMG (Object Management Group) в рамках мови IDL (Interface Definition Language) без урахування осо- бенностей Java. Тільки потім її було переведено на Java консорціумом W3C як інтерфейсів і класів, склали пакет `org.w3c.dom` . Цим пояснюється, в зокрема, широке застосування в DOM API інтерфейсів та фабричних методів замість класів та конструкторів.

Це незручність привело до появі інших розробок.

Учасники громадського проекту JDOM не стали реалізувати модель DOM, а роз- робили свою модель дерева об'єктів, що отримала назву JDOM. Вони випускають однойменний програмний продукт, що вільно розповсюджується, з яким можна ознайомитись на сайті проекту **http://www.jdom.org/** . Цей продукт широко використовува- ється для обробки документів XML засобами Java.

Учасники іншого громадського проекту – dom4j – прийняли модель W3C DOM, але спростили та впорядкували DOM API. З їх однойменним продуктом dom4j можна дізнатися. комитися на сайті **http://www.dom4j.org/** .

Перетворення дерева об'єктів в XML

Отже, дерево об'єктів DOM побудовано належним чином. Тепер треба його перетворити. зувати в документ XML, сторінку HTML, PDF або об'єкт іншого типу. Засоби для виконання такого перетворення становлять третину набору JAXP. пакети `javax.xml.transform` , `javax.xml.transform.dom` , `javax.xml.transform.sax` , `javax.xml.transform.stream` , які є реалізацією мови опису таблиць сти- лей для перетворень XSLT (XML Stylesheet Language for Transformations) засоби- ми Java.

Мова XSLT розроблена консорціумом W3 як одна з трьох частин, що становлять мову запис таблиць стилів XSL (XML Stylesheet Language). Всі матеріали по XSL можна подивитися на сайті проекту з адресою **http://www.w3.org/Style/XSL/** .

Інтерфейси та класи, що входять до пакетів `javax.xml.transform.*` , керують процесо- ром XSLT, в якості якого обраний процесор Xalan, розроблений в рамках про- екта Apache Software Foundation, **http://xml.apache.org/xalan-j/** .

Вихідний об'єкт перетворення повинен мати тип `Source` . Інтерфейс `Source` визна- ляє всього два методу доступу до ідентифікатора об'єкта:

```
public  String  getSystemId();
public void setSystemId(String id);
```

Інтерфейс `Source` має три реалізації. Клас `DOMSource` готує до перетворення. вання дерево об'єктів DOM, клас `SAXSource` готує SAX-об'єкт, а клас `StreamSource` - простий потік даних. У конструктори цих класів заноситься посилання на вихідний об'єкт - для конструктора класу `DOMSource` це вузол дерева, для конст- рутора класу `SAXSource` - ім'я файлу, для конструктора класу `StreamSource` - вхідний потік. Методи перерахованих класів дозволяють поставити додаткові властивостівихідних об'єктів перетворення.

Результат перетворення описується інтерфейсом `Result` . Він також визначає точно такі ж методи доступу до ідентифікатору об'єкта-результату, як і інтерфейс `Source` . У нього також є три реалізації - класи `DOMResult` , `SAXResult` і `StreamResult` . У конст- Руктори цих класів заноситься на вихідний об'єкт. У першому випадку це вузол дерева, в другому - об'єкт типу `ContentHandler` , в третьому - файл, в який буде занесений результат перетворення, або вихідний потік.

Саме перетворення виконується об'єктом класу `Transformer` . Ось стандартна схема перетворення дерева об'єктів DOM в документ XML, що записується в файл.

```
TransformerFactory transFactory = TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
DOMSource source = new DOMSource(document);
File newXMLFile = new File("ntb1.xml");
FileOutputStream fos = new FileOutputStream(newXMLFile);
```

```
StreamResult result = new StreamResult(fos);
transformer.transform(source, result);
```

Спочатку методом `newInstance()` створюється екземпляр `transfactory` фабрики об'єктів- перетворювачів. Методом

```
public void setAttrbute(String name, String value);
```

Класу `TransformerFactory` можна встановити деякі атрибути цього екземпляра. Імена і значення атрибутів залежать від реалізацію фабрики.

З допомогою фабрики перетворювачів створюється об'єкт-перетворювач класу `Transformer` . При формуванні цього об'єкта в нього можна занести об'єкт, що містить чий правила перетворення, наприклад таблицю стилів XSL.

У створений об'єкт класу `Transformer` методом

```
public void setParameter(String name, String value);
```

можна, можливо занести параметри перетворення, а методами

```
public void setOutputProperties(Properties out);
public void setOutputProperty(String name, String value);
```

легко визначити властивості перетвореного об'єкта. Імена властивостей `name` задаються константами , які зібрані в спеціально визначений клас `OutputKeys` щий тільки ці константи. Ось їх перелік:

- `CDATA_SECTION_ELEMENTS` - перелік імен секцій `CDATA` через пробіл;
- `DOCTYPE_PUBLIC` - відкритий ідентифікатор `PUBLIC` перетвореного документа;
- `DOCTYPE_SYSTEM` - системний ідентифікатор `SYSTEM` перетвореного документа;
- `ENCODING` - кодування символів перетвореного документа, значення атрибуту `encoding` оголошення XML;
- `INDENT` - робити чи відступи в тексті перетвореного документа. Значення цього властивості `"yes"` або `"no"` ;
- `MEDIA_TYPE` - MIME-тип вмісту перетвореного документа;
- `METHOD` - метод висновку, одне з значень: `"xml"` , `"html"` або `"text"` ;
- `OMIT_XML_DECLARATION` - не включати оголошення XML. Значення `"yes"` або `"no"` ;
- `STANDALONE` - окремий або вкладений документ, значення атрибуту `standalone` оголошення XML. Значення `"yes"` або `"no"` ;
- `VERSION` - номер версії XML для атрибуту `version` оголошення XML.

Наприклад, можна, можливо поставити кодування символів перетвореного документа наступнимметодом:

```
transformer.setOutputProperty(OutputKeys.ENCODING, "Windows-1251");
```

Потім у наведеному прикладі по дереву об'єктів `document` типу `Node` створюється об'єкт класу `DOMSource` - упаковка дерева об'єктів для наступного перетворення. Тип аргументу конструктора цього класу - `Node` _ звідки видно, що можна, можливо перетворити

не Усе дерево, а якесь його піддерево, записавши в конструкторі класу `DOMSource` кореневій вузол піддерева.

Нарешті визначається результуючий об'єкт `result`, пов'язаний з файломnewCourses.xml, і здійснюється перетворення методом `transform()`.

Більше складні перетворення виконуються з допомогою таблиці стилів XSL.

Таблиці стилів XSL

У документах HTML часто застосовуються таблиці стилів CSS (Cascading Style Sheet), загальні правила оформлення документів HTML: колір, шрифт, заголовки. Винаповнення цих правил надає документам єдиний стиль оформлення.

Для документів XML, де взагалі не визначаються правила візуалізації, ідея Застосувати таблиці стилів виявилася дуже плідною. Таблиці стилів для доку- ментів XML записуються на спеціально зробленій реалізації мови XML, названо- ної XSL (XML Stylesheet Language). Всі теги документів XSL відносяться до простору. ву імен з ідентифікатором `http://www.w3.org/1999/XSL/Transform`. Зазвичай вони записуються з префіксом `xsl`. Якщо прийнято цей префікс, то кореневий елемент таблиці стилів XSL буде називатися `<xsl:stylesheet>`.

Найпростіша таблиця стилів виглядає так, як записано в лістингу 28.12.

Листинг 28.12. Простейшая таблица стилей XSL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="
       http://www.w3.org/1999/XSL/Transform ">
   <xsl:output method="text" encoding="CP866"/>
</xsl:stylesheet>
```

Тут тільки визначається префікс простору імен `xsl` та правила виведення, а саме виводиться "плоский" текст, на що показує значення `text` (інші значення - `html` і `xml`), кодування CP866. Таке кодування вибрано для виведення кирилиці на консоль MS Windows. Об'єкт класу `Transformer`, керуючись таблицею стилів листин- га 28.12 просто виводить тіла елементів так, як вони записані в документі XML, тобто. просто видаляє теги разом з атрибутами, залишаючи їх вміст.

Цю таблицю стилів записуємо у файл, наприклад simple.xsl. Посилання на таблицю стилей можна помістити у документ XML як одну з інструкцій з обробці:

```
<?xml version="1.0" encoding="Windows-1251"?>
<?xml-stylesheet type="text/xsl" href="simple.xsl"?>
<notebook>
<!-- Продовження адресною книжки -->
```

Після цього XML-парсер, якщо він, крім того, є XSLT-процесором, виконає перетворення, задане в файл simple.xsl.

Інший шлях - використовувати таблицю стилів при створенні об'єкта-перетворювача, наприклад, так як записано в лістингу 28.13.

Листинг 28.13. Консольная программа преобразования документа XML

```
import java.io.*;
import  javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class SimpleTransform{

   public static void main(String[] args) throws TransformerException{
     if (args.length != 2) {
        System.out.println("Usage: " +
            "java SimpleTransform xmlFileName xsltFileName");
        System.exit(1);
 }

     File xmlFile = new File(args[0]);
 File xsltFile = new File(args[1]);

     Source xmlSource = новий StreamSource(xmlFile);
     Source xsltSource = new StreamSource(xsltFile);
 Result result = new StreamResult(System.out);

     TransformerFactory transFact = TransformerFactory.newInstance();
     Transformer trans = transFact.newTransformer(xsltSource);
     trans.transform(xmlSource, result);
  }
}
```

Після компіляції набираємо у командній рядку

```
java SimpleTransform ntb.xml simple.xsl
```

і отримуємо на екрані дисплея вміст тегів документа ntb.xml.

У складніших випадках просто змінюємо таблицю стилів. Наприклад, якщо ми хочемо отримати вміст документа на консолі, то таблицю стилів для об'єкта- перетворювача класу `Transformer` треба записати так, як показано в лістингу 28.14.

Листинг 28.14. Таблица стилей для показа адресной книжки

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="
    http://www.w3.org/1999/XSL/Transform ">

  <xsl:output method="text" encoding="CP866" />

  <xsl:template match="person">
       <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="name">
     <xsl:value-of select="@first" />  <xsl:text> </xsl:text>
     <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
```

```
    <xsl:value-of select="@surname" />
  </xsl:template>

  <xsl:template match="address">
    <xsl:value-of select="street" /> <xsl:text> </xsl:text>
    <xsl:value-of select="city" />   <xsl:text> </xsl:text>
    <xsl:value-of select="zip" />
  </xsl:template>

  <xsl:template match="phone-list">
    <xsl:value-of select="work-phone" /> <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="home-phone" /> <xsl:text>&#xA;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

Ми не будемо в даній книзі займатися мовою XSL - один його опис буде тов- ще всієї цієї книги. Російською мовою перекладено " Біблія " XSLT [20]. Її автор Майкл Кей (Michael H. Kay) створив та вільно поширює популярний XSLT-процесор Saxon, **http://saxon.sourceforge.net/** .

Перетворення документа XML в HTML

За допомогою мови XSL та методів класу `Transformer` можна легко перетворити документ XML у документ HTML. Достатньо написати відповідну таблицю стилів. У лістингу 28.15 показано, як це можна зробити для адресною книжки лістингу 28.2.

Листинг 28.15. Таблица стилей для преобразования XML в HTML

```
<?xml version="1.0" encoding="Windows-1251"?>

<xsl:stylesheet version="1.0" xmlns:xsl="
    http://www.w3.org/1999/XSL/Transform ">

<xsl:output method="html" encoding="Windows-1251"/>

  <xsl:template match="/">
    <html><head><title>Адресна книжка</title></head>
    <body><h2>Прізвища, адреси і телефони</h2>
       <xsl:apply-templates />
    </body></html>
</xsl:template>

  <xsl:template match="name">
    <p />
    <xsl:value-of select="@first" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@surname" /> <br> />
  </xsl:template>

  <xsl:template match="address">
     <br> />
```

```
    <xsl:value-of select="street" /> <xsl:text> </xsl:text>
    <xsl:value-of select="city" /> <xsl:text> </xsl:text>
    <xsl:value-of select="zip" /> <br> />
  </xsl:template>

  <xsl:template match="phone-list">
    Робочий: <xsl:value-of select="work-phone" /> <br />
    Домашній: <xsl:value-of select="home-phone" /> <br> />
  </xsl:template>

</xsl:stylesheet>
```

Цю таблицю стилів можна записати у файл, наприклад, ntb.xsl, і послатися на нього документі XML, що описує адресну книжку:

```
<?xml version="1.0" encoding="Windows-1251"?>
<?xml-stylesheet type="text/xsl" href="ntb.xsl"?>
<notebook>
<!-- Зміст адресною книжки -->
```

Після цього будь-який браузер, "розуміє" XML і XSLT, наприклад, Mozilla Firefox або Internet Explorer, покаже вміст адресної книжки, як наказано листин- гом 28.15.

Якщо набрати в командної рядку

```
$ java SimpleTransform ntb.xml ntb.xsl > ntb.html
```

то у поточному каталозі буде записаний перетворений HTML-файл з ім'ям ntb.html.

Запитання для самоперевірки

1. Навіщо знадобився новий мова розмітки XML?
2. Який Основний цілі служать елементи XML?
3. Яким чином виявляється сенс елементів XML?
4. Обов'язковий чи кореневий елемент в документі XML?
5. Чому документ XML повинен бути забезпечений описом DTD, схемою XSD або описомна якомусь іншою мовою?
6. Як виконується синтаксичний аналіз документа XML?
7. У яких випадках зручніше проводити синтаксичний аналіз документа XML, грунтуючись на подіях, а в яких - побудовою дерева?
8. Яким чином можна, можливо перетворити один документ XML в інший документ?
9. Чому в технології XML віддають перевагу використовувати таблиці стилів XSL, а не CSS?
10. Якими способами можна, можливо перебрати лише вузли-елементи?
11. Перетворюються чи вузли, для яких не написані правила перетворення?