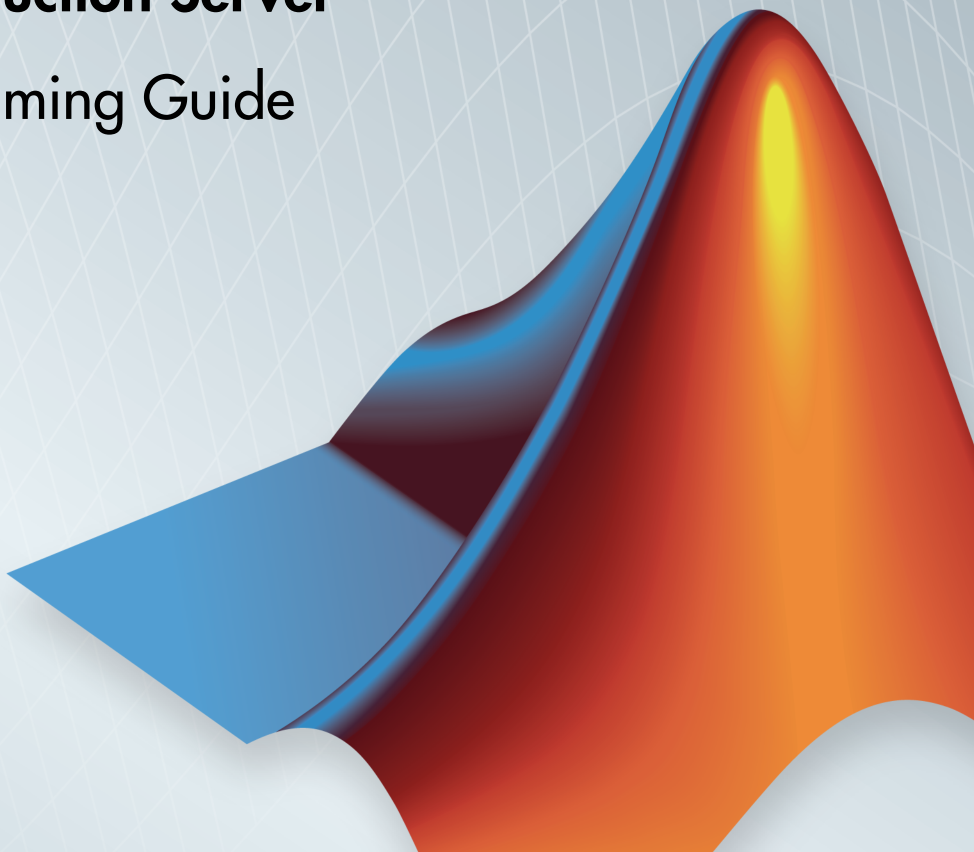
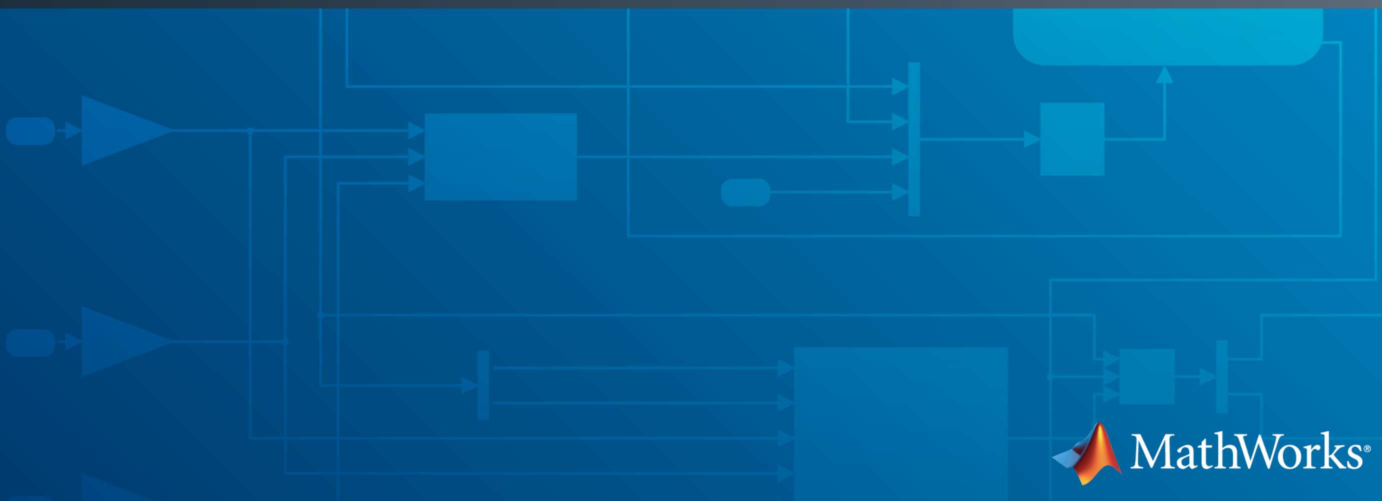
MATLAB® Production Server™

Java® Programming Guide

R2014**b**

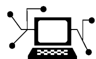


MATLAB®



MathWorks®

How to Contact MathWorks



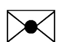
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000



The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Production Server™ Java® Programming Guide

© COPYRIGHT 2012–2014 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2014	Online only	New for Version 1.2 (Release R2014a)
October 2014	Online only	Revised for Version 2.0 (Release R2014b)

Contents

1 Client Programming

MATLAB Production Server Examples 1-2

Create a MATLAB Production Server Client in Java 1-3

Create a Java Application That Calls a Deployed Function . 1-4

Unsupported MATLAB Data Types for Client and Server Marshaling 1-8

2 Java Client Programming

Java Client Coding Best Practices 2-2
Static Proxy Interface Guidelines 2-2
Java Client Prerequisites 2-2
Manage Client Lifecycle 2-3
Handling Java Client Exceptions 2-4
Managing System Resources 2-4
Where to Find the Javadoc 2-4

Configure the Client-Server Connection 2-5
Create a Connection with the Default Configuration 2-5
Create a Connection with a Custom Configuration 2-5
Implement a Custom Connection Configuration 2-6

Invoke MATLAB Functions Dynamically 2-8
Create a Proxy for Dynamic Invocation 2-8
Invoke a MATLAB Function Dynamically 2-9
Marshal MATLAB Structures 2-11

Access Secure Programs Using HTTPS **2-13**
Overview **2-13**
Configure the Client's Environment for SSL **2-13**
Establish an HTTPS Connection **2-14**
Advanced Security Configuration **2-14**

Bond Pricing Tool for Java Client **2-17**
Objectives **2-17**
Step 1: Write MATLAB Code **2-17**
Step 2: Create a Deployable Archive with the Library Compiler App **2-18**
Step 3: Share the Deployable Archive on a Server **2-18**
Step 4: Create the Java Client Code **2-18**
Step 5: Build the Client Code and Run the Example **2-21**

Code Multiple Outputs for Java Client **2-23**

Code Variable-Length Inputs and Outputs for Java Client **2-24**

Marshal MATLAB Structures (Structs) in Java **2-25**
Marshaling a Struct Between Client and Server **2-25**

Data Conversion with Java and MATLAB Types **2-33**
Working with MATLAB Data Types **2-33**
Scalar Numeric Type Coercion **2-34**
Dimensionality in Java and MATLAB Data Types **2-35**
Empty (Zero) Dimensions **2-37**
Boxed Types **2-38**
Signed and Unsigned Types in Java and MATLAB Data Types **2-39**

A Data Conversion Rules

Conversion of Java Types to MATLAB Types **A-2**

Conversion of MATLAB Types to Java Types **A-4**

Client Programming

- “MATLAB Production Server Examples” on page 1-2
- “Create a MATLAB Production Server Client in Java” on page 1-3
- “Create a Java Application That Calls a Deployed Function” on page 1-4
- “Unsupported MATLAB Data Types for Client and Server Marshaling” on page 1-8

MATLAB Production Server Examples

Additional Client examples for MATLAB Production Server are available in the `client` folder of your MATLAB Production Server.

Create a MATLAB Production Server Client in Java

To create a MATLAB Production Server client in Java:

1. Obtain `mps_client.jar` from *$MPS_INSTALL*`/client`.
2. Configure your development environment to use `mps_client.jar`.
3. Based on your requirements, decide if the client uses a static proxy or a dynamic proxy.

   - A static proxy uses an object implementing an interface that mirrors the deployed MATLAB functions. You provide the interface for the static proxy.

     See "Static Proxy Interface Guidelines" on page 2-2.
   - A dynamic proxy creates server requests based on the MATLAB function name provided to the `invoke()` method. You provide the function name, the number of output arguments, and all of the input arguments required to evaluate the functions.

     See "Invoke MATLAB Functions Dynamically" on page 2-8.
4. Write a the Java code to instantiate a proxy to a MATLAB Production Server instance and call the MATLAB functions.

   a. Create an `MWClient` object for communicating with the service hosted by a MATLAB Production Server instance.
   b. Create MATLAB data structures to hold the data passed between the client and server.
   c. Invoke MATLAB functions.
   d. Free system resources using the `close` method of the `MWClient` object.

Create a Java Application That Calls a Deployed Function

This example shows how to write a MATLAB Production Server client using the Java client API. In your Java code, you will:

- Define a Java interface that represents the MATLAB function.
- Instantiate a proxy object to communicate with the server.
- Call the deployed function in your Java code.

To create a Java MATLAB Production Server client application:

1. Create a new file called `addmatrix_client.java`.
2. Using a text editor, open `addmatrix_client.java`.
3. Add the following import statements to the file:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;
```

4. Add a Java interface that represents the deployed MATLAB function.

   The interface for the `addmatrix` function

```
function a = addmatrix(a1, a2)

a = a1 + a2;
```

   looks like this:

```
interface MATLABAddMatrix {
       double[][] addmatrix(double[][] a1, double[][] a2)
         throws MATLABException, IOException;
   }
```

   When creating the interface, note the following:

   - You can give the interface any valid Java name.
   - You must give the method defined by this interface the same name as the deployed MATLAB function.
   - The Java method must support the same inputs and outputs supported by the MATLAB function, in both type and number. For more information about data

type conversions and how to handle more complex MATLAB function signatures, see “Java Client Programming”.

- The Java method must handle MATLAB exceptions and I/O exceptions.

5 Add the following class definition:

```
public class MPSClientExample
{
}
```

This class now has a single main method that calls the generated class.

6 Add the `main()` method to the application.

```
public static void main(String[] args)
{
}
```

7 Add the following code to the top of the `main()` method:

```
double[][] a1={{1,2,3},{3,2,1}};
double[][] a2={{4,5,6},{6,5,4}};
```

These statements initialize the variables used by the application.

8 Instantiate a client object using the `MWHttpClient` constructor.

```
MWClient client = new MWHttpClient();
```

This class establishes an HTTP connection between the application and the server instance.

9 Call the client object’s `createProxy` method to create a dynamic proxy.

You must specify the URL of the deployable archive and the name of your interface `class` as arguments:

```
MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
                                       MATLABAddMatrix.class);
```

The URL value (`"http://localhost:9910/addmatrix"`) used to create the proxy contains three parts:

- the server address (`localhost`).
- the port number (`9910`).
- the archive name (`addmatirx`)

For more information about the `createProxy` method, see the Javadoc included in the *$MPS_INSTALL*/`client` folder, where *$MPS_INSTALL* is the name of your MATLAB Production Server installation folder.

10 Call the deployed MATLAB function in your Java application by calling the public method of the interface.

```
double[][] result = m.addmatrix(a1,a2);
```

11 Call the client object's `close()` method to free system resources.

```
client.close();
```

12 Save the Java file.

The completed Java file should resemble the following:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;

interface MATLABAddMatrix
  {
    double[][] addmatrix(double[][] a1, double[][] a2)
         throws MATLABException, IOException;
  }

public class MPSClientExample {

    public static void main(String[] args){

        double[][] a1={{1,2,3},{3,2,1}};
        double[][] a2={{4,5,6},{6,5,4}};

        MWClient client = new MWHttpClient();

        try{
            MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
                                                    MATLABAddMatrix.class);
            double[][] result = m.addmatrix(a1,a2);

            // Print the magic square

            printResult(result);

        }catch(MATLABException ex){

            // This exception represents errors in MATLAB
               System.out.println(ex);
        }catch(IOException ex){

            // This exception represents network issues.
               System.out.println(ex);
        }finally{

            client.close();
```

```
        }
    }

    private static void printResult(double[][] result){
        for(double[] row : result){
            for(double element : row){
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

13 Compile the Java application, using the `javac` command or use the build capability of your Java IDE.

For example, enter the following (on one line):

```
H:\Work>javac -classpath "MPS_INSTALL_ROOT\client\java\mps_client.jar" addmatrix_client.java
```

14 Run the application using the `java` command or your IDE.

For example, enter the following (on one line):

```
H:\Work>java -classpath .;"MPS_INSTALL_ROOT\client\java\mps_client.jar" MPSClientExample
```

The application returns the following at the console:

```
5.0 7.0 9.0
9.0 7.0 5.0
```

Unsupported MATLAB Data Types for Client and Server Marshaling

These data types are not supported for marshaling between MATLAB Production Server instances and clients:

- MATLAB function handles
- Complex (imaginary) data
- Sparse arrays

Java Client Programming

- "Java Client Coding Best Practices" on page 2-2
- "Configure the Client-Server Connection" on page 2-5
- "Invoke MATLAB Functions Dynamically" on page 2-8
- "Access Secure Programs Using HTTPS" on page 2-13
- "Bond Pricing Tool for Java Client" on page 2-17
- "Code Multiple Outputs for Java Client" on page 2-23
- "Code Variable-Length Inputs and Outputs for Java Client" on page 2-24
- "Marshal MATLAB Structures (Structs) in Java" on page 2-25
- "Data Conversion with Java and MATLAB Types" on page 2-33

Java Client Coding Best Practices

Static Proxy Interface Guidelines

When you write Java interfaces to invoke MATLAB code, remember these considerations:

- The method name exposed by the interface *must* match the name of the MATLAB function being deployed.
- The method must have the same number of inputs and outputs as the MATLAB function.
- The method input and output types must be convertible to and from MATLAB.
- If you are working with MATLAB structures, remember that the field names are case sensitive and must match in both the MATLAB function and corresponding user-defined Java type.
- The name of the interface can be any valid Java name.

Java Client Prerequisites

Complete the following steps to prepare your MATLAB Production Server Java development environment.

1. Install a Java IDE of your choice. Follow instructions on the Oracle Web site for downloading Java, if needed.
2. Add `mps_client.jar` (located in *`$MPS_INSTALL`*`\client\java`) to your Java **`CLASSPATH`** and Build Path. This JAR file is sometimes defined in separate GUIs, depending on your IDE.

   Generate one deployable archive into your server's `auto_deploy` folder for each MATLAB application you plan to deploy. For information about creating a deployable archive with the Server Archive Compiler, see "Compile a Deployable Archive with the Production Server Compiler App".

   Your server's `main_config` file should point to where your MCR instance is installed.
3. The server hosting your deployable archive must be running.

Manage Client Lifecycle

A single Java client connects to one or more servers available at various URLs. Even though you create multiple instances of `MWHttpClient`, one instance is capable of establishing connections with multiple servers.

Proxy objects communicate with the server until the `close` method of that instance is invoked.

For a locally scoped instance of `MWHttpClient`, the Java client code looks like the following:

Locally Scoped Instance

```
MWClient client = new MWHttpClient();
try{
    // Code that uses client to communicate with the server
}finally{
    client.close();
}
```

When using a locally scoped instance of `MWHttpClient`, tie it to a servlet.

When using a servlet, initialize the `MWHttpClient` inside the `HttpServlet.init()` method, and `close` it inside the `HttpServlet.destroy()` method, as in the following code:

Servlet Implementation

```
public class MPSServlet extends HttpServlet{
    private final MWClient client;

    public void init(ServletConfig config) throws ServletException{
        client = new MWHttpClient();
    }

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException,java.io.IOException{

       // Code that uses client to communicate with the server
    }
```

```
    public void destroy(){
        client.close();
    }
}
```

Handling Java Client Exceptions

The Java interface must declare checked exceptions for the following errors:

Java Client Exceptions

Exception	Reason for Exception	Additional Information
com.mathworks.mps.client.MATLABExceptio	A MATLAB error occurred when a proxy object method was executed.	The exception provides the following: • MATLAB Stack trace • Error ID • Error message
java.io.IOException	• A network-related failure has occurred. • The server returns an HTTP error of either 4*xx* or 5*xx*.	Use java.io.IOException to handle an HTTP error of 4*xx* or 5*xx* in a particular manner.

Managing System Resources

A single Java client connects to one or more servers available at different URLs. Instances of MWHttpClient can communicate with multiple servers.

All proxy objects, created by an instance of MWHttpClient, communicate with the server until the close method of MWHttpClient is invoked.

Call close only if you no longer need to communicate with the server and you are ready to release the system resources. Closing the client terminates connections to all created proxies.

Where to Find the Javadoc

The API doc for the Java client is installed in *$MPS_INSTALL*/client.

Configure the Client-Server Connection

In this section...

"Create a Connection with the Default Configuration" on page 2-5
"Create a Connection with a Custom Configuration" on page 2-5
"Implement a Custom Connection Configuration" on page 2-6

You configure the client-server connection using an object that implements the `MWHttpClientConfig` interface. This interface defines these properties:

- `Interruptable` determines if MATLAB functions can be interrupted
- `TimeOutMs` determines the amount of time, in milliseconds, the client waits for a response before timing out
- `MaxConnections` - determines the maximum number of connections the client opens to fulfill multiple requests
- `ResponseSizeLimit` - determines the maximum size, in bytes, of the response a client accepts.

The API provides a default implementation, `MWHttpClientDefaultConfig`, that is automatically used when an HTTP client is instantiated. To modify the configuration, extend `MWHttpClientDefaultConfig` and pass it to the HTTP client constructor.

Create a Connection with the Default Configuration

When you create a client connection using the default constructor, `MWHttpClient()`, an instance of `MWHttpClientDefaultConfig` is automatically used to configure the client-server connection. The default configuration sets these connection properties:

- `Interruptable` = `false`
- `TimeOutMs` = `120000`
- `MaxConnections` = `-1`, specifying that the client uses as many connections as the system allows
- `ResponseSizeLimit` = `64*1024*1024` (64 MB)

Create a Connection with a Custom Configuration

To change one or more connection properties:

1 Implement a custom connection configuration by extending the `MWHttpClientDefaultConfig` interface.
2 Create the client connection using one of the constructors that accepts a configuration object:

   - `MWHttpClient(MWHttpClientConfig config)`
   - `MWHttpClient(MWHttpClientConfig config, MWSSLConfig sslConfig)`

This code sample creates a client connection with a timeout value of 1000 ms:

```
class MyClientConfig extends MWClientDefaultConfig
{
   public long getTimeOutMs()
   {
     return 1000;
   }
 }
 ...
 MWClient client = new MWHttpClient(new MyClientConfig());
 ...
```

Implement a Custom Connection Configuration

To implement a custom connection configuration extend the `MWHttpClientDefaultConfig` interface. The `MWHttpClientDefaultConfig` interface has one getter method for each configuration property:

- `public int getMaxConnectionsPerAddress()` returns the value for the maximum number of connections a client can use to handle simultaneous requests.
- `public long getTimeOutMs()` returns the number of milliseconds the client will wait for a response before generating an error.
- `public boolean isInterruptible()` returns a boolean specifying if the MATLAB function can be interrupted while waiting for a response.

  Note: If this method returns `true`, then `getMaxConnectionsPerAddress()` must return `-1`.

- `public int getResponseSizeLimit()` returns the maximum number of bytes the client can accept in a response.

You only need to overrides the getters for properties you wish to change. For example, if you want the MATLAB functions of a client to be interruptible, then specify an override for `isInterruptible()`:

```
class MyClientConfig extends MWClientDefaultConfig
 {
   public boolean isInterruptible()
   {
     return true;
   }
 }
```

To specify that a client times out after 1 s and can accept 4 MB responses, override `getTimeOutMs()` and `getResponseSizeLimit()`:

```
class MyClientConfig extends MWClientDefaultConfig
 {
   public long getTimeOutMs()
   {
     return 60000;
   }
   public int getResponseSizeLimit()
   {
     return 4*1024*1024;
   }
 }
```

Invoke MATLAB Functions Dynamically

In this section...
"Create a Proxy for Dynamic Invocation" on page 2-8
"Invoke a MATLAB Function Dynamically" on page 2-9
"Marshal MATLAB Structures" on page 2-11

To dynamically invoke functions on an MATLAB Production Server instance, you use a reflection-based proxy to construct the MATLAB function request. The function name and all of the inputs and outputs are passed as parameters to the method invoking the request. This means that you do not need to recompile your application every time you add a function to a deployed archive.

To dynamically invoke a MATLAB function:

1. Instantiate an instance of the `MWHttpClient` class.
2. Create a reflection-based proxy object using one of the `createComponentProxy()` methods of the client connection.
3. Invoke the function using one of the `invoke()` methods of the reflection-based proxy.

Create a Proxy for Dynamic Invocation

A reflection-based proxy implements the `MWInvokable` interface and provides methods that enables you to directly invoke any MATLAB function in a deployable archive. As with the interface-based proxy, the reflection-based proxy is created from the client connection object. The `MWHttpClient` class has two methods for creating a reflection-based proxy:

- `MWInvokable createComponentProxy(URL archiveURL)` creates a proxy that uses standard MATLAB data types.
- `MWInvokable createComponentProxy(URL archiveURL, MWMarshalingRules marshalingRules)` creates a proxy that uses structures.

To create a reflection-based proxy for invoking functions in the archive `myMagic` hosted on your local computer:

```
MWClient myClient = new MWHttpClient();
```

```
URL archiveURL = new URL("http://localhost:9910/myMagic");
MWInvokable myProxy =  myClient.createComponentProxy(archiveURL);
```

Invoke a MATLAB Function Dynamically

A reflection-based proxy has three methods for invoking functions on a server:

- Object[] invoke(final String functionName, final int nargout, final Class<T> targetType, final Object... inputs) invokes a function that returns nargout values.
- <T> T invoke(final String functionName, final Class<T> targetType, final Object... inputs) invokes a functions that returns a single value.
- invokeVoid(final String functionName, final Object... inputs) invokes a function that returns no values.

All methods map to the MATLAB function as follows:

- First argument is the function name
- Middle set of arguments, nargout and targetType, represent the return values of the function
- Last arguments are the function inputs

Return Multiple Outputs

The MATLAB function myLimits returns two values.

```
function [myMin,myMax] = myLimits(myRange)
 myMin = min(myRange);
 myMax = max(myRange);
end
```

To invoke myLimits from a Java client, use the invoke() method that takes the number of return arguments:

```
double[] myRange = new double[]{2,5,7,100,0.5};
try
{
  Object[] myLimits = myProxy.invoke("myLimits",
                                     2,
                                     Object[].class,
```

```
                                         myRange);
  double myMin = ((Double) myLimits[0]).doubleValue();
  double myMax = ((Double) myLimits[1]).doubleValue();
  System.out.printf("min: %f max: %f",myMin,myMax);
}
catch (Throwable e)
{
  e.printStackTrace();
}
```

Because Java cannot determine the proper types for each of the returned values, this form of `invoke` always returns `Object[]` and always takes `Object[].class` as the target type. You must cast the returned values into the proper types.

Return a Single Output

The MATLAB function `addmatrix` returns a single value.

```
function a = addmatrix(a1, a2)
a = a1 + a2;
```

To invoke `addmatrix` from a Java client, use the `invoke()` method that does not take the number of return arguments:

```
double[][] a1={{1,2,3},{3,2,1}};
double[][] a2={{4,5,6},{6,5,4}};
try
{
  Double[][] result = myProxy.invoke("addmatrix",
                                         Double[][].class,
                                         a1,
                                         a2);

  for(Double[] row : result)
  {
    for(double element : row)
    {
      System.out.print(element + " ");
    }
  }
} catch (Throwable e)
{
  e.printStackTrace();
}
```

Return No Outputs

The MATLAB function `foo` does not return value.

```
function foo(a1)
min(a1);
```

To invoke `foo` from a Java client, use the `invokeVoid()` method:

```
double[][] a={{1,2,3},{3,2,1}};
try
{
  myProxy.invokeVoid("foo", (Object)a);
}
catch (Throwable e)
{
  e.printStackTrace();
}
```

Marshal MATLAB Structures

If any MATLAB function in a deployable archive uses structures, you need to provide marshaling rules to the reflection-based proxy. To provide marshaling rules to the proxy:

1. Implement a new set of marshaling rules by extending the `MWDefaultMarshalingRules` interface to use a list of the classes being marshalled.
2. Create the proxy using the `createComponentProxy(URL archiveURL, MWMarshalingRules marshalingRules)` method.

The deployable archive `studentChecker` includes functions that use a MATLAB structure of the form

```
S =
name: 'Ed Plum'
score: 83
grade: 'B+'
```

Java client code represents the MATLAB structure with a class named `Student`. To create a marshalling rule for dynamically invoking the functions in `studentChecker`, create a class named `studentMarshaller`.

```
class studentMarshaller extends MWDefaultMarshalingRules
{
```

```
  public List<Class> getStructTypes() {
    List structType = new ArrayList<Class>();
    structType.add(Student.class);
    return structType;
  }
}
```

Create the proxy for `studentChecker` by passing `studentMarshaller` to `createComponentProxy()`.

```
URL archiveURL = new URL("http://localhost:9910/studentCheck");
myProxy =  myClient.createComponentProxy(archiveURL,
                                          new StudentMarshaller());
```

For more information about using MATLAB structures, see "Marshal MATLAB Structures (Structs) in Java" on page 2-25.

Access Secure Programs Using HTTPS

In this section...
“Overview” on page 2-13
“Configure the Client’s Environment for SSL” on page 2-13
“Establish an HTTPS Connection” on page 2-14
“Advanced Security Configuration” on page 2-14

Overview

Connecting to a MATLAB Production Server instance over HTTPS provides a secure channel for executing MATLAB functions. To establish an HTTPS connection with a MATLAB Production Server instance:

1. Ensure that the server is configured to use HTTPS.
2. Install the required credentials on the client system.
3. Configure the client's Java environment to use the credentials.
4. Create the program proxy using the program's `https://` URL.

MATLAB Production Server Java client API provides:

- Hooks for providing your own `HostnameVerifier` implementation
- Hooks for implementing server authorization beyond that provided by HTTPS

Configure the Client’s Environment for SSL

To manage the key store and trust stores on the client machine, use `keytool`.

At a minimum the client requires the server's root CA (Certificate Authority) in its trust store.

To connect to a server that requires client-side authentication, the client also needs a signed certificate in its key store.

For information on using `keytool` see Oracle's keytool documentation.

Establish an HTTPS Connection

Create a secure proxy connection with a MATLAB Production Server instance by using the `https://` URL for the desired program:

```
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://hostname:port/myArchive");
MyProxy sslProxy = client.createProxy(sslURL, MyProxy.class);
```

The `sslProxy` object uses the default Java trust store, stored in `JAVA_HOME\lib\security\cacerts`, to perform the HTTPS server authentication. If the server requests client authentication, the HTTPS handshake fails because the default `SSLContext` object created by the JRE does not provide a key store.

To enable the client to connect with a server instance requiring client authentication, set the key store location and password using Java system properties:

```
System.setProperty("javax.net.ssl.keyStore", "PATH_TO_KEYSTORE");
System.setProperty("javax.net.ssl.keyStorePassword", "keystore_pass");
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://hostname:port/myfun");
MyProxy sslProxy = client.createProxy(sslURL, MyProxy.class);
```

To use a location other than the default for the client trust store, set the trust store location and password using Java system properties:

```
System.setProperty("javax.net.ssl.trustStore", "PATH_TO_TRUSTSTORE");
System.setProperty("javax.net.ssl.trustStorePassword", "truststore_pass");
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://hostname:port/myfun");
MyProxy sslProxy = client.createProxy(sslURL, MyProxy.class);
```

You must provide a custom implementation of the `MWSSLConfig` interface to use a custom `SSLContext` implementation, add a custom `HostnameVerifier` implementation, or use the server authorization of the MATLAB Production Server Java client API. See “Advanced Security Configuration” on page 2-14.

Advanced Security Configuration

- “SSL API Configuration” on page 2-15
- “Override Default Hostname Verification” on page 2-15
- “Use Additional Server Authentication” on page 2-16

SSL API Configuration

The Java API uses an `MWSSLConfig` object to get the information it needs to use HTTPS and perform the additional server authorization. The `MWSSLConfig` interface has three methods:

- `getSSLContext()` — Returns the `SSLContext` object
- `getHostnameVerifier()` — Returns a `HostnameVerifier` object to use if HTTPS hostname verification fails
- `getServerAuthorizer()` — Returns a `MWSSLServerAuthorizer` object to perform server authorization based on the server's certificate

The Java client API provides a default `MWSSLConfig` implementation, `MWSSLDefaultConfig`, which it uses when no SSL configuration is passed to the `MWHTTPClient` constructor. The `MWSSLDefaultConfig` object is implemented such that:

- `getSSLContext()` returns the default `SSLContext` object created by the JRE
- `getHostnameVerifier()` returns a `HostnameVerifier` implementation that always returns false. If the HTTPS hostname verification fails, this will not override the decision.
- `getServerAuthorizer()` returns a `MWSSLServerAuthorizer` implementation that authorizes all MATLAB Production Server instances.

Override Default Hostname Verification

As part of the SSL handshake, the HTTPS layer attempts to match the hostname in the provided URL to the hostname provided in the server's certificate. If the two hostnames do not match, the HTTPS layer calls the `HostnameVerifier.verify()` method as an additional check. The return value of the `HostnameVerifier.verify()` method determines if the hostname is verified.

The implementation of the `HostnameVerifier.verify()` method provided by the `MWSSLDefaultConfig` object always returns `false`. The result is that if the hostname in the URL and the hostname in the server certificate do not match, the HTTPS handshake fails.

For a more robust hostname verification scheme, extend the `MWSSLDefaultConfig` class to return an implementation of `HostnameVerifier.verify()` that uses custom logic. For example, if you only wanted to generate one certificate for all of the servers on which

MATLAB Production Server instances run, you could specify MPS for the certificate's hostname. Then your implementation of HostnameVerifier.verify() returns true if the certificate's hostname is MPS.

```
public class MySSLConfig extends MWSSLDefaultConfig {
  public HostnameVerifier getHostnameVerifier() {
    return new HostNameVerifier() {
      public boolean verify(String s, SSLSession sslSession) {
        if (sslSession.getPeerHost().equals("MPS"))
          return true;
        else
          return false;
      }
    }
  }
}
```

For more information on HostnameVerify see Oracle's Java Documentation.

Use Additional Server Authentication

After the HTTPS layer establishes a secure connection, a client can perform an additional authentication step before sending requests to a server. This additional authentication is performed by an implementation of the MWSSLServerAuthorizer interface. An MWSSLSServerAuthorizer implementation performs two checks to authorize a server:

- isCertificateRequired() determines if servers must present a certificate for authorization. If this returns true and the server has not provided a certificate, the client does not authorize the server.
- authorize(Certificate serverCert) uses the server's certificate to determine if the client authorizes the server to process requests.

The MWSSLSServerAuthorizer implementation returned by the MWSSLDefaultConfig object authorizes all servers without performing any checks.

To use server authentication, extend the MWSSLDefaultConfig class and override the implementation of getServerAuthorizer() to return a MWSSLSServerAuthorizer implementation that does perform authorization checks.

Bond Pricing Tool for Java Client

This example shows an application that calculates a bond price from a simple formula.

You run this example by entering the following known values into a simple graphical interface:

- Coupon payment — `C`
- Number of payments — `N`
- Interest rate — `i`
- Value of bond or option at maturity — `M`

The application calculates price (`P`) based on the following equation:

```
P = C * ( (1 - (1 + i)^-N) / i ) + M * (1 + i)^-N
```

Objectives

The Bond Pricing Tool demonstrates the following features of MATLAB Production Server:

- Deploying a simple MATLAB function with a fixed number of inputs and a single output
- Deploying a MATLAB function with a simple GUI front-end for data input
- Using `dispose()` to free system resources

Step 1: Write MATLAB Code

Implement the Bond Pricing Tool in MATLAB, by writing the following code. Name the code `pricecalc.m`.

Sample code is available in *`MPS_INSTALL`*`\client\java\examples\BondPricingTool\MATLAB`.

```
function price = pricecalc(value_at_maturity, coupon_payment,...
                           interest_rate, num_payments)

    C = coupon_payment;
    N = num_payments;
    i = interest_rate;
```

```
    M = value_at_maturity;

    price = C * ( (1 - (1 + i)^-N) / i ) + M * (1 + i)^-N;

end
```

Step 2: Create a Deployable Archive with the Library Compiler App

To create the deployable archive for this example:

1. From MATLAB, select the Library Compiler App.
2. In the **Application Type** list, select **Deployable Archive**.
3. In the **Exported Functions** field, add `pricecalc.m`.

   `pricecalc.m` is located in *`MPS_INSTALL`*`\client\java\examples\BondPricingTool\MATLAB`.
4. Under **Application Information**, change `pricecalc` to `BondTools`.
5. Click **Package**.

The generated deployable archive, `BondTools.ctf` is located in the `for_redistribution_files_only` of the project's folder.

Step 3: Share the Deployable Archive on a Server

1. Download the MATLAB runtime, if needed, at http://www.mathworks.com/products/compiler/mcr. See "Download and Install the MATLAB Runtime" for more information.
2. Create a server using `mps-new`. See "Create a Server" for more information.
3. If you have not already done so, specify the location of the MATLAB runtime to the server by editing the server configuration file, `main_config` and specifying a path for `--mcr-root`. See "Edit the Configuration File" for details.
4. "Start the server using `mps-start`" and "verify it is running with `mps-status`".
5. Copy the `BondTools.ctf` file to the `auto_deploy` folder on the server for hosting.

Step 4: Create the Java Client Code

Create a compatible client interface and define methods in Java to match MATLAB function `pricecalc.m`, hosted by the server as `BondTools.ctf`, using the guidelines in this section.

Additional Java files are also included that are typical of a standalone application. You can find the example files in *MPS_INSTALL*\client\java\examples\BondPricingTool\Java.

This Java code...	Provides this functionality...
BondPricingTool.java	Runs the calculator application. The variable values of the pricing function are declared in this class.
BondTools.java	Defines pricecalc method interface, which is later used to connect to a server to invoke pricecalc.m
BondToolsFactory.java	Factory that creates new instances of BondTools
BondToolsStub.java	Java class that implements a dummy pricecalc Java method. Creating a stub method is a technique that allows for calculations and processing to be added to the application at a later time.
BondToolsStubFactory.java	Factory that returns new instances of BondToolsStub
RequestSpeedMeter.java	Displays a GUI interface and accepts inputs using Java Swing classes
ServerBondToolsFactory.java	Factory that creates new instances of MWHttpClient and creates a proxy that provides an implementation of the BondTools interface and allows access to pricecalc.m, hosted by the server

When developing your Java code, note the following essential tasks, described in the sections that follow. For more information about clients coding basics and best practices, see "Java Client Coding Best Practices" on page 2-2.

This documentation references specific portions of the client code. You can find the complete Java client code in *MPS_INSTALL*\client\java\examples\BondPricingTool\Java.

Declare Java Method Signatures Compatible with MATLAB Functions You Deploy

To use the MATLAB functions you defined in "Step 1: Write MATLAB Code" on page 2-17, declare the corresponding Java method signature in the interface `BondTools.java`:

```
interface BondTools {
    double pricecalc (double faceValue,
                      double couponYield,
                      double interestRate,
                      double numPayments)
            throws IOException, MATLABException;
}
```

This interface creates an array of primitive `double` types, corresponding to the MATLAB primitive types (`Double`, in MATLAB, unless explicitly declared) in `pricecalc.m`. A one to one mapping exists between the input arguments in both the MATLAB function and the Java interface The interface specifies compatible type `double`. This compliance between the MATLAB and Java signatures demonstrates the guidelines listed in "Java Client Coding Best Practices" on page 2-2.

Instantiate MWClient, Create Proxy, and Specify Deployable Archive

In the `ServerBondToolsFactory` class, perform a typical MATLAB Production Server client setup:

1. Instantiate `MWClient` with an instance of `MWHttpClient`:

   ```
   ...
       private final MWClient client = new MWHttpClient();
   ```

2. Call `createProxy` on the new client instance. Specify port number (`9910`) and the deployable archive name (`BondTools`) the server is hosting in the `auto_deploy` folder:

   ```
   ...
    public BondTools newInstance () throws Exception
       {
        return client.createProxy(new URL("http://user1.dhcp.mathworks.com:9910/BondTools"),
                                                                              BondTools.class);
       }
   ...
   ```

Use dispose() Consistently to Free System Resources

This application makes use of the Factory pattern to encapsulate creation of several types of objects.

Any time you create objects—and therefore allocate resources—ensure you free those resources using `dispose()`.

For example, note that in `ServerBondToolsFactory.java`, you dispose of the `MWHttpClient` instance you created in "Instantiate MWClient, Create Proxy, and Specify Deployable Archive" on page 2-20 when it is no longer needed.

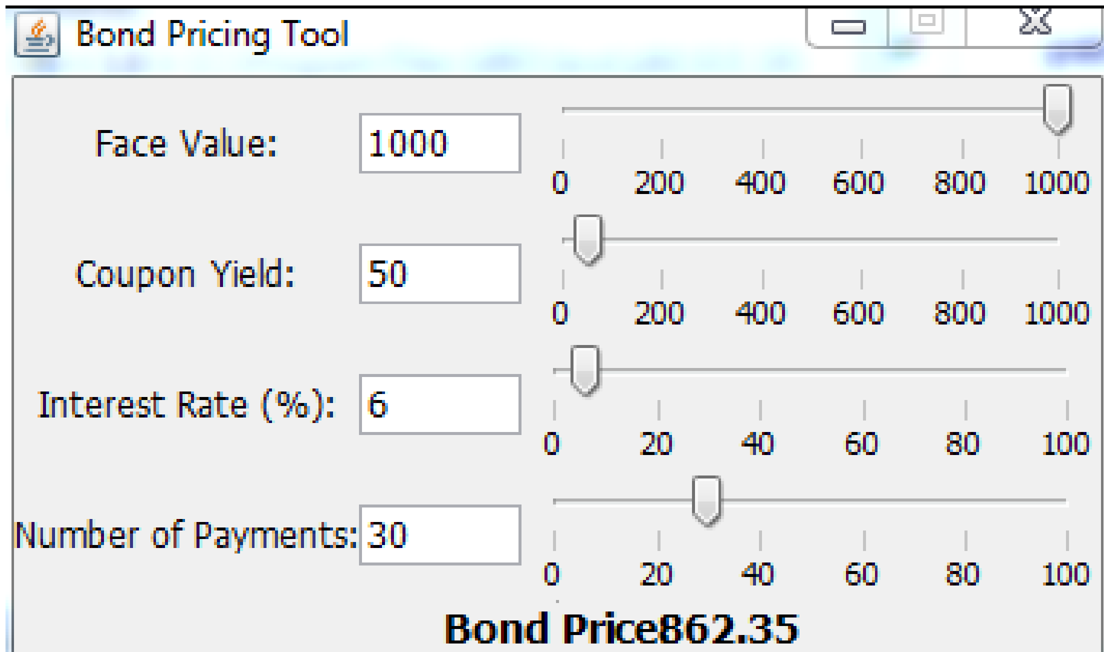
Additionally, note the `dispose()` calls to clean up the factories in `BondToolsStubFactory.java` and `BondTools.java`.

Step 5: Build the Client Code and Run the Example

Before you attempt to build and run your client code, ensure that you have done the following:

- Added `mps_client.jar` (*$MPS_INSTALL*`\client\java`) to your Java **`CLASSPATH`** and Build Path.
- Copied your deployable archive to your server's `auto_deploy` folder.
- Modified your server's `main_config` file to point to where your MCR is installed.
- "Started your server" and "verified it is running".

When you run the calculator application, you should see the following output:


Bond Pricing Tool
Face Value:
1000
0 200 400 600 800 1000
Coupon Yield:
50
0 200 400 600 800 1000
Interest Rate (%):
6
0 20 40 60 80 100
Number of Payments:
30
0 20 40 60 80 100
Bond Price862.35

Code Multiple Outputs for Java Client

MATLAB allows users to write functions that return multiple outputs.

For example, consider this MATLAB function signature:

```
function [out_double_array, out_char_array] =
                 multipleOutputs (in1_double_array, in2_char_array)
```

In the MATLAB signature, `multipleOutputs` has two outputs (`out_double_array` and `out_char_array`) and two inputs (`in1_double_array` and a `in2_char_array`, respectively)—a double array and a char array.

In order to call this function from Java, the interface in the client program must specify the number of outputs of the function as part of the function signature.

The number of expected output parameters in defined as type integer (`int`) and is the first input parameter in the function.

In this case, the matching signature in Java is:

```
public Object[] multipleOutputs(int num_args, double[]
                        in1Double, String in2Char);
```

where `num_args` specifies number of output arguments returned by the function. All output parameters are returned inside an array of type `Object`.

Note: When coding multiple outputs, if you pass an integer *as the first input argument* through a MATLAB function, you must wrap the integer in a `java.lang.Integer` object.

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the Java method signature using the name `multipleOutputs`. Both signatures define two inputs and two outputs.
- MATLAB Java interface supports direct conversion from Java double array to MATLAB double array and from Java string to MATLAB char array. For more information, see “Conversion of Java Types to MATLAB Types” on page A-2 and “Conversion of MATLAB Types to Java Types” on page A-4.

For more information, see “Java Client Coding Best Practices” on page 2-2.

Code Variable-Length Inputs and Outputs for Java Client

MATLAB supports functions with both variable number of input arguments (`varargin`) and variable number of output arguments (`varargout`).

MATLAB Production Server Java client supports the ability to work with variable-length inputs (`varargin`) and outputs (`varargout`). `varargin` supports one or more of any data type supported by MATLAB. See the *MATLAB Function Reference* for complete information on `varargin` and `varargout`.

For example, consider this MATLAB function:

```
function varargout = vararginout(double1, char2, varargin)
```

In this example, the first input is type double (`double1`) and the second input type is a char (`char2`). The third input is a variable-length array that can contain zero, or one or more input parameters of valid MATLAB data types.

The corresponding client method signature must include the same number of output arguments as the first input to the Java method.

Therefore, the Java method signature supported by MATLAB Production Server Java client, for the `varargout` MATLAB function, is as follows:

```
public Object[] vararginout(int nargout, double in1, String in2, Object... vararg);
```

In the `vararginout` method signature, you specify equivalent Java types for `in1` and `in2`.

The variable number of input parameters is specified in Java as `Object... vararg`.

The variable number of output parameters is specified in Java as return type `Object[]`.

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the Java method signature using the name `vararginout`. Both signatures define two inputs and two outputs.
- MATLAB Java interface supports direct conversion from Java double array to MATLAB double array and from Java string to MATLAB char array. For more information, see “Conversion of Java Types to MATLAB Types” on page A-2 and “Conversion of MATLAB Types to Java Types” on page A-4.

Marshal MATLAB Structures (Structs) in Java

Structures (or *structs*) are MATLAB arrays with elements accessed by textual field designators.

Structs consist of data containers, called *fields*. Each field stores an array of some MATLAB data type. Every field has a unique name.

A field in a structure can have a value compatible with any MATLAB data type, including a cell array or another structure.

In MATLAB, a structure is created as follows:

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

This code creates a scalar structure (S) with three fields:

```
S =
    name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

A multidimensional structure array can be created by inserting additional elements:

```
S(2).name = 'Toni Miller';
S(2).score = 91;
S(2).grade = 'A-';
```

In this case, a structure array of dimensions `(1,2)` is created. Structs with additional dimensions are also supported.

Since Java does not natively support MATLAB structures, marshaling structs between the server and client involves additional coding.

Marshaling a Struct Between Client and Server

MATLAB structures are ordered lists of name-value pairs. You represent them in Java with a class using fields consisting of the same case-sensitive names.

The Java class must also have `public get` and `set` methods defined for each field. Whether or not the class needs both `get` and `set` methods depends on whether it is being used as input or output, or both.

Following is a simple example of how a MATLAB structure can be marshaled between Java client and server.

In this example, MATLAB function `sortstudents` takes in an array of structures (see "Marshal MATLAB Structures (Structs) in Java" on page 2-25 for details).

Each element in the struct array represents different information about a student. `sortstudents` sorts the input array in ascending order by score of each student, as follows:

```
function sorted = sortstudents(unsorted)
% Receive a vector of students as input
% Get scores of all the students
scores = {unsorted.score};
% Convert the cell array containing scores into a numeric array or doubles
scores = cell2mat(scores);
% Sort the scores array
[s i] = sort(scores);
% Sort the students array based on the sorted scores array
sorted = unsorted(i);
```

Note: Even though this example only uses the `scores` field of the input structure, you can also work with `name` and `grade` fields in a similar manner.

You compile `sortstudents` into a deployable archive (`scoresorter.ctf`) using the Archive Compiler (see "Compile a Deployable Archive with the Production Server Compiler App" for details) and make it available on the server at `http://localhost:9910/scoresorter` for access by the Java client (see "Share the Deployable Archive").

Before defining the Java interface required by the client, define the MATLAB structure, `Student`, using a Java class.

`Student` declares the fields `name`, `score` and `grade` with appropriate types. It also contains `public get` and `set` functions to access these fields.

Java Class Student

```
public class Student{

    private String name;
    private int score;
    private String grade;

    public Student(){
```

```
    }

    public Student(String name, int score, String grade){
        this.name = name;
        this.score = score;
        this.grade = grade;
    }

    public String getName(){
        return name;
    }

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

    public int getScore(){
        return score;
    }

    public void setScore(int score){
        this.score = score;
    }

    public String getGrade(){
        return grade;
    }

    public void setGrade(String grade){
        this.grade = grade;
    }

    public String toString(){
        return "Student:\n\tname : " + name +
               "\n\tscore : " + score + "\n\tgrade : " + grade;
    }
}
```

Note: Note that this example uses the `toString` method for marshaling convenience. It is not required.

Next, define the Java interface `StudentSorter`, which calls method `sortstudents` and uses the `Student` class to marshal inputs and outputs.

Since you are working with a struct type, `Student` must be included in the annotation `MWStructureList` .

```
interface StudentSorter {
    @MWStructureList({Student.class})
    Student[] sortstudents(Student[] students)
            throws IOException, MATLABException;
}
```

Finally, you write the Java application (`MPSClientExample`) for the client:

1 Create `MWHttpClient` and associated proxy (using `createProxy`) as shown in "Create a Java Application That Calls a Deployed Function" on page 1-4.
2 Create an unsorted student struct array in Java that mimics the MATLAB struct in naming, number of inputs and outputs, and type validity in MATLAB. See "Java Client Coding Best Practices" on page 2-2 for more information.
3 Sort the student array and display it.

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;
import com.mathworks.mps.client.annotations.MWStructureList;

interface StudentSorter {
    @MWStructureList({Student.class})
    Student[] sortstudents(Student[] students)
            throws IOException, MATLABException;
}

public class ClientExample {

    public static void main(String[] args){

        MWClient client = new MWHttpClient();
        try{
            StudentSorter s =
               client.createProxy(new URL("http://localhost:9910/scoresorter"),
                                                          StudentSorter.class );
            Student[] students = new Student[]{new Student("Toni Miller", 90, "A"),
                                               new Student("Ed Plum",     80, "B+"),
                                               new Student("Mark Jones",  85, "A-")};
            Student[] sorted = s.sortstudents(students);
            System.out.println("Student list sorted in the
                            ascending order of scores : ");
```

```
            for(Student st:sorted){
                System.out.println(st);
            }
        }catch(IOException ex){
            System.out.println(ex);
        }catch(MATLABException ex){
            System.out.println(ex);
        }finally{
            client.close();
        }
    }
}
```

Map Java Field Names to MATLAB Field Names

Java classes that represent MATLAB structures use the Java Beans `Introspector` class (http://docs.oracle.com/javase/6/docs/api/java/beans/Introspector.html) to map properties to fields and its default naming conventions are used.

This means that by default its `decapitalize()` method is used. This maps the first letter of a Java field into a lower case letter. By default, it is not possible to define a Java field which will map to a MATLAB field which starts with an upper case.

You can override this behavior by implementing a `BeanInfo` class with a custom `getPropertyDescriptors()` method. For example:

```
import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;
public class StudentBeanInfo extends SimpleBeanInfo
{
  @Override
  public PropertyDescriptor[] getPropertyDescriptors()
  {
  PropertyDescriptor[] props = new  PropertyDescriptor[3];
  try
  {
    // name uses default naming conventions so we do not need to explicitly specify the
    props[0] = new PropertyDescriptor("name",MyStruct.class);
    // score uses default naming conventions so we do not need to explicitly specify th
    props[1] = new PropertyDescriptor("score",MyStruct.class);
    // Grade uses a custom naming convention so we do need to explicitly specify the ac
    props[1] = new PropertyDescriptor("Grade",MyStruct.class,"getGrade","setGrade");
    return props;
  }
  catch (IntrospectionException e)
```

```
  {
    e.printStackTrace();
  }

  return null;
  }
}
```

Defining MATLAB Structures Only Used as Inputs

When defining Java structs as inputs, follow these guidelines:

- Ensure that the fields in the Java class match the field names in the MATLAB struct *exactly*. The field names are case sensitive.
- Use `public get` methods on the fields in the Java class. Whether or not the class needs both `get` and `set` methods for the fields depends on whether it is being used as input or output or both. In this example, note that when `student` is passed as an input to method `sortstudents`, only the `get` methods for its fields are used by the data marshaling algorithm.

As a result, if a Java class is defined for a MATLAB structure that is only used as an input value, the `set` methods are not required. This version of the `Student` class only represents input values:

```
public class Student{

    private String name;
    private int score;
    private String grade;

    public Student(String name, int score, String grade){
        this.name = name;
        this.score = score;
        this.grade = grade;
    }

    public String getName(){
        return name;
    }

    public int getScore(){
        return score;
    }
```

```
    public String getGrade(){
        return grade;
    }
}
```

Defining MATLAB Structures Only Used as an Output

When defining Java structs as outputs, follow these guidelines:

- Ensure that the fields in the Java class match the field names in the MATLAB struct *exactly*. The field names are case sensitive.
- Create a new instance of the Java class using the structure received from MATLAB. Do so by using `set` methods or `@ConstructorProperties` annotation provided by Java. `get` methods are not required for a Java class when defining output-only MATLAB structures.

An output-only `Student` class using `set` methods follows:

Java Class Student with Struct as Output

```
public class Student{

    private String name;
    private int score;
    private String grade;

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

    public void setScore(int score){
        this.score = score;
    }

    public void setGrade(String grade){
        this.grade = grade;
    }
}
```

An output-only `Student` class using `@ConstructorProperties` follows:

Defining MATLAB structures for output using @ConstructorProperties annotation

```
public class Student{

    private String name;
    private int score;
    private String grade;

    @ConstructorProperties({"name","score","grade"})
    public Student(String n, int s, String g){
        this.name = n;
        this.score = s;
        this.grade = g;
    }
}
```

Note: If both `set` methods and `@ConstructorProperties` annotation are provided, `set` methods take precedence over `@ConstructorProperties` annotation.

Defining MATLAB Structures Used as Both Inputs and Outputs

If the `Student` class is used as both an input and output, you need to provide `get` methods to perform marshaling to MATLAB. For marshaling from MATLAB, use `set` methods or `@ConstructorProperties` annotation.

Data Conversion with Java and MATLAB Types

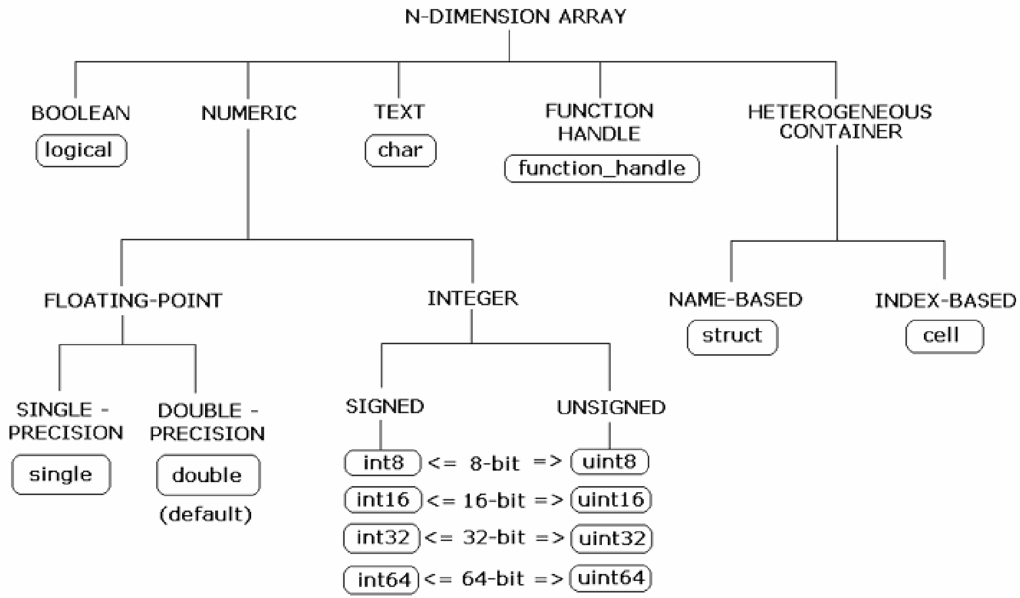
Working with MATLAB Data Types

There are many data types that you can work with in MATLAB. Each of these data types is in the form of a matrix or array. You can build matrices and arrays of floating-point and integer data, characters and strings, and logical true and false states. Structures and cell arrays provide a way to store dissimilar types of data in the same array.

All of the fundamental MATLAB classes are circled in the diagram Fundamental MATLAB Data Types.

The Java client follows *Java-MATLAB-Interface* (JMI) rules for data marshaling. It expands those rules for scalar Java boxed types, allowing auto-boxing and un-boxing, which JMI does not support.

Note: Function Handles are not supported by MATLAB Production Server.



Fundamental MATLAB Data Types

The expected conversion results for Java to MATLAB types are listed in "Conversion of Java Types to MATLAB Types" on page A-2. The expected conversion results for MATLAB to Java types are listed in "Conversion of MATLAB Types to Java Types" on page A-4.

Scalar Numeric Type Coercion

Scalar numeric MATLAB types can be assigned to multiple Java numeric types as long as there is no loss of data or precision.

The main exception to this rule is that MATLAB `double` scalar data can be mapped into any Java numeric type. Because `double` is the default numeric type in MATLAB, this exception provides more flexibility to the users of MATLAB Production Server Java client API.

MATLAB to Java Numeric Type Compatibility describes the type compatibility for scalar numeric coercion.

MATLAB to Java Numeric Type Compatibility

MATLAB Type	Java Types
`uint8`	`short`, `int`, `long`, `float`, `double`
`int8`	`short`, `int`, `long`, `float`, `double`
`uint16`	`int`, `long`, `float`, `double`
`int16`	`int`, `long`, `float`, `double`
`uint32`	`long`, `float`, `double`
`int32`	`long`, `float`, `double`
`uint64`	`float`, `double`
`int64`	`float`, `double`
`single`	`double`
`double`	`byte`, `short`, `int`, `long`, `float`

Dimensionality in Java and MATLAB Data Types

In MATLAB, dimensionality is an attribute of the fundamental types and does not add to the number of types as it does in Java.

In Java, `double`, `double[]` and `double[][][]` are three different data types. In MATLAB, there is only a `double` data type and possibly a scalar instance, a vector instance, or a multi-dimensional instance.

Java Signature	Value Returned from MATLAB
`double[][][] foo()`	`ones(1,2,3)`

Dimension Coercion

How you define your MATLAB function and corresponding Java method signature determines if your output data will be coerced, using padding or truncation.

This coercion is automatically performed for you. This section describes the rules followed for padding and truncation.

Padding

When a Java method's return type has more dimensions than MATLAB's, MATLAB's dimensions are be padded with ones (1s) to match the required number of output dimensions in Java.

You, as a developer, do not have to do anything to pad dimensions.

The following tables provide examples of how padding is performed for you:

How MATLAB Pads Your Java Method Return Type

When Dimensions in MATLAB are:	And Dimensions in Java are:	This Type in Java:	Returns this Type in MATLAB:
`size(a)` is`[2,3]`	Array will be returned as size `2,3,1,1`	`double [][][][] foo()`	`function a = foo a = ones(2,3);`

Padding Dimensions in MATLAB and Java Data Conversion

MATLAB Array Dimensions	Declared Output Java Type	Output Java Dimensions
`2 x 3`	`double[][][]`	`2 x 3 x 1`
`2 x 3`	`double[][][][]`	`2 x 3 x 1 x 1`

Truncation

When a Java method's return type has fewer dimensions than MATLAB's, MATLAB's dimensions are truncated to match the required number of output dimensions in Java. This is only possible when extra dimensions for MATLAB array have values of ones (1s) only.

To compute appropriate number of dimensions in Java, excess ones are truncated, in this order:

1. From the end of the array
2. From the array's beginning
3. From the middle of the array (scanning front-to-back).

You, as a developer, do not have to do anything to truncate dimensions.

The following tables provide examples of how truncation is performed for you:

How MATLAB Truncates Your Java Method Return Type

When Dimensions in MATLAB are:	And Dimensions in Java are:	This Type in Java:	Returns this Type in MATLAB
`size(a)` is `[1,2,1,1,3,1]`	Array will be returned as size `2,3`	`double [][] foo()`	`function a = foo a = ones(1,2,1,1,3,1);`

Following are some examples of dimension shortening using the `double numeric` type:

Truncating Dimensions in MATLAB and Java Data Conversion

MATLAB Array Dimensions	Declared Output Java Type	Output Java Dimensions
`1 x 1`	`double`	`0`
`2 x 1`	`double[]`	`2`
`1 x 2`	`double[]`	`2`
`2 x 3 x 1`	`double[][]`	`2 x 3`
`1 x 3 x 4`	`double[][]`	`3 x 4`
`1 x 3 x 4 x 1 x 1`	`double[][][]`	`1 x 3 x 4`
`1 x 3 x 1 x 1 x 2 x 1 x 4 x 1`	`double[][][][]`	`3 x 2 x 1 x 4`

Empty (Zero) Dimensions

Passing arrays of zero (`0`) dimensions (sometimes called *empties*) results in an empty matrix from MATLAB.

Java Signature	Value Returned from MATLAB
`double[] foo()`	`[]`

Passing Java Empties to MATLAB

When a `null` is passed from Java to MATLAB, it will always be marshaled into `[]` in MATLAB as a zero by zero (`0` x `0`) double. This is independent of the declared input type used in Java. For example, all the following methods can accept `null` as an input value:

```
void foo(String input);
void foo(double[] input);
void foo(double[][] input);
void foo(Double input);
```

And in MATLAB, `null` will be received as:

```
[] i.e. 0x0 double
```

Passing MATLAB Empties to Java

An empty array in MATLAB has at least one zero (`0`) assigned in at least one dimension. For `function a = foo`, for example, any one of the following values is acceptable:

```
 a = [];
 a = ones(0);
 a = ones(0,0);
 a = ones(1,2,0,3);
```

Empty MATLAB data will be returned to Java as `null` for all the above cases.

For example, in Java, the following signatures return `null` when a MATLAB function returns an empty array:

```
double[] foo();
double[][] foo();
Double foo();
```

However, when MATLAB returns an empty array and the return type in Java is a `scalar` primitive (as with `double foo();`, for example) an exception is thrown . :

```
IllegalArgumentException
("An empty MATLAB array cannot be represented by a
  primitive scalar Java type")
```

Boxed Types

Boxed Types are used to wrap opaque C structures.

Java client will perform primitive to boxed type conversion if boxed types are used as return types in the Java method signature.

Java Signature	Value Returned from MATLAB
Double foo()	1.0

For example, the following method signatures work interchangeably:

```
double[] foo();         Double[] foo();
double[][][] foo();     Double[][][] foo();
```

Signed and Unsigned Types in Java and MATLAB Data Types

Numeric classes in MATLAB include signed and unsigned integers. Java does not have unsigned types.

Data Conversion Rules

Conversion of Java Types to MATLAB Types

Value Passed to Java Method is:	Input type Received by MATLAB is:	Dimension of Data in MATLAB is:
`java.lang.Byte`, `byte`	`int8`	{1,1}
`byte[]` *data*		{1, *data.length*}
`java.lang.Short` `short`	`int16`	{1,1}
`short[]` *data*		{1, *data.length*}
`java.lang.Integer`, `int`	`int32`	{1,1}
`int[]` *data*		{1, *data.length*}
`java.lang.Long`, `long`	`int64`	{1,1}
`long[]` *data*		{1, *data.length*}
`java.lang.Float`, `float`	`single`	{1,1}
`float[]` *data*		{1, *data.length*}
`java.lang.Double`, `double`	`double`	{1,1}
`double[]` *data*		{1, *data.length*}
`java.lang.Boolean`, `boolean`	`logical`	{1,1}
`boolean[]` *data*		{1, *data.length*}
`java.lang.Character`, `char`	`char`	{1,1}
`char[]` *data*		{1, *data.length*}
`java.lang.String` *data*		{1, *data.length*()}
`java.lang.String[]` *data*	`cell`	{1, *data.length*}
`java.lang.Object[]` *data*		{1, *data.length*}

Value Passed to Java Method is:	Input type Received by MATLAB is:	Dimension of Data in MATLAB is:
`T[]` *`data`* $_1$	MATLAB type for `T` $_1$	{ *`data.length`*, *`dimensions`*(`T`[0]) }, if `T` is an array
		{ 1, *`data.length`*}, if `T` is not an array

Where `T` represents any supported MATLAB type. If T is an array type, then all elements of data must have exactly the same length

Conversion of MATLAB Types to Java Types

When MATLAB Returns:	Dimension of Data in MATLAB is:	MATLAB Data Converts To Java Type:
int8, uint8	{1,1}	byte, java.lang.Byte
	{1,*n*} , {*n*,1}	byte[*n*], java.lang.Byte[*n*]
	{*m*,*n*,*p*,...}	byte[*m*][*n*][*p*]... , java.lang.Byte[*m*][*n*][*p*]...
int16, uint16	{1,1}	short, java.lang.Short
	{1,*n*} , {*n*,1}	short[*n*], java.lang.Short[*n*]
	{*m*,*n*,*p*,...}	short[*m*][*n*][*p*]... , java.lang.Short[*m*][*n*][*p*]...
int32, uint32	{1,1}	int, java.lang.Integer
	{1,*n*} , {*n*,1}	int[*n*], java.lang.Integer[*n*]
	{*m*,*n*,*p*,...}	int[*m*][*n*][*p*]... , java.lang.Integer[*m*][*n*][*p*]...
int64, uint64	{1,1}	long, java.lang.Long
	{1,*n*} , {*n*,1}	long[*n*], java.lang.Long[*n*]
	{*m*,*n*,*p*,...}	long[*m*][*n*][*p*]... , java.lang.Long[*m*][*n*][*p*]...
single	{1,1}	float, java.lang.Float
	{1,*n*} , {*n*,1}	float[*n*], java.lang.Float[*n*]
	{*m*,*n*,*p*,...}	float[*m*][*n*][*p*]... , java.lang.Float[*m*][*n*][*p*]...
double	{1,1}	double, java.lang.Double
	{1,*n*} , {*n*,1}	double[*n*], java.lang.Double[*n*]
	{*m*,*n*,*p*,...}	double[*m*][*n*][*p*]... ,

When MATLAB Returns:	Dimension of Data in MATLAB is:	MATLAB Data Converts To Java Type:
		java.lang.Double[*m*][*n*][*p*]...
logical	{1,1}	boolean, java.lang.Boolean
	{1,*n*} , {*n*,1}	boolean[*n*], java.lang.Boolean[*n*]
	{*m*,*n*,*p*,...}	boolean[*m*][*n*][*p*]... , java.lang.Boolean[*m*][*n*][*p*]...
char	{1,1}	char, java.lang.Character
	{1,*n*} , {*n*,1}	java.lang.String
	{*m*,*n*,*p*,...}	char[*m*][*n*][*p*]... , java.lang.Character[*m*][*n*][*p*]...
cell (containing only strings)	{1,1}	java.lang.String
	{1,*n*} , {*n*,1}	java.lang.String[*n*]
	{*m*,*n*,*p*,...}	java.lang.String[*m*][*n*][*p*]...
cell (containing multiple types)	{1,1}	java.lang.Object
	{1,*n*} , {*n*,1}	java.lang.Object[*n*]
	{*m*,*n*,*p*,...}	java.lang.Object[*m*][*n*][*p*]...