"**_blank**" – в новое окно, "**_top**" – на все окно, "**_parent**" – в родительском окне, "**имя_окна**" – в окне с указанным именем. Для корректной работы данного примера апплет следует запускать из браузера, используя следующий HTML-документ:

```
<html>
<body align=center>
<applet code=chapt15.MyShowDocument.class></applet>
</body></html>
```

В следующей программе читается содержимое HTML-файла по указанному адресу и выводится в окно консоли.

```
/* пример # 4 : чтение документа из интернета: ReadDocument.java */
package chapt15;
import java.net.*;
import java.io.*;

public class ReadDocument {
   public static void main(String[] args) {
      try {
            URL lab = new URL("http://www.bsu.by");
            InputStreamReader isr =
                  new InputStreamReader(lab.openStream());
            BufferedReader d = new BufferedReader(isr);
            String line = "";
            while ((line = d.readLine()) != null) {
                  System.out.println(line);
            }
      } catch (MalformedURLException e) {
      // некорректно заданы протокол, доменное имя или путь к файлу
            e.printStackTrace();
      } catch (IOException e) {
            e.printStackTrace();
      }
   }
}
```

Сокетные соединения по протоколу TCP/IP

Сокеты (сетевые разъёмы) – это логическое понятие, соответствующее разъёмам, к которым подключены сетевые компьютеры и через которые осуществляется двунаправленная поточная передача данных между компьютерами. Сокет определяется номером порта и IP-адресом. При этом IP-адрес используется для идентификации компьютера, номер порта – для идентификации процесса, работающего на компьютере. Когда одно приложение знает сокет другого, создается сокетное протоколо-ориентированное соединение по протоколу TCP/IP. Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Сервер прослушивает сообщение и ждет, пока клиент не свяжется с ним. Первое сообщение, посылаемое клиентом на сервер, содержит сокет клиента. Сервер, в свою очередь, создает сокет, который будет использоваться для связи с клиентом, и

посылает его клиенту с первым сообщением. После этого устанавливается коммуникационное соединение.

Сокетное соединение с сервером создается клиентом с помощью объекта класса **Socket**. При этом указывается IP-адрес сервера и номер порта. Если указано символьное имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу. Например, если сервер установлен на этом же компьютере, соединение с сервером можно установить из приложения клиента с помощью инструкции:

```
Socket socket = new Socket("ИМЯ_СЕРВЕРА", 8030);
```

Сервер ожидает сообщения клиента и должен быть заранее запущен с указанием определенного порта. Объект класса **ServerSocket** создается с указанием конструктору номера порта и ожидает сообщения клиента с помощью метода **accept()** класса **ServerSocket**, который возвращает сокет клиента:

```
ServerSocket server = new ServerSocket(8030);
Socket socket = server.accept();
```

Таким образом, для установки необходимо установить IP-адрес и номер порта сервера, IP-адрес и номер порта клиента. Обычно порт клиента и сервера устанавливаются одинаковыми. Клиент и сервер после установления сокетного соединения могут получать данные из потока ввода и записывать данные в поток вывода с помощью методов **getInputStream()** и **getOutputStream()** или к **PrintStream** для того, чтобы программа могла трактовать поток как выходные файлы.

В следующем примере для отправки клиенту строки "привет!" сервер вызывает метод **getOutputStream()** класса **Socket**. Клиент получает данные от сервера с помощью метода **getInputStream()**. Для разъединения клиента и сервера после завершения работы сокет закрывается с помощью метода **close()** класса **Socket**. В данном примере сервер отправляет клиенту строку "привет!", после чего разрывает связь.

```
/* пример # 5 : передача клиенту строки : MyServerSocket.java */
package chapt15;
import java.io.*;
import java.net.*;

public class MyServerSocket {
     public static void main(String[] args) {
          Socket s = null;
          try { // отправка  строки клиенту
               //создание объекта и назначение номера порта
               ServerSocket server = new ServerSocket(8030);
               s = server.accept();//ожидание соединения
               PrintStream ps =
                    new PrintStream(s.getOutputStream());
               // помещение строки "привет!" в буфер
               ps.println("привет!");
               // отправка содержимого буфера клиенту и его очищение
               ps.flush();
               ps.close();
```

```
          } catch (IOException e) {
               System.err.println("Ошибка: " + e);
          } finally {
             if (s != null)
               s.close(); // разрыв соединения
          }
     }
}
/* пример # 6 : получение клиентом строки: MyClientSocket.java */
package chapt15;
import java.io.*;
import java.net.*;

public class MyClientSocket {
     public static void main(String[] args) {
          Socket socket = null;
          try { // получение строки клиентом
               socket = new Socket("ИМЯ_СЕРВЕРА", 8030);
                              /* здесь "ИМЯ_СЕРВЕРА" компьютер,
                              на котором стоит сервер-сокет"*/
               BufferedReader br =
                    new BufferedReader(
                         new InputStreamReader(
                              socket.getInputStream()));
               String msg = br.readLine();
               System.out.println(msg);
               socket.close();
          } catch (IOException e) {
               System.err.println("ошибка: " + e);
          }
       }
}
```

Аналогично клиент может послать данные серверу через поток вывода, полученный с помощью метода **getOutputStream()**, а сервер может получать данные через поток ввода, полученный с помощью метода **getInputStream()**.

Если необходимо протестировать подобный пример на одном компьютере, можно выступать одновременно в роли клиента и сервера, используя статические методы **getLocalHost()** класса **InetAddress** для получения динамического IP-адреса компьютера, который выделяется при входе в сеть Интернет.

Многопоточность

Сервер должен поддерживать многопоточность, иначе он будет не в состоянии обрабатывать несколько соединений одновременно. В этом случае сервер содержит цикл, ожидающий нового клиентского соединения. Каждый раз, когда клиент просит соединения, сервер создает новый поток. В следующем примере создается класс **ServerThread**, расширяющий класс **Thread**, и используется затем для соединений с многими клиентами, каждый в своем потоке.