

# BSF4ooRexx

Sockets ("java.net", "javax.net"), SSL/TLS ("javax.net.ssl")

**Business Programming 2**



**BSF4ooRexx**



**NetRexx**

Graphical  
User  
Interfaces  
(GUI)

Sockets  
SSL/TLS

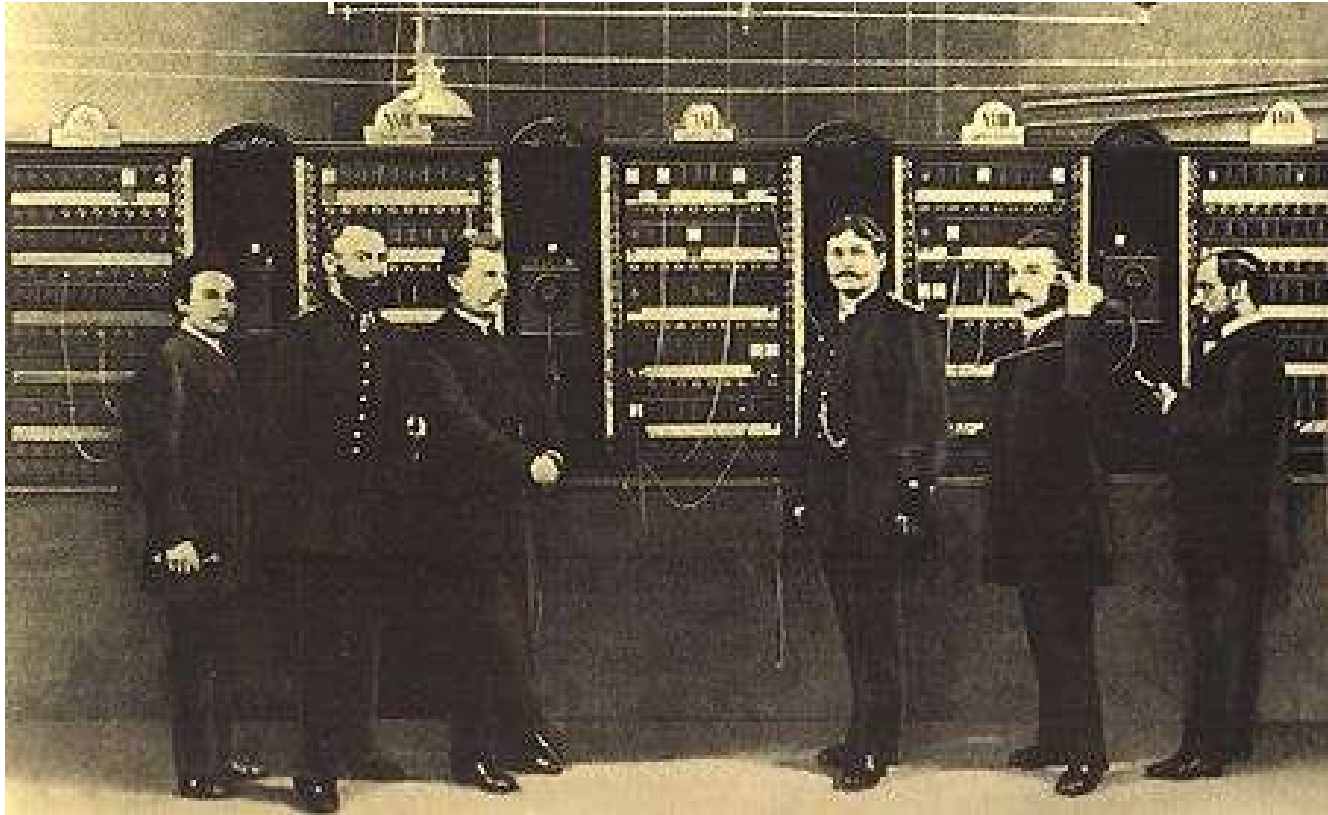
XML  
SAX/DOM  
JSON

(Java) Web  
Server:  
Tomcat

Java Classes  
written in  
Rexx style

- Operating system independency
  - Graphical and graphical user interface (GUI) programs should ideally run unchanged on at least
    - Linux
    - MacOS
    - Windows
  - Ideally wherever ooRexx is available
- "Omni-available"
  - Java and the Java runtime environment (JRE)
  - JRE already installed on most computers!
- Bridging ooRexx with Java
  - BSF4ooRexx

# Switchboard, 1



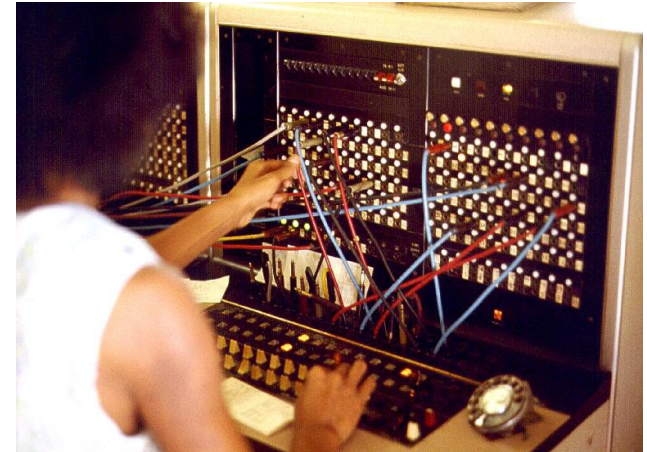
Switchboard, Berlin around 1881

- "Socket"
  - Concept originates from the early days of telephony
    - "Switchboard", which possesses a number of sockets (end-points), each with a single port
      - Each phone line leads to exactly one port
  - "Patch cords" (connection cables) are used to connect two sockets via their ports
    - Sockets are end-points of a bi-directional connection

# Sockets, 2



- "Socket" (Telephony)
  - (1) Switchboard operator answers call and asks for the callee
  - (2) Switchboard operator inserts a "phone-plug" into the ports of the caller and callee
  - (3) Caller and callee communicate bi-directionally



Joseph A. Carr, (C) 1975:  
JT Switchboard

# Sockets, 3: TCP/IP "Sockets"



- Caller and callee are two programs
- 65.536 ports
  - Numbered from 0 to 65.535
    - "Portnumber"
    - Port numbers 0 through 1024 are protected
      - E.g. on Unix systems not available to normal users
  - "Well-Known-Ports"
    - E.g. port number "80": reserved for WWW requests
      - A (server, daemon) program which must be addressed using the "http" protocol
      - E.g. Apache Server, Microsoft's IIS (Internet Information Server)



# Sockets, 4: TCP/IP "Sockets"



- Assignment of "port" numbers
  - Pre-defined "well-known ports": http, ftp, telnet, gopher, ...
    - Usually from the reserved number range 1 through 1024
  - File "services"
    - E.g. well-known port numbers for databases like DB2, Oracle, ...
  - Port numbers starting from 1025 freely available
    - Collisions, if a port number is already in use by another program
    - E.g. unprivileged users using port "8080" for their http daemon
      - Used quite often by programmers who are temporarily in a need for a web server (e.g. for developing or testing purposes)



# Switchboard, 2



- On your computer: 65.535 ports!
  - Ports to 1023 are known system ports, e.g.:
    - Port 22: Secure Shell Host (SSH)
    - Port 25: Simple Mail Transfer Protocol (SMTP)
    - Port 53: Domain Name System (DNS)
    - Port 67/68: Dynamic Host Configuration Protocol (DHCP)
    - Port 80: Hypertext Transfer Protocol (HTTP)
    - Port 443: HTTP Secure (HTTPS)
  - Port 1024 to 49151 are registered ports
  - Port 49152 to 65535 are dynamic or private ports



"Switchboard" with 100 ports



# IP-Addresses and Computer Names, 1



- Each computer possesses a **worldwide (!) unique** IP-Address
  - Internet "phone number" of a computer
  - Two versions of addresses in use
    - **IPv4-Addresses: 32-Bit** values(4 bytes), notation "A.B.C.D"
      - Each computer possesses usually a "loopback" address
        - [127.0.0.1](#)
        - "localhost"
      - Used for developing and testing of socket programs
    - **IPv6-Addresses: 128-Bit** values (16 bytes), different notations
      - Address space of IPv4 has become far too small
      - Newer version which will replace IPv4 addresses in the long run
      - Address range practically unlimited
  - Java class "[InetAddress](#)"




# IP-Addresses and Computer Names, 2



- Computer name
  - Easier to memorize for humans than "naked" numbers
  - "Name services"
    - Maintaining mapping of "computer name → IP address"
    - Makes it possible that a computer may have any number of names
      - If a computer possesses more than one name, all names resolve to the same IP address
    - Program "**nslookup**" (name server lookup)
      - Resolves computer names and IP addresses at the command line

# IP-Addresses and Computer Names, 3



- Computer name and domain
  - Each computer belongs to a domain
  - Domain names are maintained on a world-wide scope
    - Intended to assure name resolving works world-wide
  - Examples Resolution direction  
A blue horizontal line with a blue arrowhead pointing to the left, positioned below the word "Examples" and above the list of domain names.
    - www.wu.ac.at
    - www.keio.ac.jp
    - teletext.orf.at
    - www.RexxLA.org
    - zicklin.baruch.cuny.edu
    - www.ibm.com

# IP-Addresses and Computer Names, 4



- IP address objects in Java
  - Class "**java.net.InetAddress**"
  - Attention! No public constructors available!
    - Instead use the following static methods which return an instance of this class:

**InetAddress InetAddress.getName(String hostname)**

```
InetAddress ia=InetAddress.getName("www.wu.ac.at");
```

**InetAddress InetAddress.getByAddress(byte[] addr)**

```
byte addr[]={ (byte)137, (byte)208, (byte)3, (byte)112 };
```

```
InetAddress ia=InetAddress.getByAddress(addr);
```

**InetAddress InetAddress.getLocalHost()**

- Address of your own computer

```
InetAddress ia=InetAddress.getLocalHost();
```

# IP-Addresses and Computer Names, 5



```
import java.io.*;
import java.net.*;
import java.util.*;
class GetHostInfo
{
    public static void main (String args[])
    {
        InetAddress ia=null;
        if (args.length==0) // give localhost-infos
        {
            // attempt to get InetAddress object of localhost
            try { ia=InetAddress.getLocalHost(); }
            catch (UnknownHostException uhe)
            {
                System.err.println("No localhost defined for your computer!");
                System.exit(-1);
            }
        }
        else
        {
            // attempt to get InetAddress object by hostname
            try { ia=InetAddress.getByName(args[0]); }
            catch (UnknownHostException uhe)
            {
                try // attempt to get InetAddress object by IP address
                {
                    StringTokenizer st = new StringTokenizer(args[0], ".");
                    byte ab[] = new byte [st.countTokens()]; // create appropriate byte array
                    for (int i=0; st.hasMoreTokens();i++) // loop over tokens
                        ab[i]=Byte.parseByte(st.nextToken()); // get value and create byte of it
                    ia=InetAddress.getByAddress(ab);
                }
                catch (UnknownHostException uhe2)
                {
                    System.err.println("[ "+args[0]+" ]: not a valid hostname nor IP-address!");
                    System.exit(-1);
                }
            }
        }
        System.out.println("        HostName: ["+ia.getHostName()+"\n"+
            "        IP address: ["+ia.getHostAddress()+"\n\n"+
            "        canonicalHostName: ["+ia.getCanonicalHostName()+"\n"+
            "        toString(): ["+ia+"]");
    }
}
```

Possible Output:

```
        HostName: [waldi]
        IP address: [127.0.1.1]

        canonicalHostName: [waldi.localdomain]
        toString(): [waldi/127.0.1.1]
```

# Java Class "java.net.Socket"



- Allows to establish a connection to a server program on a(nother) computer
  - Computer will be identified by IP address or name and
  - Port number ("int")
- Input and output operations via "Stream" objects
  - "java.io.InputStream" object for reading (receiving) data
    - "getInputStream()"
  - "java.io.OutputStream" object for writing (sending) data
    - "getOutputStream()"
- Sockets get closed with the method "close()"

# Server Program "java.net.ServerSocket"



- "Server"
  - Program, which accepts requests for services
  - Must be assigned a port
    - Program listens at the port for requests from possible client programs
  - Server programs need to
    - Create a "**ServerSocket**" object
      - The port number to listen to can be supplied to the constructor or can be assigned later with the method "**bind(...)**"
    - Wait for a request to accept
      - Method "**accept()**"
    - Answering a request
      - Communication is carried out with the help of a **Socket** object, which the method "**accept()**" creates and returns

# Client Program: Client and Server

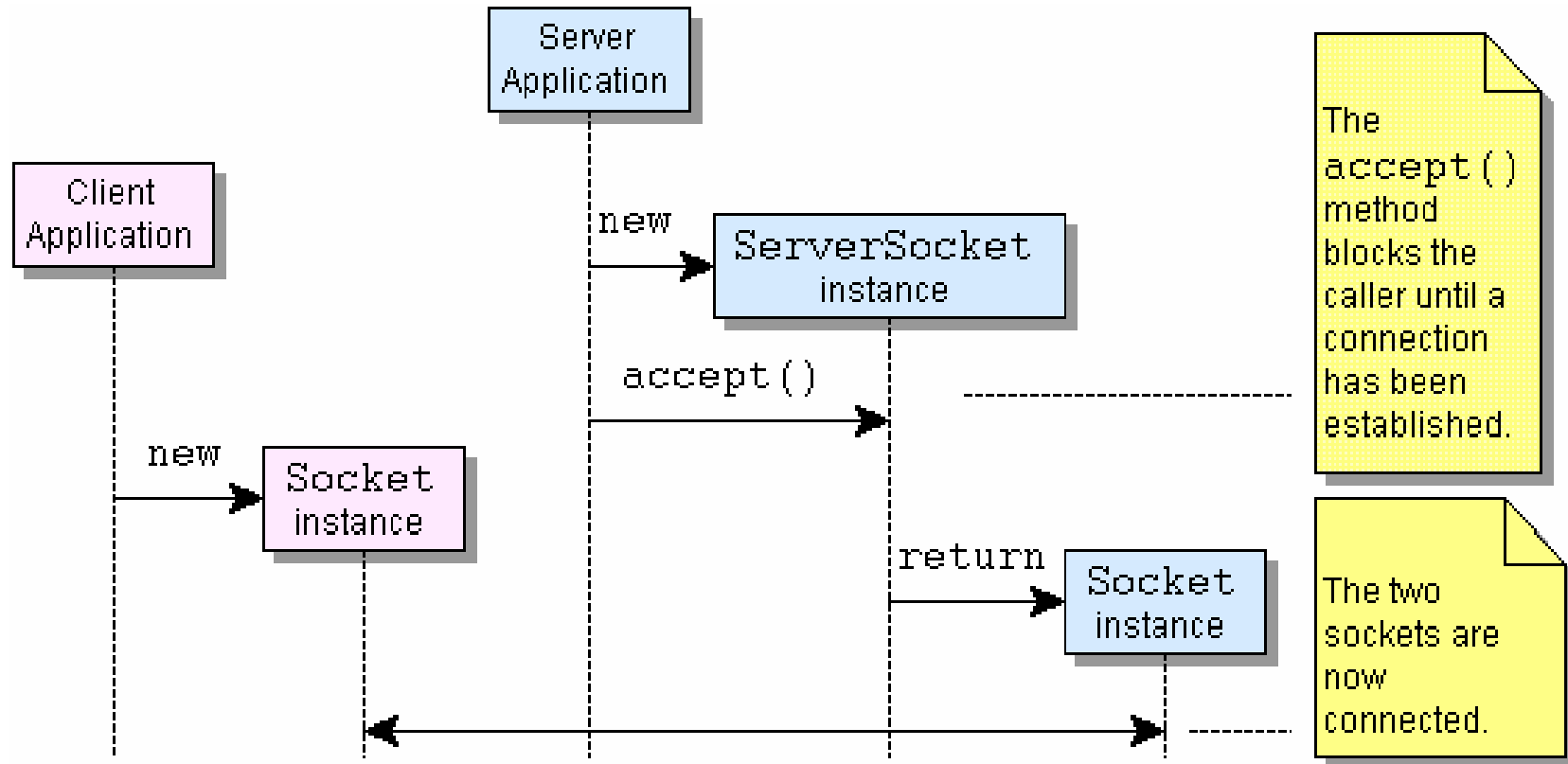


- "Client"
  - Program, which sends requests to a server
    - Creates a "**java.net.Socket**" object
      - Different constructors available
        - Possibility to denote the server computer (String, InetAddress)
        - Possibility to denote the desired port number
      - Possibility to denote computer and port number with the "connect(...)" method
- "Client" and "Server"
  - Need to define communication rules
    - "Protocol", e.g.
      - [http](#), [ftp](#), [telnet](#), [sendmail](#), ...
        - As RFCs ("request for comments") available on the Internet





# Client Program: Client and Server



# Example: Server Program (Java)



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

class TestSocketServer
{
    public static void main (String args[])
    {
        try
        {
            ServerSocket srvSock=new ServerSocket( 8888 ); // port to listen to

            Socket socket2client = null;
            System.out.println("Server (Java): starting to accept clients...");
            socket2client = srvSock.accept();
            System.out.println("Server (Java): client accepted.");

            byte b[]=new byte [2048]; // set read buffer to 2 KB
            int n=socket2client.getInputStream().read(b);
            System.out.println("Server (Java): received client data: ["+new String(b, 0, n)+"]");

            System.out.println("Server (Java): sending data to client ...");
            String msg="Hello from Server!";
            socket2client.getOutputStream().write(msg.getBytes());
        }
        catch (Exception exc)
        {
            System.err.println("Exception: ["+exc+"]");
            exc.printStackTrace();
            System.exit(-1);
        }
    }
}
```

# Example: Client Program (Java)



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

class TestSocketClient
{
    public static void main (String args[])
    {
        try
        {
            Socket socket2server= new Socket(InetAddress.getLocalHost(), 8888);
            System.out.println("CLIENT (Java): socket created!");

            System.out.println("CLIENT (Java): sending data to server ...");
            String msg="Hello, this is your client speaking!";
            socket2server.getOutputStream().write(msg.getBytes());

            byte b[]=new byte [100]; // set read buffer to 100 bytes auf
            int n=socket2server.getInputStream().read(b);
            System.out.println("CLIENT (Java): received server data: ["+new String(b,0,n)+"");
        }
        catch (Exception exc)
        {
            System.err.println("Exception: ["+exc+"");
            exc.printStackTrace();
            System.exit(-1);
        }
    }
}
```

# Example: Output



Server-side:

```
Server (Java): starting to accept clients...  
Server (Java): client accepted.  
Server (Java): received client data: [Hello, this is your client speaking!]  
Server (Java): sending data to client ...
```

Client-side:

```
CLIENT (Java): socket created!  
CLIENT (Java): sending data to server ...  
CLIENT (Java): received server data: [Hello from Server!]
```

# Example: Server Program (ooRexx)



```
say "SERVER (ooRexx): program started."

/* create server socket listening on port 8888 */
srvSock=.bsf~new("java.net.ServerSocket", 8888)
say "SERVER (ooRexx): starting to accept clients..."
socket2client=srvSock~accept -- accept client, returns socket to client
say "SERVER (ooRexx): client" pp(socket2client~toString) "accepted."

/* get data from the client */
b=.bsf~bsf.createArray('byte.class', 2048) -- create a "byte"-array
received=socket2client~getInputStream~read(b) -- returns number of bytes read
/* turn received byte array to Rexx string */
say "SERVER (ooRexx): data received from client:" pp(BsfRawBytes(b,received))

/* send data to client */
os=socket2client~getOutputStream
msg="Hello from the ooRexx-server!"
say "SERVER (ooRexx): sending" pp(msg) "to client..."
socket2client~getOutputStream~write(BsfRawBytes(msg)) -- turn Rexx string to byte array

say "SERVER (ooRexx): program ended."

::requires BSF.CLS -- get Java support
```

# Example: Client Program (ooRexx)



```
say "CLIENT (ooRexx): program started."

/* create socket and connect to server on port 8888 */
lh=.bsf~bsf.import('java.net.InetAddress') ~getLocalHost -- get InetAddress
socket2server=.bsf~new('java.net.Socket', lh, 8888) -- connect to server
say "CLIENT (ooRexx): connected to server" pp(socket2server~toString)

/* send the server data */
msg="Hello, this is from your ooRexx-client!"
say "CLIENT (ooRexx): sending" pp(msg) "to server..."
socket2server~getOutputStream~write(BsfRawBytes(msg)) -- turn Rexx string to byte array

/* receive data from server */
b=.bsf~bsf.createArray('byte.class', 100) -- create a "byte"-array
received=socket2server~getInputStream~read(b) -- returns number of bytes read
say "CLIENT (ooRexx): data received from server: ["||BsfRawBytes(b,received)"]"

say "CLIENT (ooRexx): program ended."

::requires BSF.CLS -- get Java support
```

# Example: Possible Output



## Server-side:

```
SERVER (ooRexx): program started.  
SERVER (ooRexx): starting to accept clients...  
SERVER (ooRexx): client [Socket[addr=/137.208.114.24,port=53813,localport=8888]] accepted.  
SERVER (ooRexx): data received from client: [Hello, this is from your ooRexx-client!]  
SERVER (ooRexx): sending [Hello from the ooRexx-server!] to client...  
SERVER (ooRexx): program ended.
```

## Client-side:

```
CLIENT (ooRexx): program started.  
CLIENT (ooRexx): connected to server [Socket[addr=T450sRGF/137.208.114.24,port=8888,localport=53813]]  
CLIENT (ooRexx): sending [Hello, this is from your ooRexx-client!] to server...  
CLIENT (ooRexx): data received from server: [Hello from the ooRexx-server!]  
CLIENT (ooRexx): program ended.
```

# Secure Socket Layers (SSL/TLS), 1: Overview



- Needs public key infrastructure
  - Complex to implement
  - However Java has that infrastructure "on-board!"
    - "keytool" to generate a certificate file
    - Certificate file needs to be stored on server and client
- Specialized socket classes to employ security
  - `"javax.net.ssl.SSLServerSocketFactory"`
  - `"javax.net.ssl.SSLSocketFactory"`
- Example modelled after tutorial on the Internet
  - Cf. [http://stilius.net/java/java\\_ssl.php](http://stilius.net/java/java_ssl.php) (as of: 2011-10-13)





# Secure Socket Layers (SSL/TLS), 2: Overview



- Java executable "`keytool`"
  - Create certificate file, e.g. with password "123456"  
`keytool -genkey -keystore mySrvKeystore -keyalg RSA`
  - Save certificate file on server and client
- Supply information about the certificate file either by setting "`java.lang.System`" properties or while invoking
  - SSL Server on the command line e.g. defining
    - `-Djavax.net.ssl.keyStore=mySrvKeystore -Djavax.net.ssl.keyStorePassword=123456`
  - SSL Client on the command line e.g. defining
    - `-Djavax.net.ssl.trustStore=mySrvKeystore -Djavax.net.ssl.trustStorePassword=123456`



# Example: SSL/TLS Server Program (ooRexx)



```
say "SERVER (ooRexx SSL/TLS): program started."
  -- define System properties for Java's security handling
system=.java.lang.System    -- gets preloaded by BSF4ooRexx
system~setProperty("javax.net.ssl.keyStore",      "mySrvKeystore")
system~setProperty("javax.net.ssl.keyStorePassword", "123456")

  -- create a SSL server socket, wait for a SSL client
socketFactory=bsf.loadClass("javax.net.ssl.SSLServerSocketFactory")~getDefault
srvSocket =socketFactory~createServerSocket(9999)
say "SERVER (ooRexx SSL/TLS): starting to accept clients..."
socket2client=srvSocket~accept    -- wait for a client
say "SERVER (ooRexx SSL/TLS): client" pp(socket2client~toString) "accepted."

  /* get data from the client */
b=.bsf~bsf.createArray('byte.class', 2048)    -- create a "byte"-array
n=socket2client~getInputStream~read(b)      -- returns number of bytes read
  /* turn received byte array into a Rexx string */
say "SERVER (ooRexx SSL/TLS): data received from client:" pp(BsfRawBytes(b,n))

  /* send data to client */
os=socket2client~getOutputStream
msg="Hello from the ooRexx-server via SSL/TLS!"
say "SERVER (ooRexx SSL/TLS): sending" pp(msg) "to client..."
socket2client~getOutputStream~write(BsfRawBytes(msg)) -- turn Rexx string to byte array

say "SERVER (ooRexx SSL/TLS): program ended."

::requires BSF.CLS          -- get Java support
```

# Example: SSL/TLS Client Program (ooRexx)



```
say "CLIENT (ooRexx SSL/TLS): program started."
  -- define System properties for Java's security handling
system=.java.lang.System    -- gets preloaded by BSF4ooRexx
system~setProperty("javax.net.ssl.trustStore",      "mySrvKeystore")
system~setProperty("javax.net.ssl.trustStorePassword", "123456")

  -- create a SSL server socket, wait for a SSL client
socketFactory=bsf.loadClass("javax.net.ssl.SSLSocketFactory")~getDefault
socket2server=socketFactory~createSocket("localhost", 9999) -- create socket
say "CLIENT (ooRexx SSL/TLS): connected to server" pp(socket2server~toString)

  /* send the server data */
msg="Hello, this is from your ooRexx-client via SSL/TLS!"
say "CLIENT (ooRexx SSL/TLS): sending" pp(msg) "to server..."
socket2server~getOutputStream~write(BsfRawBytes(msg)) -- turn Rexx string to byte array

  /* receive data from server */
b=.bsf~bsf.createArray('byte.class', 100) -- create a "byte"-array
n=socket2server~getInputStream~read(b)    -- returns number of bytes read
say "CLIENT (ooRexx SSL/TLS): data received from server: ["||BsfRawBytes(b,n)"]"

say "CLIENT (ooRexx SSL/TLS): program ended."

::requires BSF.CLS          -- get Java support
```

# SSL/TLS Example: Possible Output



## Server-side:

```
SERVER (ooRexx SSL/TLS): program started.  
SERVER (ooRexx SSL/TLS): starting to accept clients...  
SERVER (ooRexx SSL/TLS): client [SSLSocket[hostname=127.0.0.1, port=59276, Session(1655305614304|SSL_NULL_WITH_NULL_NULL)]] accepted.  
SERVER (ooRexx SSL/TLS): data received from client: [Hello, this is from your ooRexx-client via SSL/TLS!]  
SERVER (ooRexx SSL/TLS): sending [Hello from the ooRexx-server via SSL/TLS!] to client...  
SERVER (ooRexx SSL/TLS): program ended.
```

## Client-side:

```
CLIENT (ooRexx SSL/TLS): program started.  
CLIENT (ooRexx SSL/TLS): connected to server [SSLSocket[hostname=localhost, port=9999, Session(1655305617537|SSL_NULL_WITH_NULL_NULL)]]  
CLIENT (ooRexx SSL/TLS): sending [Hello, this is from your ooRexx-client via SSL/TLS!] to server...  
CLIENT (ooRexx SSL/TLS): data received from server: [Hello from the ooRexx-server via SSL/TLS!]  
CLIENT (ooRexx SSL/TLS): program ended.
```

# Secure Socket Layers (SSL): Echo Server



- Example modelled after tutorial on the Internet
  - Cf. [http://stilius.net/java/java\\_ssl.php](http://stilius.net/java/java_ssl.php) (as of: 2011-10-13)
- Differences to previous SSL/TLS example
  - Server echoes characters received from client, closes when empty line gets sent
  - **Streams** (byte oriented) get wrapped up in **Readers/Writers** (character oriented) which get wrapped up in **Buffers** (e.g. allowing reading/writing characters in line oriented)

# Example: SSL Echo Server Program (ooRexx)



```
-- define System properties for Java's security handling
system=.bsf4rexx~system.class      -- get preloaded java.lang.System
system~setProperty("javax.net.ssl.keyStore",      "mySrvKeystore")
system~setProperty("javax.net.ssl.keyStorePassword", "123456")

-- create a SSL server socket, wait for a SSL client
socketFactory=bsf.loadClass("javax.net.ssl.SSLServerSocketFactory")~getDefault
serverSocket =socketFactory~createServerSocket(9999)
sslSocket    =serverSocket~accept    -- wait for a client

-- client connected, start to read its data
inputStream  =sslSocket~getInputStream
inputStreamReader=.bsf~new("java.io.InputStreamReader", inputStream)
bufferedReader  =.bsf~new("java.io.BufferedReader",  inputStreamReader)

-- output text sent from SSL client
signal on syntax      -- activate signal handling, e.g. resetting socket
do until string=.nil
  string=bufferedReader~readLine
  if string<>.nil then
    say "<Rexx-SSL-server received>:" pp(string)
end
syntax:              -- label (target) for syntax exception

::requires BSF.CLS      -- get Java support
```

# Example: SSL Echo Client Program (ooRexx)



```
-- define System properties for Java's security handling
system=.bsf4rexx~system.class      -- get preloaded java.lang.System
system~setProperty("javax.net.ssl.trustStore",      "mySrvKeystore")
system~setProperty("javax.net.ssl.trustStorePassword", "123456")

-- create a SSL server socket, wait for a SSL client
socketFactory=bsf.loadClass("javax.net.ssl.SSLSocketFactory")~getDefault
sslSocket      =socketFactory~createSocket("localhost", 9999) -- create socket

-- client connected, start to read its data
outputStream      =sslSocket~getOutputStream
outputStreamWriter=.bsf~new("java.io.OutputStreamWriter", outputStream)
bufferedWriter    =.bsf~new("java.io.BufferedWriter",      outputStream)

-- get text from user, send it to SSL server
LF="0a"x          -- LF (linefeed) character
do until string=""
  parse pull string
  bufferedWriter ~~write( "["string"] "<from the Rexx SSL client>"LF) ~flush
end

::requires BSF.CLS      -- get Java support
```

- Using Java as an external function package using BSF4ooRexx
  - Camouflages Java as ooRexx
- Socket programming via Java is easy, allowing to take advantage of
  - IPv4 and IPv6 addresses
  - SSL/TLS
  - And much more ...
- Further information
  - Java tutorials that can easily be transcribed
  - Oracle Java documentation, books and tutorials