MATLAB® Builder™ JA 2
User's Guide

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science
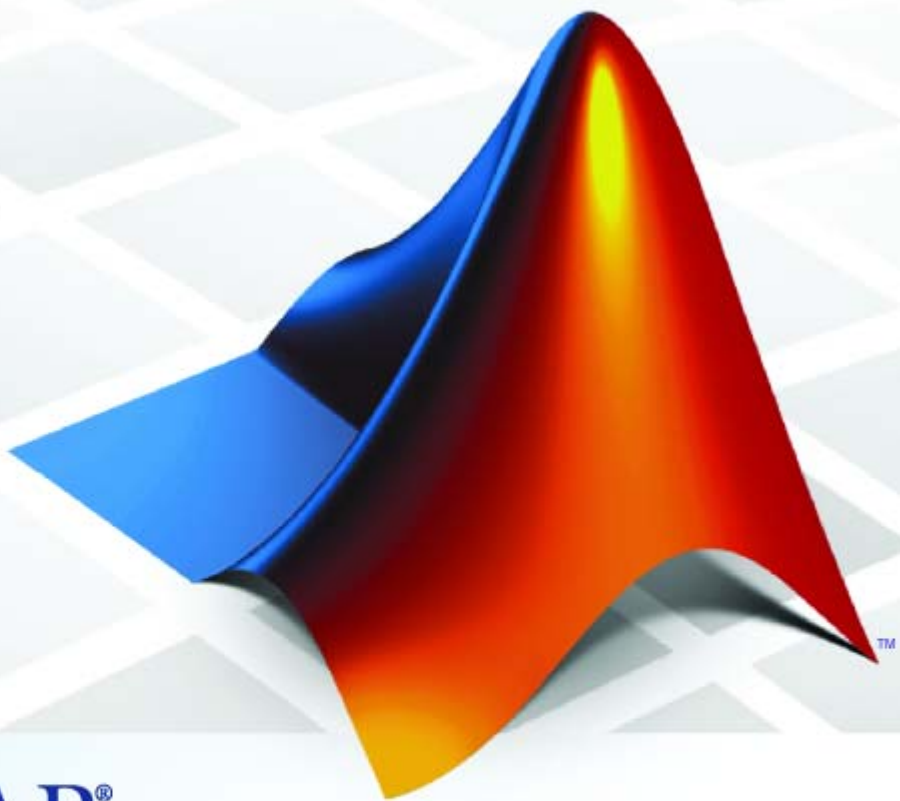
How to Contact The MathWorks

	www.mathworks.com	Web
	comp.soft-sys.matlab	Newsgroup
	www.mathworks.com/contact_TS.html	Technical Support
@	suggest@mathworks.com	Product enhancement suggestions
	bugs@mathworks.com	Bug reports
	doc@mathworks.com	Documentation error reports
	service@mathworks.com	Order status, license renewals, passcodes
	info@mathworks.com	Sales, pricing, and general information

☎ 508-647-7000 (Phone)

📠 508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Builder™ JA User's Guide

© COPYRIGHT 2006–2008 by The MathWorks, Inc.

Trademarks

Patents

Revision History

Contents

Getting Started

1

Concepts

2

<div align="right">

Programming

</div>

3

Sample Java Applications

4

Deploying a Java Component Over the Web

5

Reference Information for Java

6

Function Reference

7

The MATLAB Application Deployment Web
Example Guide

8

Examples

A

Index

1

Getting Started

- "Product Overview" on page 1-2
- "Before You Begin" on page 1-3
- "Deploying a Component with the Magic Square Example" on page 1-9
- "Next Steps" on page 1-29

Product Overview

In this section...
"MATLAB® Compiler Extension" on page 1-2
"How the MATLAB® Compiler and MATLAB® Builder JA Products Work Together" on page 1-2

MATLAB Compiler Extension

The MATLAB® Builder™ JA product is an extension to the MATLAB® Compiler™ product. Using the MATLAB Builder JA product, you can wrap M-code functions from the MATLAB® product into one or more Java™ *classes*. A Java class is a portion of Java code that houses a Java *method*, or a unit of code that performs some action. You compile Java classes into Java *components*, self-contained modules that run Java applications.

When deployed, each MATLAB function is encapsulated as a method of a Java class and can be invoked from within a Java application. When your Java packages are created, you also have the option of including the MATLAB Compiler Runtime (MCR), allowing users to run and deploy their new applications on computers that do not have MATLAB installed.

How the MATLAB Compiler and MATLAB Builder JA Products Work Together

The MATLAB Compiler product can compile M-files, MEX-files, MATLAB objects, or other MATLAB code. The MATLAB Builder JA product supports all the features of MATLAB, and adds support for Java classes, Java *objects* (instances of a class), and methods. Using these products together, you can generate the following:

- Standalone applications on UNIX®, Windows®, and Macintosh® platforms

- C and C++ shared libraries (dynamically linked libraries, or DLLs, on Microsoft® Windows)

- Enterprise Java applications for use on any Java compatible platform

Before You Begin

In this section...
"Your Role in the Java Application Deployment Process" on page 1-3
"What You Need to Know" on page 1-5
"Required Products" on page 1-6
"Configuring Your Environment" on page 1-6

Your Role in the Java Application Deployment Process

Depending on the size of your organization, you may play one role, or many, in the process of successfully deploying a Java application.

For example, your job may be to analyze user requirements and satisfy them by writing a program in M-code. Or, your role may be to implement the infrastructure needed to successfully deploy a Java application to the Web. In smaller installations, you may find one person responsible for performing tasks associated with multiple roles. The table Application Deployment Roles, Tasks, and References on page 1-4 describes some of the different roles, or jobs, that MATLAB Builder JA users typically perform and which tasks they would most likely perform when running "Deploying a Component with the Magic Square Example" on page 1-9.

Application Deployment Roles, Tasks, and References

Role	Tasks	References
MATLAB Programmer	• Understand end-user business requirements and the mathematical models needed to support them. • Write M-code. • Build an executable component with MATLAB tools (usually with support from a Java programmer). • Package the component for distribution to end users. • Pass the packaged component to the Java programmer for rollout and further integration into the end-user environment.	"Starting the MATLAB® Builder JA Product" on page 1-11 "Copying the Example Files" on page 1-11 "Testing the M-File You Want to Deploy" on page 1-12 "Creating a Java Component" on page 1-13 "Selecting Files to Package with Your Component" on page 1-16 "Running the Packaging Tool" on page 1-19 "Copy the Package You Created" on page 1-19

Application Deployment Roles, Tasks, and References (Continued)

Role	Tasks	References
Java Programmer	- Write Java code to execute the Java package built by the MATLAB programmer. - Roll out the packaged component and integrate it into the end-user environment. - Use the component in enterprise Java applications, adding and modifying code as needed. - Address data conversion issues that may be encountered, according to the end user's specifications. - Ensure the final Java application executes reliably in the end user's environment.	"Gathering Files Needed for Deployment" on page 1-20 "Testing the Java Component in a Java Application" on page 1-21 "Distributing the Component to End Users" on page 1-26 "Integrating Java Classes Generated by MATLAB into a Java Application" on page 1-26 "Calling Class Methods from Java" on page 1-27 "Handle Data Conversion as Needed" on page 1-28 "Build and Test" on page 1-28
External user	Executes the solution created by MATLAB and Java programmers.	Run the deployed application (outside the scope of this document).

What You Need to Know

The following knowledge is assumed when you use the MATLAB Builder JA product:

- If your job function is MATLAB programmer, the following is required:
  - A basic knowledge of MATLAB, and how to work with cell arrays and structures
- If your job function is Java programmer, the following is required:
  - Exposure to the Java programming language
  - Object-oriented programming concepts

Required Products

The following products are required to be installed to run the example described in this chapter:

- MATLAB
- MATLAB Compiler
- MATLAB Builder JA

If you currently have any of these products running, shut them down. You will be instructed to start them as needed in "Deploying a Component with the Magic Square Example" on page 1-9.

Configuring Your Environment

Configure your environment to work with the examples. Consult your system administrator or Java programmer before performing these tasks—some may be unnecessary. Your administrator or programmer is often the best resource for verifying, installing, or customizing your Java environment.

Verifying Your Java Environment

You may already be running a compatible version of Java. To find out if you have Java installed on your computer, and if it is compatible with the MATLAB Builder JA product:

1 Open a system command prompt.

2 Enter the command `java -version`. If Java is installed, the result looks like this:

```
java version "version_number"
Java(TM) 2 Runtime Environment, Standard Edition
(build version_number.build_number
Java HotSpot(TM) Client VM (build version_number.build_number,
mixed mode)
```

3 Enter the command `javac -version`.

> **Note** Alternately, you may have to enter `%JAVA_HOME%\bin\javac -version` if you have the user environment variable `JAVA_HOME` defined but the JDK file path is not defined to the system environment variable `PATH`. For information on locating environment variables, see "Setting Up Your Java Environment" on page 1-7.

If `javac` is installed, the results should look like this:

```
javac version_number
```

If `java` and `javac` are installed and the version numbers are at least 1.5, go to "Deploying a Component with the Magic Square Example" on page 1-9. If not, go to "Setting Up Your Java Environment" on page 1-7.

Setting Up Your Java Environment

1 Download and install the Java Developer's Kit (JDK) from Sun Microsystems™, Inc. if you do not yet have it installed.

The JDK is a collection of Java classes, run-time environment, compiler, debugger, and usually source code, for a version of Java. The contents of the JDK collectively make up a Java development environment.

The JDK includes the Java Runtime Environment (JRE), a collection of compiled classes that makes up the *Java virtual machine*, a standalone executor of the Java language, on a specific platform. Ensure your Java Runtime Environment and JDK are compatible with the Sun Microsystems JDK Version 1.6.0 before proceeding to the next step.

2 Set the environment variable `JAVA_HOME`. This tells Java where to find your installed JDK.

- On Windows platforms:

  1 Right-click the **My Computer** icon and select **Properties**.

  2 Click the **Advanced** tab.

  3 Click **Environment Variables**.

  4 In the User Variables area, click **New**.

  5 In the New User Variable dialog box, enter JAVA_HOME for **Variable name**. Enter the absolute path name where your JDK is installed for **Variable value**. Here is the value of a typical JAVA_HOME environment variable:

  ```
  C:\Program Files\Java\JDK1.6.0_03
  ```

  > **Note** If JAVA_HOME already exists, select it and click **Edit**. Enter the path name where your JDK is installed.

  6 Click **OK** to accept changes.

- On UNIX platforms:

  1 Open a command prompt.

  2 Set JAVA_HOME as follows:

  ```
  set JAVA_HOME=JDK_pathname
  ```

Deploying a Component with the *Magic Square Example*

In this section...
"About this Example" on page 1-9
"Magic Square Example: MATLAB Programmer Tasks" on page 1-9
"Magic Square Example: Java Programmer Tasks" on page 1-19

About this Example

In this section, you will step through an example of how a simple M-code function can be transformed into a deployable MATLAB Builder JA component.

The Magic Square example shows you how to create a Java component named magicsquare which contains the magic class, a .jar file, and other files needed to deploy your application.

The class wraps a MATLAB function, makesqr, which computes a magic square. A magic square is a matrix containing any number of rows. These rows, when added horizontally and vertically, equate to the same value. MATLAB contains a function, magic, that can create magic squares of any dimension. In this example, you will work with that function.

Note The examples here use the Windows deploytool GUI, a graphical front-end interface to MATLAB Compiler software. For information about how to perform these tasks using the command-line interface to MATLAB Compiler software, see the mcc reference page.

Magic Square Example: MATLAB Programmer Tasks

The following tasks are usually performed by the MATLAB Programmer.

Key Tasks for the MATLAB Programmer

Task	Reference
Start the product.	"Starting the MATLAB® Builder JA Product" on page 1-11
Prepare to run the example by copying the MATLAB example files into a work directory.	"Copying the Example Files" on page 1-11
Test the M-code to ensure it is suitable for deployment.	"Testing the M-File You Want to Deploy" on page 1-12
Create a Java package (encapsulating your M-code in a Java class) by running the **Build** function in deploytool.	"Creating a Java Component" on page 1-13
Prepare to run the Packaging Tool by determining what additional files should be included with the deployed component.	"Selecting Files to Package with Your Component" on page 1-16
Run the Packaging Tool to bundle your Java component with the additional files you selected.	"Running the Packaging Tool" on page 1-19
Copy the output from the Packaging Tool (the distrib directory).	"Copy the Package You Created" on page 1-19

The Magic Square example shows you how to create a Java component (magicsquare), which contains the magic class, a .jar file (which includes the .ctf archive described in "How Does the MATLAB® Builder JA Product Use JAR Files?" on page 2-4), and other files needed to deploy your application. The class encapsulates a MATLAB function, makesqr, which computes a magic square.

The client Java application, getmagic.java converts the array returned by makesqr to a native array and displays it on the screen. When you run the getmagic application from the command line, you can pass the dimension for the magic square as a command-line argument.

> **Note** The examples for the MATLAB Builder JA product are in
> *matlabroot*\toolbox\javabuilder\Examples. In most examples, Windows
> syntax is featured (backslashes, instead of forward slashes). This example
> assumes the work directory is on drive D:.

Starting the MATLAB Builder JA Product

The MATLAB Builder JA product is accessed through the Deployment Tool
GUI (deploytool) or through the mcc function of the MATLAB Compiler
product. deploytool is the GUI front end for mcc, the command that executes
MATLAB Compiler software.

This tutorial will use deploytool. If you want to use mcc, see the mcc
reference page for complete reference information.

To start this product:

1 Start MATLAB.

2 Type deploytool at the MATLAB command prompt. The deploytool
GUI opens.

3 Verify that MATLAB is reading the correct value of JAVA_HOME.

  a At the MATLAB command prompt, type getenv JAVA_HOME.

  b The response from MATLAB should be the path name you set to
  JAVA_HOME in "Configuring Your Environment" on page 1-6. If not,
  ensure the JDK that MATLAB is pointing to will be compatible to
  run this example (at least Java Version 1.5). Consult your system
  administrator if you are unsure.

Copying the Example Files

Prepare to run the example by copying needed files into your work area as
follows:

1 Navigate to
matlabroot\toolbox\javabuilder\Examples\MagicSquareExample.
matlabroot is the MATLAB root directory (where MATLAB is

installed). To find the value of this variable on your system, type *matlabroot* at a MATLAB command prompt.

2 Copy the MagicSquareExample folder to a work area, for example, D:\javabuilder_examples. Avoid using spaces in your directory names, if possible.

3 Rename the subdirectory MagicSquareExample to magic_square. The example files should now reside in D:\javabuilder_examples\magic_square.

4 Using a system command prompt, navigate to D:\javabuilder_examples\magic_square by switching to the D: drive and entering cd \javabuilder_examples\magic_square.

Testing the M-File You Want to Deploy

Normally you would first create the M-file you want to deploy. In this example, you will test a precreated M-file (magicsqr.m) containing the predefined MATLAB function magic.

1 Using MATLAB, locate the magicsqr.m file at D:\javabuilder_examples\magic_square\MagicDemoComp. The contents of the file are as follows:

```
function y = makesqr(x)
%MAKESQR Magic square of size x.
%   Y = MAKESQR(X) returns a magic square of size x.
%   This file is used as an example for the MATLAB
%   Builder JA product.

%   Copyright 2001-2008 The MathWorks, Inc.

y = magic(x);
```

2 In order to run makesqr, you must ensure that MATLAB can find it. Use the **File > Set Path** option in MATLAB to add the D:\javabuilder_examples\magic_square\MagicDemoComp directory to the MATLAB search path.

3 At the MATLAB command prompt, enter `makesqr(5)` and view the results. The output should appear as follows:

```
17 24  1  8 15
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
```

Creating a Java Component

You create a Java component by using the Deployment Tool GUI to build a Java class that wraps around the sample M-code discussed in "Testing the M-File You Want to Deploy" on page 1-12.

1 Create a new deployment project. A project is a collection of files that are bundled together under a project file name (`.prj` file) for your convenience by the Deployment Tool. Having all the files in one place makes it easy for you to build and run an application many times — effectively testing it — until it is ready for deployment.

a Click the Open button in the Deployment Tool toolbar.

b In the navigation pane (on the left), select **MATLAB Builder JA** as the product you want to use to create the deployment project.

c From the right pane, select **Java Package**.
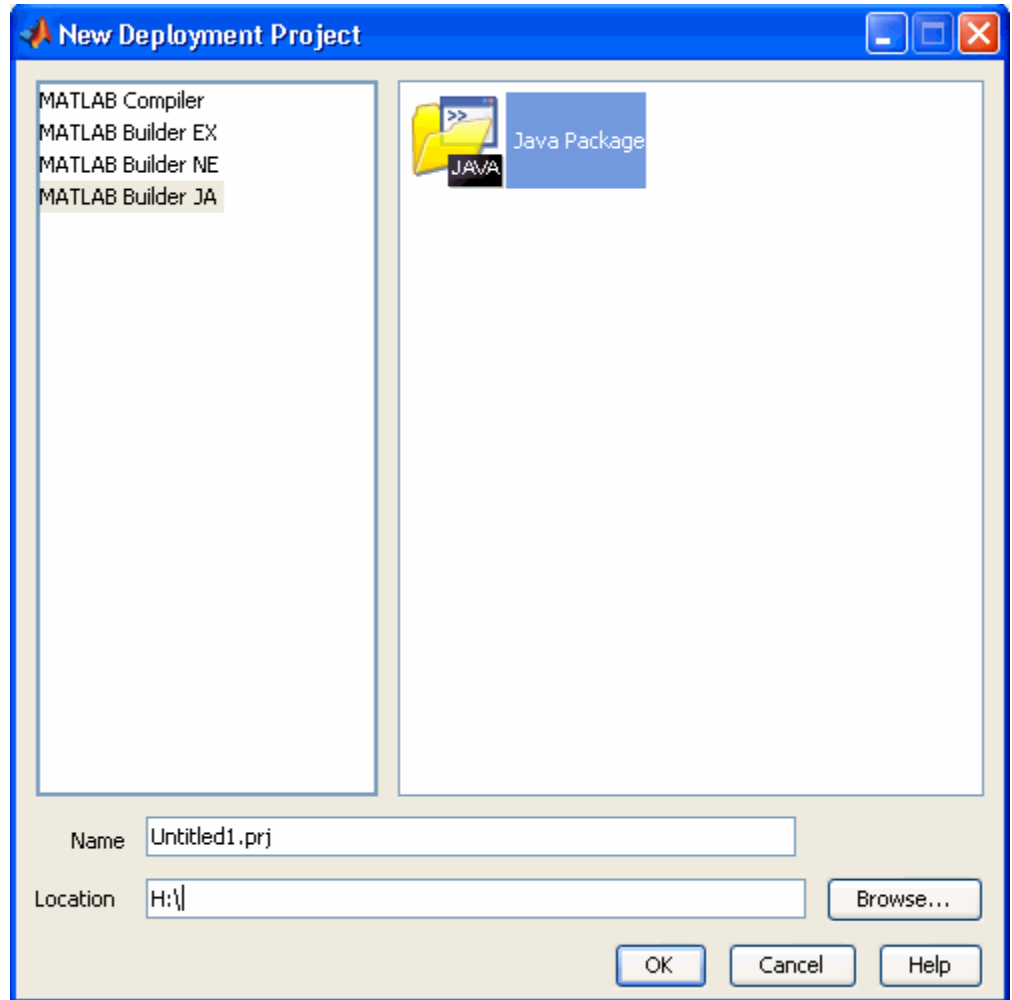
d Click **Browse** to select the location for your project. In this case, use
`D:\javabuilder_examples\magic_square`.

e Enter `magicsquare` as the project name and click **OK**. By default, the
project name is also the package name as well as the name of the
subdirectory created under `D:\javabuilder_examples\magic_square`.

2 The MATLAB Builder JA product assigns default names to the classes it
creates, based on project name. Because the example program uses the

name `magicsquare` for the class by default, you need to rename the class as follows:

a Right-click the folder under the project folder (the **class** folder), which represents the Java class you are going to create (currently named `magicsquareclass`), and select **Rename Class**.

b Enter `magic` as the new name of the class and press **Enter**.

3 In the **Deployment Tool** pane, ensure that the **Generate Verbose Output** option is selected. This option enables you to receive detailed error messages in the Deployment Tool Output window, which appears when your class is built.

Note There are many other options you can choose to customize your output from `deploytool`, such as whether or not to include the MCR. Click **Settings** and view the `deploytool` Help for details.

---

4 Add the example M-file to the project:

a In MATLAB, navigate to `D:\javabuilder_examples\magic_square\MagicDemoComp`.

b Add the `makesqr.m` file in the `MagicDemoComp` directory to the project by dragging this file to the renamed `magic` folder in the **Deployment Tool** pane.

c Save the project by clicking the Save button ![save] in the Deployment Tool toolbar.

5 Build the project, creating your initial Java package.

Build the project by clicking the Build button ![build] in the Deployment Tool toolbar.

As you build, informational messages, warnings, and errors are generated in the Deployment Tool Output window.

The MATLAB Builder JA product creates two directories, `src` and `distrib`, within the project directory. The paths to these directories are defined

in the Deployment Project Settings dialog box. A copy of the build log is placed in the `src` directory.

- The `src` directory contains the generated Java source (`magic.class`).

- The `distrib` directory contains a Java archive (`magicsquare.jar`) file that will be distributed with your deployed application, including the MCR if you chose to include it.

Selecting Files to Package with Your Component

Bundling the Java component with additional files into a JAR file that can be distributed to users is called packaging. You will perform this step using the packaging function of `deploytool`. If you are creating a shared component and want to include additional code with the component, you must perform this step.

Note "Packaging" a component and a "Java package" are different concepts.

"Packaging" in this context refers only to the act of using MATLAB Builder JA to bundle the Java component with associated files needed to successfully deploy the application. A "Java package" is a collection of Java classes and methods.

---

1 To direct the Packaging Tool to include additional files in the JAR file, select **Project > Settings** in MATLAB, then click **Packaging** in the left pane of the Deployment Project Settings dialog box and follow the instructions in the next table.

Additional File to Include for Deployment to End Users Without MATLAB

Additional File	Description	How to Include	Notes
MCR Installer	A self-contained run-time executable capable of running an instance of MATLAB on a computer that does not have MATLAB installed.	Select the option **Include MCR**.	Including the MCR can expand the size of your distributable package bundle file (and build time) dramatically. Java components created with MATLAB Builder JA are dependent on the version of MATLAB with which they were built.

Additional File to Include for Deployment to End Users Without MATLAB (Continued)

Additional File	Description	How to Include	Notes
Auto-generated Javadoc documentation	This documentation from Sun Microsystems contains information about Java classes and methods referenced in your application.	**a** Browse to the \src\doc directory in the left pane of the window in the **Additional Files** area. **b** When you have located and selected the desired file(s), click **Add**.	
readme.txt	This file contains steps and file locations often needed by end users to deploy applications.	**a** Browse to the \distrib directory in the left pane of the window in the **Additional Files** area. **b** When you have located and selected the desired file(s), click **Add**.	

2 Add any additional code or files by following the same process as above. Click **OK** when finished.

Running the Packaging Tool

1 In the Deployment Tool, click the Packaging button 🎁.
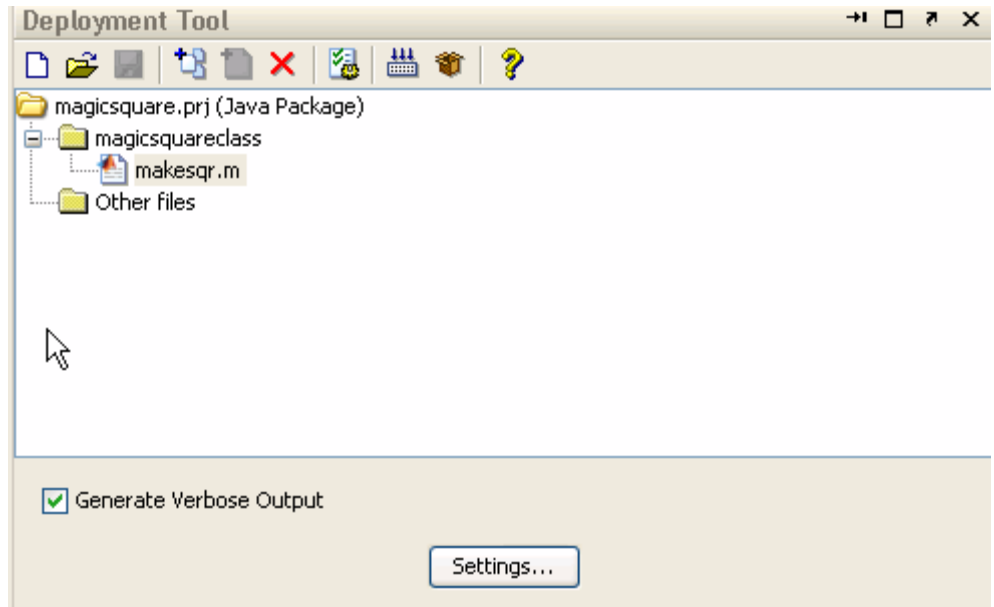
2 Verify the contents of the distrib directory contain the files you specified above. Use an archival program such as WinZip to bundle the files in distrib for enterprise-wide distribution.

Copy the Package You Created

Copy the package that you created from the distrib directory to the local directory of your choice or send them directly to the Java Programmer..

Magic Square Example: Java Programmer Tasks

The following tasks are usually performed by the Java Programmer.

Key Tasks for the Java Programmer

Task	Reference
Ensure you have the needed files from the MATLAB Programmer before proceeding.	"Gathering Files Needed for Deployment" on page 1-20
Test the Java code by using it in a Java application. Compile and run the component to ensure it produces the same results as your M-code.	"Creating a Java Component" on page 1-13
Archive and distribute the output to end users.	"Distributing the Component to End Users" on page 1-26
Import classes generated by the MATLAB Builder JA product into existing Java applications.	"Integrating Java Classes Generated by MATLAB into a Java Application" on page 1-26
Use built-in Java class methods to enhance your Java application.	"Calling Class Methods from Java" on page 1-27
Address potential data conversion issues with differing data types.	"Handle Data Conversion as Needed" on page 1-28
Verify your Java application works as expected in your end user's deployment environment.	"Build and Test" on page 1-28

Gathering Files Needed for Deployment

Before beginning, verify you have access to the following files, created by the MATLAB Programmer in "Copy the Package You Created" on page 1-19. The following files are required to deploy to users who do not have a copy of MATLAB installed:

- MCR Installer

- Javadoc documentation

- readme.txt file

See "Selecting Files to Package with Your Component" on page 1-16 for more information about these files. You will also want to communicate the location of `com.mathworks.toolbox.javabuilder` (*matlabroot*\toolbox\javabuilder\jar\javabuilder.jar). You can browse the API Javadoc for `com.mathworks.toolbox.javabuilder` from the MATLAB Help.

Testing the Java Component in a Java Application

Before deploying the created component, you need to verify that it can be used in a Java application successfully.

First, create a small Java program that uses the component created for your by the MATLAB Programmer (see "Running the Packaging Tool" on page 1-19). The example provides a sample Java program that accomplishes this (`getmagic.java` now in the directory `D:\javabuilder_examples\magic_square\MagicDemoJavaApp`).

The program imports the `magicsquare` package you created with `deploytool` and the MATLAB Builder JA package (`com.mathworks.toolbox.javabuilder`) and uses one of the MATLAB Builder JA conversion classes to convert the number passed to the program on the command line into a type that can be accepted by MATLAB, in this case a scalar double value.

The program then creates an instance of class `magic`, and calls the `makesqr` method on that object. Note how the M-file becomes a method of the Java class that encapsulates it. As explained in "Testing the M-File You Want to Deploy" on page 1-12, the `makesqr` method computes the square using the MATLAB `magic` function. The source code of `getmagic.java` follows, for your reference:

```
/* getmagic.java
 * This file is used as an example for the MATLAB
 * Builder JA product.
 *
 * Copyright 2008 The MathWorks, Inc.
 */

/* Necessary package imports */
import com.mathworks.toolbox.javabuilder.*;
```

```
import magicsquare.*;

/*
 * getmagic class computes a magic square of order N. The
 * positive integer N is passed on the command line.
 */
class getmagic
{
   public static void main(String[] args)
   {
      MWNumericArray n = null;    /* Stores input value */
      Object[] result = null;     /* Stores the result */
      magic theMagic = null;      /* Stores magic class instance */

      try
      {
         /* If no input, exit */
         if (args.length == 0)
         {
            System.out.println("Error: must input a positive
                integer");
            return;
         }

         /* Convert and print input value*/
         n = new MWNumericArray(Double.valueOf(args[0]),
                                   MWClassID.DOUBLE);

         System.out.println("Magic square of order " + n.toString());

         /* Create new magic object */
         theMagic = new magic();

         /* Compute magic square and print result */
         result = theMagic.makesqr(1, n);
         System.out.println(result[0]);
      }
      catch (Exception e)
      {
         System.out.println("Exception: " + e.toString());
```

```
        }

        finally
        {
            /* Free native resources */
            MWArray.disposeArray(n);
            MWArray.disposeArray(result);
            if (theMagic != null)
                theMagic.dispose();
        }
    }
}
```

Ensure your current working directory is set to
`D:\javabuilder_examples\magic_square` as noted previously in this
example. Then, do the following:

1 Compile the Java component with the Java compiler, `javac`. At the system
command prompt, enter one of the following commands. When entering
these commands, ensure single spaces are inserted at the end of each
line below. On Windows systems, the semicolon (`;`) is a concatenation
character. On UNIX systems, the colon (`:`) is a concatenation character.

- On Windows platforms:

```
%JAVA_HOME%\bin\javac -classpath
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;.\magicsquare\distrib\magicsquare.jar
.\MagicDemoJavaApp\getmagic.java
```

- On UNIX platforms:

```
$JAVA_HOME/bin/javac -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:./magicsquare/distrib/magicsquare.jar
./MagicDemoJavaApp/getmagic.java
```

Inspect the syntax of the `javac` compile command on Windows platforms:

```
%JAVA_HOME%\bin\javac -classpath
 matlabroot\toolbox\javabuilder\jar\javabuilder.jar;.\magicsquare\distrib\magicsquare.jar
.\MagicDemoJavaApp\getmagic.java
```

1-23

The components of this command are:

- `%JAVA_HOME%/bin/javac` — Using this command invokes the Java compiler explicitly from the version of Java you set with `JAVA_HOME` (see "Configuring Your Environment" on page 1-6).

Note `%JAVA_HOME%` is Windows syntax and `$JAVA_HOME` is UNIX syntax.

- `-classpath` — Using this argument allows Java to access the packages and other files you need to compile your component.
- *matlabroot*`\toolbox\javabuilder\jar\javabuilder.jar` — The location of the MATLAB Builder JA package file (`com.mathworks.toolbox.javabuilder`).
- `.\magicsquare\distrib\magicsquare.jar` — The location of the `magicsquare` package file you created with `deploytool`.
- `.\MagicDemoJavaApp\getmagic.java` — The location of the `getmagic.java` source file.

2 When you run `getmagic`, you pass an input argument to Java representing the dimension for the magic square. In this example, the value for the dimension is 5. Run `getmagic` by entering one of the following `java` commands at the system command prompt:

- On Windows platforms:

```
%JAVA_HOME%\bin\java
 -classpath
.\MagicDemoJavaApp;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;.\magicsquare\distrib\magicsquare.jar
getmagic 5
```

- On UNIX platforms:

```
$JAVA_HOME/bin/java
 -classpath
./MagicDemoJavaApp:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:./magicsquare/distrib/magicsquare.jar
getmagic 5
```

Inspect the syntax of the `java` command on Windows platforms:

```
%JAVA_HOME%\bin\java
 -classpath
.\MagicDemoJavaApp;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;.\magicsquare\distrib\magicsquare.jar
getmagic 5
```

Note If you are running on Solaris 64-bit, you must add the `-d64` flag in the Java command. See "Limitations and Restrictions" on page 6-2 for more specific information.

The components of this command are:

- `%JAVA_HOME%\bin\java` — Using this command invokes the `java` run time explicitly from the MATLAB JRE.

- `-classpath` — Using this argument allows Java to access the packages and other files you need to run your application.

- `.\MagicDemoJavaApp;` — The location of `getmagic.java`. The semicolon concatenates this file location with the following file location, so Java can find the files needed to run your program.

- `matlabroot\toolbox\javabuilder\jar\javabuilder.jar;` — The location of the MATLAB Builder JA package file (`com.mathworks.toolbox.javabuilder`). The semicolon concatenates this file location with the following file location, so Java can find the files needed to run your program.

- `.\magicsquare\distrib\magicsquare.jar` — The location of the `magicsquare` package file you created with `deploytool`.

- `getmagic 5` — Invokes the compiled `getmagic` application with the command-line argument `5`.

3 Verify the program output. If the program ran successfully, a magic square of order 5 will print, matching the output of the M-function you ran in "Testing the M-File You Want to Deploy" on page 1-12, as follows:

```
Magic square of order 5
```

```
17 24  1  8 15
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
```

Using mcrroot to Test Against the MCR. *jre_directory* refers to
the directory described in "Setting Up Your Java Environment" on
page 1-7. To test directly against the MCR, substitute *mcrroot*
for *matlabroot*, where *mcrroot* is the location where the MCR is
installed on your system. An example of an MCR root location is
D:\Applications\MATLAB\MATLAB_Component_Runtime*MCR_version_number*.
Remember to double-quote all parts of the java command path arguments
that contain spaces.

Distributing the Component to End Users

If you bundled the component as a self-extracting executable, paste it in a
directory on the development machine, and run it. If you are using a .zip
file bundled with WinZip, unzip and extract the contents to the development
machine.

Integrating Java Classes Generated by MATLAB into a Java Application

If you are implementing your Java component on a computer other than the
one on which it was built:

1 Install the MATLAB Compiler Runtime on the target system. See
"Deployment Process" in the MATLAB Compiler documentation.

2 Consult the Javadoc for information on classes generated by MATLAB
classes. Reference the Javadoc from the MATLAB Builder JA product
roadmap.

3 To integrate the Java class generated by MATLAB Builder JA,software
both the component and the MWArray API need to be imported in the Java
code. Import the MATLAB libraries and the component classes into your
code with the Java import function. For example:

```
import com.mathworks.toolbox.javabuilder.*;
```

```
import componentname.classname; or import componentname.*;
```

For more information, see Chapter 3, "Programming".

4 As with all Java classes, you must use the `new` function to create an instance of a class. To create an object (`theMagic`) from the `magic` class, the example application uses the following code:

```
theMagic = new magic();
```

For more information, see Chapter 3, "Programming".

5 To conserve system resources and optimize performance, it is good practice to get in the habit of destroying any instances of classes that are no longer needed. For example, to dispose of the object `theMagic`, use the following code:

```
theMagic.dispose();
/* Make it eligible for garbage collection */
theMagic = null;
```

For more information, see Chapter 3, "Programming", in particular, "Using the dispose Method" on page 3-35.

Calling Class Methods from Java

After you have instantiated the class, you can call a class method as you would with any Java object. In the Magic Square example, the `makesqr` method is called as shown:

```
result = theMagic.makesqr(1, n);
```

Here `n` is an instance of an `MWArray` class. Note that the first argument expresses number of outputs (1) and succeeding arguments represent inputs (n).

See the following code fragment for the declaration of `n`:

```
n = new MWNumericArray(Double.valueOf(args[0],
                            MWClassID.DOUBLE);
```

Note The MATLAB Builder JA product provides a rich API for integrating the generated components. Detailed examples and complete listings of input parameters and possible thrown exceptions can be found in the Javadoc, available from thethe MATLAB Builder JA roadmap.

Handle Data Conversion as Needed

When you invoke a method on a builder component, the input parameters received by the method must be in the MATLAB internal array format. You can either (manually) convert them yourself within the calling program, or pass the parameters as Java data types.

- To manually convert to one of the standard MATLAB data types, use MWArray classes in the package `com.mathworks.toolbox.javabuilder`.

- If you pass them as Java data types, they are automatically converted.

Build and Test

Build and test the Java application as you would any application in your end user's environment. Build on what you've created by working with additional classes and methods.

After you create and distribute the initial application, you will want to continue to enhance it. Details about some of the more common tasks you will perform as you develop your application are listed here. For detailed instructions, see "Next Steps" on page 1-29.

Next Steps

Understanding other concepts needed to use the MATLAB Builder JA product	Chapter 2, "Concepts"
Writing Java applications that can access Java methods that encapsulate M-code	Chapter 3, "Programming"
Sample applications that access methods developed in MATLAB	Chapter 4, "Sample Java Applications"
Deploying Java components over the Web	Chapter 5, "Deploying a Java Component Over the Web"
Reference information about automatic data conversion rules	Chapter 6, "Reference Information for Java"

Concepts

A component created by the MATLAB Builder JA product is a standalone Java package (`.jar` file). The package contains one or more Java classes that encapsulate M-code. The classes provide methods that are callable directly from Java code.

To use the MATLAB Builder JA product, you create a project, which specifies the M-code to be used in the components that you want to create. This product supports data conversion between Java types and MATLAB types.

For more information about these concepts and about how the product works, see the following topics:

What Is a Project?

Overview

A builder project contains information about the files and settings needed by the MATLAB Builder JA product to create a deployable Java component. A project specifies information about classes and methods, including the MATLAB functions to be included.

Classes and Methods

The builder transforms MATLAB functions that are specified in the component's project to methods belonging to a Java class.

When creating a component, you must provide one or more class names as well as a component name. The class name denotes the name of the class that encapsulates MATLAB functions.

To access the features and operations provided by the MATLAB functions, instantiate the Java class generated by the builder, and then call the methods that encapsulate the MATLAB functions.

Note When you add files to a project, you do not have to add any M-files for functions that are called by the functions that you add. When the MATLAB Builder JA product builds a component, it automatically includes any M-functions called by the functions that you explicitly specify for the component. See "Spectral Analysis Example" on page 4-8 for a sample application that illustrates this feature.

Naming Conventions

Typically you should specify names for components and classes that will be clear to programmers who use your components. For example, if you are encapsulating many MATLAB functions, it helps to determine a scheme of function categories and to create a separate class for each category. Also, the name of each class should be descriptive of what the class does.

Valid characters are any alpha or numeric characters, as well as the underscore (_) character.

How Does the MATLAB Builder JA Product Use JAR Files?

As of R2007b, the MATLAB Builder JA product now embeds the CTF archive within the generated JAR file, by default. This offers convenient deployment of a single output file since all encrypted M-file data is now contained within this Java archive.

For information on how to produce a separate CTF archive and JAR file (the default behavior prior to R2007b), see "Using MCR Component Cache and MWComponentOptions" on page 3-52 and learn how to use the `MWCtfExtractLocation.EXTRACT_TO_COMPONENT_DIR` value with the `ExtractLocation` option of `MWComponentOptions`.

How Does the MATLAB Builder JA Product Handle Data?

MATLAB Builder JA Product API

To enable Java applications to exchange data with MATLAB methods they invoke, the builder provides an API, which is implemented as the `com.mathworks.toolbox.javabuilder.MWArray` package. This package provides a set of data conversion classes derived from the abstract class, `MWArray`. Each class represents a MATLAB data type.

Understanding the API Data Conversion Classes

When writing your Java application, you can represent your data using objects of any of the data conversion classes. Alternatively, you can use standard Java data types and objects.

The data conversion classes are built as a class hierarchy that represents the major MATLAB array types.

Note This discussion provides conceptual information about the classes. For details, see `com.mathworks.toolbox.javabuilder`.

This discussion assumes you have a working knowledge of the Java programming language and the Java Software Development Kit (SDK). This is not intended to be a discussion on how to program in Java. Refer to the documentation that came with your Java SDK for general programming information.

Overview of Classes and Methods in the Data Conversion Class Hierarchy

The root of the data conversion class hierarchy is the `MWArray` abstract class. The `MWArray` class has the following subclasses representing the major MATLAB types: `MWNumericArray`, `MWLogicalArray`, `MWCharArray`, `MWCellArray`, and `MWStructArray`.

Each subclass stores a reference to a native MATLAB array of that type. Each class provides constructors and a basic set of methods for accessing the underlying array's properties and data. To be specific, `MWArray` and the classes derived from `MWArray` provide the following:

• Constructors and finalizers to instantiate and dispose of MATLAB arrays

• `get` and `set` methods to read and write the array data

• Methods to identify properties of the array

• Comparison methods to test the equality or order of the array

• Conversion methods to convert to other data types

Advantage of Using Data Conversion Classes

The `MWArray` data conversion classes let you pass native type parameters directly without using explicit data conversion. If you pass the same array frequently, you might improve the performance of your program by storing the array in an instance of one of the `MWArray` subclasses.

Automatic Conversion to MATLAB Types

Note Because the conversion process is automatic (in most cases), you do not need to understand the conversion process to pass and return arguments with MATLAB Builder JA components.

When you pass an `MWArray` instance as an input argument, the encapsulated MATLAB array is passed directly to the method being called.

In contrast, if your code uses a native Java primitive or array as an input parameter, the builder converts it to an instance of the appropriate `MWArray`

class before it is passed to the method. The builder can convert any Java string, numeric type, or any multidimensional array of these types to an appropriate MWArray type, using its data conversion rules. See "Data Conversion Rules" on page 6-8 for a list of all the data types that are supported along with their equivalent types in MATLAB.

The conversion rules apply not only when calling your own methods, but also when calling constructors and factory methods belonging to the MWArray classes.

Note There are some data types commonly used in MATLAB that are not available as native Java types. Examples are cell arrays and arrays of complex numbers. Represent these array types as instances of MWCellArray and MWNumericArray, respectively.

Understanding Function Signatures Generated by the MATLAB Builder JA Product

The Java programming language now supports optional function arguments in the way that MATLAB does with varargin and varargout. To support this feature of MATLAB, the builder generates a single overloaded Java method that accommodates any number of input arguments. This behavior is an enhancement over previous versions of varargin support that only handled a limited number of arguments.

Note In addition to handling optional function arguments, the overloaded Java methods that wrap MATLAB functions handle data conversion. See "Automatic Conversion to MATLAB Types" on page 2-6 for more details.

Understanding MATLAB Function Signatures

As background, recall that the generic MATLAB function has the following structure:

```
function [Out1, Out2, ..., varargout]=foo(In1, In2, ..., varargin)
```

To the *left* of the equal sign, the function specifies a set of explicit and optional return arguments.

To the *right* of the equal sign, the function lists explicit *input* arguments followed by one or more optional arguments.

Each argument represents a MATLAB type. When you include the varargin or varargout argument, you can specify any number of inputs or outputs beyond the ones that are explicitly declared.

Overloaded Methods in Java That Encapsulate M-Code

When the MATLAB Builder JA product encapsulates your M-code, it creates an overloaded method that implements the MATLAB functions. This overloaded method corresponds to a call to the generic MATLAB function for each combination of the possible number and type of input arguments.

In addition to encapsulating input arguments, the builder creates another method, which represents the output arguments, or return values, of the MATLAB function. This additional overloaded method takes care of return values for the encapsulated MATLAB function. This method of encapsulating the information about return values simulates the mlx interface in the MATLAB Compiler product.

These overloaded methods are called the standard interface (encapsulating input arguments) and the mlx interface (encapsulating return values). See "Programming Interfaces Generated by the MATLAB® Builder JA Product" on page 6-12 for details.

Returning Data from MATLAB to Java

All data returned from a method coded in MATLAB is passed as an instance of the appropriate MWArray subclass. For example, a MATLAB cell array is returned to the Java application as an MWCellArray object.

Return data is *not* converted to a Java type. If you choose to use a Java type, you must convert to that type using the toArray method of the MWArray subclass to which the return data belongs.

What Happens in the Build Process?

Note The MATLAB Builder JA product uses the `JAVA_HOME` variable to locate the Java Software Development Kit (SDK) on your system. The compiler uses this variable to set the version of the `javac.exe` command it uses during compilation.

To create a component, the builder does the following:

1 Generates Java code to implement your component. The files are as follows:

myclass`.java`	Contains a Java class with methods encapsulating the M-functions specified in the project for that class.
mycomponent`MCR.java`	Contains the CTF decryption keys and code to initialize the MCR for the component.

2 Compiles the Java code produced in step 1.

3 Generates `/distrib` and `/src` subdirectories.

4 Invokes the Jar utility to package the Java class files it has created into a Java archive file (*mycomponent*`.jar`).

What Happens in the Packaging Process?

The packaging process creates a self-extracting executable (on Windows platforms) or a .zip file (on platforms other than Windows). The package contains at least the following:

- The builder component
- The MCR Installer (if the **Install MCR** option was selected when the component was built)
- Documentation generated by the Sun Microsystems Javadoc tool

Note The packaging process is not available when using mcc directly.

Note For guidelines on multi-platform portability, reference "Ensuring Multi-Platform Portability" on page 3-50.

How Does Component Deployment Work?

There are two kinds of deployment:

- Installing components and setting up support for them on a development machine so that they can be accessed by a developer who seeks to use them in writing a Java application.

- Deploying support for the components when they are accessed at run time on an end-user machine.

  To accomplish this kind of deployment, you must make sure that the installer you create for the application takes care of supporting the Java components on the target machine. In general, this means the MCR must be installed, on the target machine. You must also install the MATLAB Builder JA component.

Note Java components created with the MATLAB Builder JA product are dependent on the version of MATLAB with which they were built.

Programming

To access a Java component built and packaged by the MATLAB Builder JA product, you must first unpackage and install components so you can use them on a particular machine. See Chapter 1, "Getting Started" for more information.

Then you perform the following programming tasks:

Note For conceptual information that might help you in approaching these tasks, see Chapter 2, "Concepts".

For examples of these tasks, see Chapter 4, "Sample Java Applications".

For information about deploying your application after you complete these tasks, see "How Does Component Deployment Work?" on page 2-11.

Importing Classes

To use a component generated by the MATLAB Builder JA product:

1 Import MATLAB libraries with the Java `import` function, for example:

```
import com.mathworks.toolbox.javabuilder.*;
```

2 Import the component classes created by the builder, for example:

```
import com.mathworks.componentname.classname;
```

> **Note** It is important to note the difference between the component and the package names. The component name is the last part of the full package name, and is what is used in the .JAR file (and the embedded CTF file within the JAR). For example, in `mcc -W java:com.mathworks.demos,HelloDemo hello.m`, the component name is `demos` and the package name is `com.mathworks.demos`. The `import` statement should include the full package name: `import com.mathworks.demos.HelloDemo;`

> **Note** When you use the MATLAB Builder JA product to create classes, you must create those classes on the same operating system to which you are deploying them for development (or for use by end users running an application). For example, if your goal is to deploy an application to end users to run on Windows, you must create the Java classes with the MATLAB Builder JA product running on Windows.

> The reason for this limitation is that although the `.jar` file itself might be platform independent, the `.jar` file is dependent on the `.ctf` file, which is intrinsically platform dependent. It is possible to make your `.ctf` file platform independent in certain circumstances; see "Ensuring Multi-Platform Portability" on page 3-50 for more details.

Creating an Instance of the Class

What Is an Instance?

As with any Java class, you need to instantiate the classes you create with the MATLAB Builder JA product before you can use them in your program.

Suppose you build a component named `MyComponent` with a class named `MyClass`. Here is an example of creating an instance of the `MyClass` class:

```
MyClass ClassInstance = new MyClass();
```

Code Fragment: Instantiating a Java Class

The following Java code shows how to create an instance of a class that was built with MATLAB Builder JA. The application uses a Java class that encapsulates a MATLAB function, myprimes.

```
/*
 * usemyclass.java uses myclass
 */

/* Import all com.mathworks.toolbox.javabuilder classes */
import com.mathworks.toolbox.javabuilder.*;

/* Import all com.mycompany.mycomponent classes */
import com.mycompany.mycomponent.*;

/*
 * usemyclass class
 */
public class usemyclass
{
   /** Constructs a new usemyclass */
   public usemyclass()
```

```
      {
         super();
      }

      /* Returns an array containing the primes between O and n */
      public double[] getprimes(int n) throws MWException
      {
         myclass cls = null;
         Object[] y = null;

         try
         {
            cls = new myclass();
            y = cls.myprimes(1, new Double((double)n));
            return (double[])((MWArray)y[O]).getData();
         }

         catch (MWException e) {
             // something went wrong while initializing the component - the
             //  MWException's message contains more information
         }

         finally
         {
            MWArray.disposeArray(y);
            if (cls != null)
            cls.dispose();
         }
      }
   }
```

The import statements at the beginning of the program import packages that define all the classes that the program requires. These classes are defined in javabuilder.* and mycomponent.*; the latter defines the class myclass.

The following statement instantiates the class myclass:

```
  cls = new myclass();
```

The following statement calls the class method myprimes:

```
y = cls.myprimes(1, new Double((double)n));
```

The sample code passes a `java.lang.Double` to the `myprimes` method. The `java.lang.Double` is automatically converted to the `double` data type required by the encapsulated MATLAB `myprimes` function.

When `myprimes` executes, it finds all prime numbers between 0 and the input value and returns them in a MATLAB `double` array. This array is returned to the Java program as an `MWNumericArray` with its `MWClassID` property set to `MWClassID.DOUBLE`.

The `myprimes` method encapsulates the `myprimes` function.

myprimes Function

The code for `myprimes` is as follows:

```
function p = myprimes(n)
%    MYPRIMES Returns the primes between 0 and n.
%    P = MYPRIMES(N) Returns the primes between 0 and n.
%    This file is used as an example for the MATLAB
%    Builder for Java product.

%    Copyright 2001-2007 The MathWorks, Inc.

if length(n) ~= 1
   error('N must be a scalar');
end

if n < 2
   p = zeros(1,0);
   return
end

p = 1:2:n;
q = length(p);
p(1) = 2;

for k = 3:2:sqrt(n)
   if p((k+1)/2)
```

```
        p(((k*k+1)/2):k:q) = 0;
    end
end

p = (p(p>0));
```

Passing Arguments to and from Java

In this section...

Format

When you invoke a method on a MATLAB Builder JA component, the input arguments received by the method must be in the MATLAB internal array format. You can either convert them yourself within the calling program, or pass the arguments as Java data types, which are then automatically converted by the calling mechanism.

To convert them yourself, use instances of the MWArray classes; in this case you are using *manual conversion.* Storing your data using the classes and data types defined in the Java language means that you are relying on *automatic conversion.* Most likely, you will use a combination of manual and automatic conversion.

Manual Conversion of Data Types

To manually convert to one of the standard MATLAB data types, use the MWArray data conversion classes provided by the builder. For class reference and usage information, see the com.mathworks.toolbox.javabuilder package.

Code Fragment: Using MWNumericArray

The Magic Square example ("Deploying a Component with the Magic Square Example" on page 1-9) exemplifies manual conversion. The following code fragment from that program shows a java.lang.Double argument that is converted to an MWNumericArray type that can be used by the M-function without further conversion:

```
MWNumericArray dims = null;
dims = new MWNumericArray(Double.valueOf(args[O]),
                MWClassID.DOUBLE);

result = theMagic.makesqr(1, dims);
```

Code Fragment: Passing an MWArray. This example constructs an MWNumericArray of type MWClassID.DOUBLE. The call to myprimes passes the number of outputs, 1, and the MWNumericArray, x:

```
x = new MWNumericArray(n, MWClassID.DOUBLE);
cls = new myclass();
y = cls.myprimes(1, x);
```

MATLAB Builder JA product converts the MWNumericArray object to a MATLAB scalar double to pass to the M-function.

Automatic Conversion to a MATLAB Type

When passing an argument only a small number of times, it is usually just as efficient to pass a primitive Java type or object. In this case, the calling mechanism converts the data for you into an equivalent MATLAB type.

For instance, either of the following Java types would be automatically converted to the MATLAB double type:

- A Java double primitive
- An object of class java.lang.Double

For reference information about data conversion (tables showing each Java type along with its converted MATLAB type, and each MATLAB type with its converted Java type), see "Data Conversion Rules" on page 6-8.

Code Fragment: Automatic Data Conversion

When calling the makesqr method used in the getmagic application, you could construct an object of type MWNumericArray. Doing so would be an example of manual conversion. Instead, you could rely on automatic conversion, as shown in the following code fragment:

```
result = M.makesqr(1, arg[0]);
```

In this case, a Java `double` is passed as `arg[0]`.

Here is another example:

```
result = theFourier.plotfft(3, data, new Double(interval));
```

In this Java statement, the third argument is of type `java.lang.Double`. According to conversion rules, the `java.lang.Double` automatically converts to a MATLAB 1-by-1 `double` array.

Code Fragment: Passing a Java Double Object

The example calls the `myprimes` method with two arguments. The first specifies the number of arguments to return. The second is an object of class `java.lang.Double` that passes the one data input to `myprimes`.

```
cls = new myclass();
y = cls.myprimes(1, new Double((double)n));
```

This second argument is converted to a MATLAB 1-by-1 `double` array, as required by the M-function. This is the default conversion rule for `java.lang.Double`.

Code Fragment: Passing an MWArray

This example constructs an `MWNumericArray` of type `MWClassID.DOUBLE`. The call to `myprimes` passes the number of outputs, 1, and the `MWNumericArray`, x.

```
x = new MWNumericArray(n, MWClassID.DOUBLE);
cls = new myclass();
y = cls.myprimes(1, x);
```

builder converts the `MWNumericArray` object to a MATLAB scalar `double` to pass to the M-function.

Code Fragment: Calling MWArray Methods

The conversion rules apply not only when calling your own methods, but also when calling constructors and factory methods belonging to the `MWArray` classes.

For example, the following code fragment calls the constructor for the `MWNumericArray` class with a Java `double` as the input argument:

```
double Adata = 24;
MWNumericArray A = new MWnumericArray(Adata);
System.out.println("Array A is of type " + A.classID());
```

The builder converts the input argument to an instance of `MWNumericArray`, with a `ClassID` property of `MWClassID.DOUBLE`. This `MWNumericArray` object is the equivalent of a MATLAB 1-by-1 `double` array.

When you run this example, the result is as follows:

```
Array A is of type double
```

Changing the Default by Specifying the Type

When calling an `MWArray` class method constructor, supplying a specific data type causes the MATLAB Builder JA product to convert to that type instead of the default.

For example, in the following code fragment, the code specifies that A should be constructed as a MATLAB 1-by-1 16-bit integer array:

```
double Adata = 24;
MWNumericArray A = new MWnumericArray(Adata, MWClassID.INT16);
System.out.println("Array A is of type " + A.classID());
```

When you run this example, the result is as follows:

```
Array A is of type int16
```

Specifying Optional Arguments

So far, the examples have not used M-functions that have `varargin` or `varargout` arguments. Consider the following M-function:

```
function y = mysum(varargin)
%   MYSUM Returns the sum of the inputs.
%   Y = MYSUM(VARARGIN) Returns the sum of the inputs.
%   This file is used as an example for the MATLAB
%   Builder for Java product.

%   Copyright 2001-2007 The MathWorks, Inc.

y = sum([varargin{:}]);
```

This function returns the sum of the inputs. The inputs are provided as a varargin argument, which means that the caller can specify any number of inputs to the function. The result is returned as a scalar double.

Code Fragment: Passing Variable Numbers of Inputs

The MATLAB Builder JA product generates a Java interface to this function as follows:

```
/* mlx interface - List version*/
public void mysum(List lhs, List rhs)
                            throws MWException
{
   (implementation omitted)
}
/* mlx interface - Array version*/
public void mysum(Object[] lhs, Object[] rhs)
                            throws MWException
{
   (implementation omitted)
}

/* standard interface - no inputs */
public Object[] mysum(int nargout) throws MWException
{
   (implementation omitted)
}

/* standard interface - variable inputs */
public Object[] mysum(int nargout, Object varargin)
```

```
                                    throws MWException
{
    (implementation omitted)
}
```

In all cases, the `varargin` argument is passed as type `Object`. This lets you provide any number of inputs in the form of an array of Object, that is `Object[]`, and the contents of this array are passed to the compiled M-function in the order in which they appear in the array. Here is an example of how you might use the `mysum` method in a Java program:

```
public double getsum(double[] vals) throws MWException
{
    myclass cls = null;
    Object[] x = {vals};
    Object[] y = null;

    try
    {
        cls = new myclass();
        y = cls.mysum(1, x);
        return ((MWNumericArray)y[0]).getDouble(1);
    }

    finally
    {
        MWArray.disposeArray(y);
        if (cls != null)
         cls.dispose();
    }
}
```

In this example, an `Object` array of length 1 is created and initialized with a reference to the supplied `double` array. This argument is passed to the `mysum` method. The result is known to be a scalar `double`, so the code returns this `double` value with the statement:

```
return ((MWNumericArray)y[0]).getDouble(1);
```

Cast the return value to `MWNumericArray` and invoke the `getDouble(int)` method to return the first element in the array as a primitive `double` value.

Code Fragment: Passing Array Inputs. The next example performs a more general calculation:

```
public double getsum(Object[] vals) throws MWException
{
   myclass cls = null;
   Object[] x = null;
   Object[] y = null;

   try
   {
      x = new Object[vals.length];
      for (int i = O; i < vals.length; i++)
         x[i] = new MWNumericArray(vals[i], MWClassID.DOUBLE);

      cls = new myclass();
      y = cls.mysum(1, x);
      return ((MWNumericArray)y[O]).getDouble(1);
   }
   finally
   {
      MWArray.disposeArray(x);
      MWArray.disposeArray(y);
      if (cls != null)
         cls.dispose();
   }
}
```

This version of `getsum` takes an array of `Object` as input and converts each value to a `double` array. The list of `double` arrays is then passed to the `mysum` function, where it calculates the total sum of each input array.

Code Fragment: Passing a Variable Number of Outputs

When present, `varargout` arguments are handled in the same way that `varargin` arguments are handled. Consider the following M-function:

```
function varargout = randvectors
```

```
%   RANDVECTORS Returns a list of random vectors.
%   VARARGOUT = RANDVECTORS Returns a list of random
%   vectors such that the length of the ith vector = i.
%   This file is used as an example for the MATLAB
%   Builder for Java product.

%   Copyright 2001-2007 The MathWorks, Inc.

for i=1:nargout
   varargout{i} = rand(1, i);
end
```

This function returns a list of random double vectors such that the length of
the ith vector is equal to i. The MATLAB Compiler product generates a Java
interface to this function as follows:

```
/* mlx interface - List version */
 public void randvectors(List lhs, List rhs) throws MWException
{
   (implementation omitted)
}
/* mlx interface   Array version */
public void randvectors(Object[] lhs, Object[] rhs) throws MWException
{
   (implementation omitted)
}
/* Standard interface   no inputs*/
public Object[] randvectors(int nargout) throws MWException
{
   (implementation omitted)
}
```

**Code Fragment: Passing Optional Arguments with the Standard
Interface.** Here is one way to use the randvectors method in a Java
program:

```
public double[][] getrandvectors(int n) throws MWException
{
   myclass cls = null;
   Object[] y = null;
```

```
try
{
    cls = new myclass();
    y = cls.randvectors(n);
    double[][] ret = new double[y.length][];

    for (int i = 0; i < y.length; i++)
        ret[i] = (double[])((MWArray)y[i]).getData();
    return ret;
}

finally
{
    MWArray.disposeArray(y);
    if (cls != null)
        cls.dispose();
}
}
```

The getrandvectors method returns a two-dimensional double array with
a triangular structure. The length of the ith row equals i. Such arrays are
commonly referred to as *jagged* arrays. Jagged arrays are easily supported in
Java because a Java matrix is just an array of arrays.

Handling Return Values

The previous examples used the fact that you knew the type and
dimensionality of the output argument. In the case that this information is
unknown, or can vary (as is possible in M-programming), the code that calls
the method might need to query the type and dimensionality of the output
arguments.

There are several ways to do this. Do one of the following:

• Use reflection support in the Java language to query any object for its type.

• Use several methods provided by the MWArray class to query information
  about the underlying MATLAB array.

• Coercing to a specific type using the to*Type*Array methods.

Code Fragment: Using Java Reflection

This code sample calls the myprimes method, and then determines the type
using reflection. The example assumes that the output is returned as a
numeric matrix but the exact numeric type is unknown.

```java
public void getprimes(int n) throws MWException
{
   myclass cls = null;
   Object[] y = null;

   try
   {
      cls = new myclass();
      y = cls.myprimes(1, new Double((double)n));
      Object a = ((MWArray)y[0]).toArray();

      if (a instanceof double[][])
      {
         double[][] x = (double[][])a;

         /* (do something with x...) */
      }

      else if (a instanceof float[][])
      {
         float[][] x = (float[][])a;

         /* (do something with x...) */
      }

      else if (a instanceof int[][])
      {
         int[][] x = (int[][])a;

         /* (do something with x...) */
      }

      else if (a instanceof long[][])
      {
         long[][] x = (long[][])a;
```

```
            /* (do something with x...) */
        }

        else if (a instanceof short[][])
        {
            short[][] x = (short[][])a;

            /* (do something with x...) */
        }

        else if (a instanceof byte[][])
        {
            byte[][] x = (byte[][])a;

            /* (do something with x...) */
        }

        else
        {
            throw new MWException(
                "Bad type returned from myprimes");
        }
    }
```

This example uses the toArray method to return a Java primitive array
representing the underlying MATLAB array. The toArray method works just
like getData in the previous examples, except that the returned array has the
same dimensionality as the underlying MATLAB array.

Code Fragment: Using MWArray Query

The next example uses the MWArray classID method to determine the type
of the underlying MATLAB array. It also checks the dimensionality by
calling numberOfDimensions. If any unexpected information is returned, an
exception is thrown. It then checks the MWClassID and processes the array
accordingly.

```
    public void getprimes(int n) throws MWException
    {
```

```
myclass cls = null;
Object[] y = null;

try
{
   cls = new myclass();
   y = cls.myprimes(1, new Double((double)n));
   MWClassID clsid = ((MWArray)y[0]).classID();

   if (!clsid.isNumeric() ||
      ((MWArray)y[0]).numberOfDimensions() != 2)
   {
      throw new MWException("Bad type returned from myprimes");
   }

   if (clsid == MWClassID.DOUBLE)
   {
      double[][] x = (double[][])((MWArray)y[0]).toArray();

      /* (do something with x...) */
   }

   else if (clsid == MWClassID.SINGLE)
   {
      float[][] x = (float[][])((MWArray)y[0]).toArray();

      /* (do something with x...) */
   }

   else if (clsid == MWClassID.INT32 ||
         clsid == MWClassID.UINT32)
   {
      int[][] x = (int[][])((MWArray)y[0]).toArray();

      /* (do something with x...) */
   }

   else if (clsid == MWClassID.INT64 ||
         clsid == MWClassID.UINT64)
   {
```

3-19

```
               long[][] x = (long[][])((MWArray)y[0]).toArray();

               /* (do something with x...) */
           }

           else if (clsid == MWClassID.INT16 ||
                    clsid == MWClassID.UINT16)
           {
               short[][] x = (short[][])((MWArray)y[0]).toArray();

               /* (do something with x...) */
           }

           else if (clsid == MWClassID.INT8 ||
                    clsid == MWClassID.UINT8)
           {
               byte[][] x = (byte[][])((MWArray)y[0]).toArray();

               /* (do something with x...) */
           }
       }
       finally
       {
           MWArray.disposeArray(y);
           if (cls != null)
               cls.dispose();
       }
   }
```

Code Fragment: Using to*Type*Array Methods

The next example demonstrates how you can coerce or force data to a specified numeric type by invoking any of the to*Type*Array methods. These methods return an array of Java types matching the primitive type specified in the name of the called method. The data is coerced or forced to the primitive type specified in the method name. Note that when using these methods, data will be truncated when needed to allow conformance to the specified data type.

```
   Object results = null;
```

```
try {
    // call a compiled m-function
    results = myobject.myfunction(2);

    // first output is known to be a numeric matrix
    MWArray resultA = (MWNumericArray) results[0];
  double[][] a = (double[][]) resultA.toDoubleArray();

    // second output is known to be a 3-dimensional numeric array
    MWArray resultB = (MWNumericArray) results[1];
  Int[][][] b = (Int[][][]) resultB.toIntArray();
}
 finally {
    MWArray.disposeArray(results);
}
```

Passing Java Objects by Reference

In this section...

"MATLAB Array" on page 3-22

"Wrappering and Passing Java Objects to M-Functions with MWJavaObjectRef" on page 3-22

MATLAB Array

`MWJavaObjectRef`, a special subclass of `MWArray`, can be used to create a MATLAB array that references Java objects. For detailed usage information on this class, constructor, and associated methods, see the MWJavaObjectRef page in the Javadoc or search for `MWJavaObjectRef` in the MATLAB Help browser **Search** field.

Wrappering and Passing Java Objects to M-Functions with MWJavaObjectRef

You can create an M-code wrapper around Java objects using `MWJavaObjectRef`. Using this technique, you can pass objects by reference to MATLAB functions, clone a Java object inside a MATLAB Builder JA component, as well as perform other object marshalling specific to the MATLAB Compiler product. The examples in this section present some common use cases.

Code Fragment: Passing a Java Object into a MATLAB Builder JA Component

To pass an object into a MATLAB Builder JA component:

1 Use `MWJavaObjectRef` to wrap your object.

2 Pass your object to an M-function. For example:

```
/* Create an object */
java.util.Hashtable<String,Integer> hash =
                    new java.util.Hashtable<String,Integer>();
hash.put("One", 1);
```

```
hash.put("Two", 2);
System.out.println("hash: ");
System.out.println(hash.toString());

/* Create a MWJavaObjectRef to wrap this object */
origRef = new MWJavaObjectRef(hash);

/* Pass it to an M-function that lists its methods, etc */
result = theComponent.displayObj(1, origRef);
MWArray.disposeArray(origRef);
```

For reference, here is the source code for displayObj.m:

displayObj.m.

```
function className = displayObj(h)

disp('--------------------------');
disp('Entering M-function')
h
className = class(h)
whos('h')
methods(h)


disp('Leaving M-function')
disp('--------------------------');
```

Code Fragment: Cloning an Object Inside a Builder Component

You can also use MWJavaObjectRef to clone an object inside a MATLAB Builder JA component. Continuing with the example in "Code Fragment: Passing a Java Object into a MATLAB® Builder JA Component" on page 3-22, do the following:

1 Add to the original hash.

2 Clone the object.

3 (Optional) Continue to add items to each copy. For example:

```
                     origRef = new MWJavaObjectRef(hash);
                     System.out.println("hash:");
                     System.out.println(hash.toString());
                     result = theComponent.addToHash(1, origRef);

                     outputRef = (MWJavaObjectRef)result[0];

                     /* We can typecheck that the reference contains a      */
                     /*       Hashtable but not <String,Integer>;            */
                     /* this can cause issues if we get a Hashtable<wrong,wrong>. */
                     java.util.Hashtable<String, Integer> outHash =
                             (java.util.Hashtable<String,Integer>)(outputRef.get());

                     /* We've added items to the original hash, cloned it, */
                     /* then added items to each copy */
                     System.out.println("hash:");
                     System.out.println(hash.toString());
                     System.out.println("outHash:");
                     System.out.println(outHash.toString());
```

For reference, here is the source code for addToHash.m:

addToHash.m.

```
function h2 = addToHash(h)
%ADDTOHASH Add elements to a java.util.Hashtable<String, Integer>
%   This file is used as an example for the
%   MATLAB Builder JA product.

%   Copyright 2001-2007 The MathWorks, Inc.
%   $Revision: 1.1.4.67 $  $Date: 2008/08/07 12:22:51 $

% Validate input
if ~isa(h,'java.util.Hashtable')
    error('addToHash:IncorrectType', ...
        'addToHash expects a java.util.Hashtable');
end

% Add an item
h.put('From MATLAB',12);
```

```
% Clone the Hashtable and add items to both resulting objects
h2 = h.clone();
h.put('Orig',20);
h2.put('Clone',21);
```

Code Fragment: Passing a Date into a Component and Getting a Date from a Component

In addition to passing in created objects, as in "Code Fragment: Passing a Java Object into a MATLAB® Builder JA Component" on page 3-22, you can also use MWJavaObjectRef to pass in Java utility objects such as java.util.date. To do so, perform the following steps:

1 Get the current date and time using the Java object java.util.date.

2 Create an instance of MWJavaObjectRef in which to wrap the Java object.

3 Pass it to an M-function that performs further processing, such as nextWeek.m. For example:

```
/* Get the current date and time */
java.util.Date nowDate = new java.util.Date();
System.out.println("nowDate:");
System.out.println(nowDate.toString());

/* Create a MWJavaObjectRef to wrap this object */
origRef = new MWJavaObjectRef(nowDate);

/* Pass it to an M-function that calculates one week */
/* in the future */
result = theComponent.nextWeek(1, origRef);

outputRef = (MWJavaObjectRef)result[0];
java.util.Date nextWeekDate =
        (java.util.Date)outputRef.get();
System.out.println("nextWeekDate:");
System.out.println(nextWeekDate.toString());
```

For reference, here is the source code for nextWeek.m:

nextWeek.m.

```
function nextWeekDate = nextWeek(nowDate)
%NEXTWEEK Given one Java Date, calculate another
% one week in the future
%   This file is used as an example for the
%   MATLAB Builder JA product.

%   Copyright 2001-2007 The MathWorks, Inc.
%   $Revision: 1.1.4.67 $  $Date: 2008/08/07 12:22:51 $

% Validate input
if ~isa(nowDate,'java.util.Date')
    error('nextWeekDate:IncorrectType', ...
        'nextWeekDate expects a java.util.Date');
end

% Use java.util.Calendar to calculate one week later
% than the supplied
% java.util.Date
cal = java.util.Calendar.getInstance();
cal.setTime(nowDate);
cal.add(java.util.Calendar.DAY_OF_MONTH, 7);
nextWeekDate = cal.getTime();
```

Returning Java Objects Using unwrapJavaObjectRefs

If you want actual Java objects returned from a component (without the
MATLAB wrapping), use unwrapJavaObjectRefs.

This method recursively connects a single MWJavaObjectRef or a
multidimensional array of MWJavaObjectRef objects to a reference or array
of references.

The following code snippets show two examples of calling
unwrapJavaObjectRefs:

**Code Snippet: Returning a Single Reference or Reference to an Array
of Objects with unwrapJavaObjectRefs.**

```
Hashtable<String,Integer> myHash =
```

```
                            new Hashtable<String,Integer>();
            myHash.put("a", new Integer(3));
            myHash.put("b", new Integer(5));
            MWJavaObjectRef A = new MWJavaObjectRef(new Integer(12));
            System.out.println("A referenced the object: "
                        + MWJavaObjectRef.unwrapJavaObjectRefs(A));

            MWJavaObjectRef B = new MWJavaObjectRef(myHash);
            Object bObj = (Object)B;
            System.out.println("B referenced the object: "
                        + MWJavaObjectRef.unwrapJavaObjectRefs(bObj))
```

Produces the following output:

```
            A referenced the object: 12
            B referenced the object: {b=5, a=3}
```

Code Snippet: Returning an Array of References with unwrapJavaObjectRefs.

```
            MWJavaObjectRef A = new MWJavaObjectRef(new Integer(12));
            MWJavaObjectRef B = new MWJavaObjectRef(new Integer(104));
            Object[] refArr = new Object[2];
            refArr[0] = A;
            refArr[1] = B;
            Object[] objArr =
                              MWJavaObjectRef.unwrapJavaObjectRefs(refArr);
            System.out.println("refArr referenced the objects: " +
                              objArr[0] + " and " + objArr[1]);
```

Produces the following output:

```
            refArr referenced the objects: 12 and 104
```

Optimization Example Using MWJavaObjectRef

For a full example of how to use MWJavaObjectRef to create a reference to a Java object and pass it to a component, see "Optimization Example" on page 4-41.

Handling Errors

Error Overview

Errors that occur during execution of an M-function or during data conversion are signaled by a standard Java exception. This includes MATLAB run-time errors as well as errors in your M-code.

In general, there are two types of exceptions in Java: checked exceptions and unchecked exceptions.

Handling Checked Exceptions

Checked exceptions must be declared as thrown by a method using the Java language throws clause. MATLAB Builder JA components support one checked exception: com.mathworks.toolbox.javabuilder.MWException. This exception class inherits from java.lang.Exception and is thrown by every MATLAB Compiler generated Java method to signal that an error has occurred during the call. All normal MATLAB run-time errors, as well as user-created errors (e.g., a calling error in your M-code) are manifested as MWExceptions.

The Java interface to each M-function declares itself as throwing an MWException using the throws clause. For example, the myprimes M-function shown previously has the following interface:

```
/* mlx interface   List version */
public void myprimes(List lhs, List rhs) throws MWException
{
    (implementation omitted)
}
/* mlx interface   Array version */
public void myprimes(Object[] lhs, Object[] rhs) throws MWException
```

```
    {
       (implementation omitted)
    }
/* Standard interface    no inputs*/
public Object[] myprimes(int nargout) throws MWException
   {
       (implementation omitted)
   }
/* Standard interface    one input*/
public Object[] myprimes(int nargout, Object n) throws MWException
   {
       (implementation omitted)
   }
```

Any method that calls myprimes must do one of two things:

- Catch and handle the MWException.

- Allow the calling program to catch it.

The following two sections provide examples of each.

Code Fragment: Handling an Exception in the Called Function

The getprimes example shown here uses the first of these methods. This method handles the exception itself, and does not need to include a throws clause at the start.

```
public double[] getprimes(int n)
{
   myclass cls = null;
   Object[] y = null;

   try
   {
      cls = new myclass();
      y = cls.myprimes(1, new Double((double)n));
      return (double[])((MWArray)y[0]).getData();
   }

   /* Catches the exception thrown by myprimes */
```

```
        catch (MWException e)
        {
           System.out.println("Exception: " + e.toString());
           return new double[0];
        }

        finally
        {
           MWArray.disposeArray(y);
           if (cls != null)
              cls.dispose();
        }
   }
```

Note that in this case, it is the programmer's responsibility to return something reasonable from the method in case of an error.

The `finally` clause in the example contains code that executes after all other processing in the `try` block is executed. This code executes whether or not an exception occurs or a control flow statement like `return` or `break` is executed. It is common practice to include any cleanup code that must execute before leaving the function in a `finally` block. The documentation examples use `finally` blocks in all the code samples to free native resources that were allocated in the method.

For more information on freeing resources, see "Managing Native Resources" on page 3-34.

Code Fragment: Handling an Exception in the Calling Function

In this next example, the method that calls `myprimes` declares that it throws an `MWException`:

```
public double[] getprimes(int n) throws MWException
{
   myclass cls = null;
   Object[] y = null;

   try
   {
```

```
    cls = new myclass();
    y = cls.myprimes(1, new Double((double)n));
    return (double[])((MWArray)y[0]).getData();
}

finally
{
    MWArray.disposeArray(y);
    if (cls != null)
    cls.dispose();
}
}
```

Handling Unchecked Exceptions

Several types of unchecked exceptions can also occur during the course of execution. Unchecked exceptions are Java exceptions that do not need to be explicitly declared with a throws clause. The MWArray API classes all throw unchecked exceptions.

All unchecked exceptions thrown by MWArray and its subclasses are subclasses of java.lang.RuntimeException. The following exceptions can be thrown by MWArray:

- java.lang.RuntimeException

- java.lang.ArrayStoreException

- java.lang.NullPointerException

- java.lang.IndexOutOfBoundsException

- java.lang.NegativeArraySizeException

This list represents the most likely exceptions. Others might be added in the future.

Code Fragment: Catching General Exceptions

You can easily rewrite the getprimes example to catch any exception that can occur during the method call and data conversion. Just change the catch clause to catch a general java.lang.Exception.

```
public double[] getprimes(int n)
{
   myclass cls = null;
   Object[] y = null;

   try
   {
      cls = new myclass();
      y = cls.myprimes(1, new Double((double)n));
      return (double[])((MWArray)y[0]).getData();
   }

   /* Catches the exception thrown by anyone */
   catch (Exception e)
   {
      System.out.println("Exception: " + e.toString());
      return new double[0];
   }

   finally
   {
      MWArray.disposeArray(y);
      if (cls != null)
         cls.dispose();
   }
}
```

Code Fragment: Catching Multiple Exception Types

This second, and more general, variant of this example differentiates between
an exception generated in a compiled method call and all other exception
types by introducing two catch clauses as follows:

```
public double[] getprimes(int n)
{
   myclass cls = null;
   Object[] y = null;

   try
   {
```

```
cls = new myclass();
y = cls.myprimes(1, new Double((double)n));
return (double[])((MWArray)y[0]).getData();
}

/* Catches the exception thrown by myprimes */
catch (MWException e)
{
    System.out.println("Exception in MATLAB call: " +
        e.toString());
    return new double[0];
}

/* Catches all other exceptions */
catch (Exception e)
{
    System.out.println("Exception: " + e.toString());
    return new double[0];
}

finally
{
    MWArray.disposeArray(y);
    if (cls != null)
        cls.dispose();
}
}
```

The order of the catch clauses here is important. Because MWException is a
subclass of Exception, the catch clause for MWException must occur before
the catch clause for Exception. If the order is reversed, the MWException
catch clause will never execute.

Managing Native Resources

What Are Native Resources?

When your code accesses Java classes created by the MATLAB Builder JA product, your program uses native resources, which exist outside the control of the Java Virtual Machine (JVM).

Specifically, each *MWArray* data conversion class is a wrapper class that encapsulates a MATLAB mxArray. The encapsulated MATLAB array allocates resources from the native memory heap.

Note Native arrays should always be explicitly freed. Because the Java wrapper is small and the mxArray is relatively large, the JVM memory manager may not call the garbage collector before the native memory becomes exhausted or badly fragmented.

Using Garbage Collection Provided by the JVM

When you create a new instance of a Java class, the JVM allocates and initializes the new object. When this object goes out of scope, or becomes otherwise unreachable, it becomes eligible for garbage collection by the JVM. The memory allocated by the object is eventually freed when the garbage collector is run.

When you instantiate MWArray classes, the encapsulated MATLAB also allocates space for native resources, but these resources are not visible to the JVM and are not eligible for garbage collection by the JVM. These resources are not released by the class finalizer until the JVM determines that it is appropriate to run the garbage collector.

The resources allocated by MWArray objects can be quite large and can quickly exhaust your available memory. To avoid exhausting the native memory heap, MWArray objects should be explicitly freed as soon as possible by the application that creates them.

Using the dispose Method

The best technique for freeing resources for classes created by the MATLAB Builder JA product is to call the dispose method explicitly. Any Java object, including an MWArray object, has a dispose method.

The MWArray classes also have a finalize method, called a finalizer, that calls dispose. Although you can think of the MWArray finalizer as a kind of safety net for the cases when you do not call dispose explicitly, keep in mind that you cannot determine exactly when the JVM calls the finalizer, and the JVM might not discover memory that should be freed.

Code Fragment: Using dispose

The following example allocates an approximate 8 MB native array. To the JVM, the size of the wrapped object is just a few bytes (the size of an MWNumericArray instance) and thus not of significant size to trigger the garbage collector. This example shows why it is good practice to free the MWArray explicitly.

```
/* Allocate a huge array */
int[] dims = {1000, 1000};
MWNumericArray a = MWNumericArray.newInstance(dims,
   MWClassID.DOUBLE, MWComplexity.REAL);
        .
        .  (use the array)
        .

/* Dispose of native resources */
a.dispose();

/* Make it eligible for garbage collection */
a = null;
```

The statement `a.dispose()` frees the memory allocated by both the managed wrapper and the native MATLAB array.

The `MWArray` class provides two disposal methods: `dispose` and `disposeArray`. The `disposeArray` method is more general in that it disposes of either a single `MWArray` or an array of arrays of type `MWArray`.

Code Fragment: Using try-finally to Ensure Resources Are Freed

Typically, the best way to call the `dispose` method is from a `finally` clause in a `try-finally` block. This technique ensures that all native resources are freed before exiting the method, even if an exception is thrown at some point before the cleanup code.

Code Fragment: Using dispose in a finally Clause.

This example shows the use of `dispose` in a `finally` clause:

```
/* Allocate a huge array */
MWNumericArray a;
try
{
   int[] dims = {1000, 1000};
   a = MWNumericArray.newInstance(dims,
      MWClassID.DOUBLE, MWComplexity.REAL);
   .
   . (use the array)
   .
}

/* Dispose of native resources */
finally
{
   a.dispose();
   /* Make it eligible for garbage collection */
   a = null;
}
```

Overriding the Object.Finalize Method

You can also override the `Object.Finalize` method to help clean up native resources just before garbage collection of the managed object. Refer to your Java language reference documentation for detailed information on how to override this method.

Improving Data Access Using the MCR User Data Interface and MATLAB Builder JA

This feature provides a lightweight interface for easily accessing MCR data. It allows data to be shared between an MCR instance, the M-code running on that MCR, and the wrapper code that created the MCR. Through calls to the MCR User Data interface API, you access MCR data through creation of a per-MCR-instance associative array of `mxArrays`, consisting of a mapping from string keys to `mxArray` values. Reasons for doing this include, but are not limited to:

- You need to supply run–time configuration information to a client running an application created with the Parallel Computing Toolbox. Configuration information may be supplied (and change) on a per-execution basis. For example, two instances of the same application may run simultaneously with different configuration files.

- You want to initialize the MCR with constant values that can be accessed by all your M applications

- You want to set up a global workspace — a global variable or variables that MATLAB and your client can access

- You want to store the state of any variable or group of variables

MATLAB Builder JA supports per-MCR instance state access through an object-oriented API. Unlike MATLAB Compiler, access to per-MCR instance state is optional, rather than on by default. You can access this state by adding `setmcruserdata.m` and `getmcruserdata.m` to your deployment project or by specifying them on the command line.

For more information, see the MATLAB Compiler User's Guide.

Code Snippets

The following code snippets demonstrate storing and retrieving MCR state while working with the `MagicMatrix` function:

The MagicMatrix Function

```
function magicmatrix
```

```
key = 'MagicMatrix';
m = getmcruserdata(key);
disp(m);
m = m + 1;
setmcruserdata(key, m);
```

Building the MagicMatrix Component

```
mcc -W java:com.mathworks.magicComponent,
    magicmatrix magicmatrix.m setmcruserdata.m getmcruserdata.m
```

Calling setmcruserdata

```
// Create the key and value
String key = "MagicMatrix";
MWNumericArray inputValue = new MWNumericArray(magicData);

// Create the component class (and the associated MCR instance)
m = new magicmatrix();

// Set the key into the per-MCR instance data map under the key.
m.setmcruserdata(key, inputValue);
```

Calling getmcruserdata

```
// Create the key
string key = "MagicMatrix";

// Create an instance of the component created by .NET Builder
MagicMatrix m = new MagicMatrix();

// Retrieve the new value stored under "MagicMatrix."
MWArray result = m.getmcruserdata(key);
```

Calling pctCleanupForJavaBuilder

```
try
{
MyJavaBuilderClass m = new MyJavaBuilderClass();
m.callPCTFunctionality();
}
```

```
finally
{
m.pctRunDeployedCleanup();
m.dispose();
}
```

Caution Neglecting to call `pctRunDeployedCleanup` during program
clean-up may cause system instability and unpredictable results.

Example

This following is an end to end example showing how to build, run, and work
with the `magicmatrix` application and the MCR User Data interface API. This
function retrieves the numeric matrix stored under the key `MagicMatrix`,
adds one to it and stores the result back into the per-MCR instance data
map, overwriting the previous value.

The MagixMatrix Function

```
function magicmatrix
    key = 'MagicMatrix';
    m = getmcruserdata(key);
    disp(m);
    m = m + 1;
    setmcruserdata(key, m);
```

Building (on UNIX)

This command assumes the existence of a component named
`MagicMatrixComponent` and a user-written main program in `udata.java`.

```
javac -g -classpath \
      .:"$MWE_INSTALL/toolbox/javabuilder/jar/javabuilder.jar"\
       :./magicmatrixComponent.jar udata.java
```

Executing (on UNIX)

```
java  -classpath \
      .:"$MWE_INSTALL/toolbox/javabuilder/jar/javabuilder.jar"\
```

```
:./magicmatrixComponent.jar udata
```

udata.java

```java
/* udata.java
 * Demonstrate MCR User Data interface API in Java Builder.
 */

/* Java Builder imports */
import com.mathworks.toolbox.javabuilder.*;
import com.mathworks.magicmatrixComponent.*;

public class udata
{
    public static void main(String[] args)
    {
        MWNumericArray inputValue = null;
        Object[] result = null;
        String key = "MagicMatrix";
        MWCharArray keyMatrix = new MWCharArray(key);

        magicmatrix m = null;

        try
        {
            double[][] magicData = { {8, 1, 6},
                                     {3, 5, 7},
                                     {4, 9, 2} };
            inputValue = new MWNumericArray(magicData);
            m = new magicmatrix();
            m.setmcruserdata(key, inputValue);
            m.magicmatrix();

            result = m.getmcruserdata(1, key);

            MWClassID classID = ((MWArray)result[0]).classID();
            if (classID == MWClassID.DOUBLE)
            {
                double[][] x =
                    (double[][])((MWArray)result[0]).toArray();
```

```
                    for (int i=0; i<x.length; i++)
                    {
                        java.lang.String row = new java.lang.String();
                        for (int j=0; j<x[i].length; j++)
                            row = row + "\t" + x[i][j];
                        System.out.println(row);
                        row = "";
                    }
                }

            }
            catch (Exception e)
            {
                System.out.println("Exception: " + e.toString());
            }
            finally
            {
                if (m != null)
                    m.dispose();
                MWArray.disposeArray(inputValue);
                MWArray.disposeArray(result[0]);
            m.pctRunDeployedCleanup();
            m.dispose();

            }
        }
    }
```

Handling Data Conversion Between Java and MATLAB

Overview

The call signature for a method that encapsulates a MATLAB function uses one of the MATLAB data conversion classes to pass arguments and return output. When you call any such method, all input arguments not derived from one of the MWArray classes are converted by the builder to the correct MWArray type before being passed to the MATLAB method.

For example, consider the following Java statement:

```
result = theFourier.plotfft(3, data, new Double(interval));
```

The third argument is of type `java.lang.Double`, which converts to a MATLAB 1-by-1 `double` array.

Calling MWArray Methods

The conversion rules apply not only when calling your own methods, but also when calling constructors and factory methods belonging to the MWArray classes. For example, the following code calls the constructor for the MWNumericArray class with a Java `double` input. The MATLAB Builder JA product converts the Java `double` input to an instance of MWNumericArray having a `ClassID` property of `MWClassID.DOUBLE`. This is the equivalent of a MATLAB 1-by-1 `double` array.

```
double Adata = 24;
MWNumericArray A = new MWnumericArray(Adata);
System.out.println("Array A is of type " + A.classID());
```

When you run this example, the results are as follows:

```
Array A is of type double
```

Specifying the Type

There is an exception: if you supply a specific data type in the same constructor, the MATLAB Builder JA product converts to that type rather than following the default conversion rules. Here, the code specifies that A should be constructed as a MATLAB 1-by-1 16-bit integer array:

```
double Adata = 24;
MWNumericArray A = new MWnumericArray(Adata, MWClassID.INT16);
System.out.println("Array A is of type " + A.classID());
```

When you run this example, the results are as follows:

```
Array A is of type int16
```

Creating Buffered Images from a MATLAB Array

Use the renderArrayData method to:

- Create a buffered image from data in a given MATLAB array.

- Verify the array is of three dimensions (height, width, and color component).

- Verify the color component order is red, green, and blue.

  Search on renderArrayData in the Javadoc for information on input parameters, return values, exceptions thrown, and examples.

For a complete example of renderArrayData's implementation, see "Buffered Image Creation Example" on page 4-36.

Setting Java Properties

In this section...

How to Set Java System Properties

Set Java system properties in one of two ways:

- *In the* `Java` *statement*. Use the syntax: `java -D`*propertyname*`=`*value*, where *propertyname* is the name of the Java system property you want to set and *value* is the value to which you want the property set.

- *In the Java code.* Insert the following statement in your Java code near the top of the `main` function, before you initialize any Java components:

  ```
  System.setProperty(key,value)
  ```

  key is the name of the Java system property you want to set, and *value* is the value to which you want the property set.

Ensuring a Consistent GUI Appearance

After developing your initial GUI using the MATLAB Builder JA product, subsequent GUIs that you develop may inherit properties of the MATLAB GUI, rather than properties of your initial design. To preserve your original look and feel, set the `mathworks.DisableSetLookAndFeel` Java system property to `true`.

Code Fragment: Setting DisableSetLookAndFeel

The following are examples of how to set `mathworks.DisableSetLookAndFeel` using the techniques in "How to Set Java System Properties" on page 3-45:

- In the `Java` statement:

  ```
  java -classpath X:/mypath/tomy/javabuilder.jar
  -Dmathworks.DisableSetLookAndFeel=true
  ```

- In the Java code:

```
Class A {
main () {
        System.getProperties().set("mathworks.DisableSetLookAndFeel","true");
        foo f = newFoo();
         }
         }
```

Blocking Execution of a Console Application that Creates Figures

waitForFigures Method

The MATLAB Builder JA product adds a special `waitForFigures` method to each Java class that it creates. `waitForFigures` takes no arguments. Your application can call `waitForFigures` any time during execution.

The purpose of `waitForFigures` is to block execution of a calling program as long as figures created in encapsulated M-code are displayed. Typically you use `waitForFigures` when:

- There are one or more figures open that were created by a Java component created by the MATLAB Builder JA product.
- The method that displays the graphics requires user input before continuing.
- The method that calls the figures was called from `main()` in a console program.

When `waitForFigures` is called, execution of the calling program is blocked if any figures created by the calling object remain open.

Caution   Use care when calling the `waitForFigures` method. Calling this method from an interactive program like Microsoft® Excel® can hang the application. This method should be called *only* from console-based programs.

Code Fragment: Using waitForFigures to Block Execution of a Console Application

The following example illustrates using waitForFigures from a Java application. The example uses a Java component created by the MATLAB Builder JA product; the object encapsulates M-code that draws a simple plot.

1 Create a work directory for your source code. In this example, the directory is D:\work\plotdemo.

2 In this directory, create the following M-file:

```
drawplot.m

function drawplot()
    plot(1:10);
```

3 Use the MATLAB Builder JA product to create a Java component with the following properties:

Package name	examples
Class name	Plotter

4 Create a Java program in a file named runplot.java with the following code:

```java
import com.mathworks.toolbox.javabuilder.*;
import examples.Plotter;

public class runplot {
  public static void main(String[] args) {
    try {
      plotter p = new Plotter();
      try {
        p.showPlot();
        p.waitForFigures();
      }
      finally {
       p.dispose();
      }
```

```
        }
        catch (MWException e) {
          e.printStackTrace();
        }
      }
    }
```

5 Compile the application with the `javac` command. For an example, see "Testing the Java Component in a Java Application" on page 1-21.

When you run the application, the program displays a plot from 1 to 10 in a MATLAB figure window. The application ends when you dismiss the figure.

Note To see what happens without the call to `waitForFigures`, comment out the call, rebuild the application, and run it. In this case, the figure is drawn and is immediately destroyed as the application exits.

Ensuring Multi-Platform Portability

CTF archives containing only M-files are platform independent, as are .jar files. These files can be used out of the box on any platform providing that the platform has either MATLAB or the MCR installed.

However, if your CTF archive or JAR file contains MEX-files, which are platform dependent, do the following:

1 Compile your MEX-file once on each platform where you want to run your MATLAB Builder JA application.

For example, if you are running on a Windows machine, and you want to also run on the Linux® 64-bit platform, compile *my_mex*.c twice (once on a PC to get *my_mex*.mexw32 and then again on a Linux 64-bit machine to get *my_mex*.mexa64).

2 Create the MATLAB Builder JA component on one platform using the mcc command, using the -a flag to include the MEX-file compiled on the other platform(s). In the example above, run mcc on Windows and include the -a flag to include *my_mex*.mexa64. In this example, the mcc command would be:

```
mcc -W 'java:mycomp,myclass' my_m-file.m -a my_mex.mexa64
```

Note In this example, it is not necessary to explicitly include *my_mex*.mexw32 (providing you are running on Windows). This example assumes that *my_mex*.mexw32 and *my_mex*.mexa64 reside in the same directory.

For example, if you are running on a Windows machine and you want to ensure portability of the CTF file for a MATLAB Builder JA component that invokes the yprimes.c file (from *matlabroot*\extern\examples\mex) on the Linux 64-bit platform, execute the following mcc command:

```
mcc -W 'java:mycomp,myclass' callyprime.m -a yprime.mexa64
```

where callyprime.m can be a simple M-function as follows:

```
function callyprime
disp(yprime(1,1:4));
```

Ensure the yprime.mexa64 file is in the same directory as your Windows MEX-file.

Tip If you are unsure if your JAR file contains MEX-files, do the following:

1 Run mcc with the -v option to list the names of the MEX-files. See "-v Verbose" in the MATLAB Compiler documentation for more information.

2 Obtain appropriate versions of these files from the version of MATLAB installed on your target operating system.

3 Include these versions in the archive by running mcc with the -a option as documented in this section. See "-a Add to Archive" in the MATLAB Compiler documentation for more information.

Using MCR Component Cache and MWComponentOptions

In this section...

MWComponentOptions

As of R2007b, CTF data is now automatically extracted directly from the JAR file with no separate CTF or *componentname*mcr directory needed on the target machine. This behavior is helpful when storage space on a file system is limited.

If you don't want to use this feature, use the MWComponentOptions class to specify how the MATLAB Builder JA product handles CTF data extraction and utilization.

Selecting Options

Choose from the following CtfSource or ExtractLocation instantiation options to customize how the MATLAB Builder JA product manages CTF content with MWComponentOptions:

- CtfSource — This option specifies where the CTF file may be found for an extracted component. It defines a binary data stream comprised of the bits of the CTF file. The following values are objects of some type extending MWCtfSource:

  - MWCtfSource.NONE — Indicates that no CTF file is to be extracted. This implies that the extracted CTF data is already accessible somewhere on your file system. This is a public, static, final instance of MWCtfSource.

  - MWCtfFileSource — Indicates that the CTF data resides within a particular file location that you specify. This class takes a java.io.File object in its constructor.

  - MWCtfDirectorySource — Indicates a directory to be scanned when instantiating the component: if a file with a .ctf suffix is found in the

directory you supply, the CTF archive bits are loaded from that file. This class takes a `java.io.File` object in its constructor.

- **•** `MWCtfStreamSource` — Allows CTF bits to be read and extracted directly from a specified input stream. This class takes a `java.io.InputStream` object in its constructor.

• `ExtractLocation` — This option specifies where the extracted CTF content is to be located. Since the MCR requires all CTF content be located somewhere on your file system, use the desired `ExtractLocation` option, along with the component type information, to define a unique location. A value for this option is an instance of the class `MWCtfExtractLocation`. An instance of this class can be created by passing a `java.io.File` or `java.lang.String` into the constructor to specify the file system location to be used or one of these predefined, static final instances may be used:

- **•** `MWCtfExtractLocation.EXTRACT_TO_CACHE` — use to indicate that the CTF content is to be placed in the MCR component cache. This is the default setting for releases R2007a and forward (see "How Does the MATLAB® Builder JA Product Use JAR Files?" on page 2-4).

- **•** `MWCtfExtractLocation.EXTRACT_TO_COMPONENT_DIR` — Use when you want to locate the JAR or `.class` files from which the component has been loaded. If the location is found (e.g.: it is on the file system), then the CTF data is extracted into the same directory. This option most closely matches the behavior of R2007a and previous releases.

Setting Options

Use the following methods to get or set the location where the CTF archive may be found for an extracted component:

- `getCtfSource()`
- `setCtfSource( )`

Use the following methods to get or set the location where the extracted CTF content is to be located:

- `getExtractLocation()`
- `setExtractLocation( )`

Example: Enabling MCR Component Cache, Utilitzing CTF Content Already on Your System

If you want to enable the MCR Component Cache for a Java component (in this example, using the user-built Java class MyModel) utilizing CTF content already resident in your file system, instantiate MWComponentOptions using the following statements:

```
MWComponentOptions options = new MWComponentOptions();

// set options for the component by calling setter methods
// on `options'
options.setCtfSource(MWCtfSource.NONE);
    options.setExtractLocation(
            new MWCtfExtractLocation( C:\\readonlydir\\MyModel_mcr ));

// instantiate the component using the desired options
MyModel m = new MyModel(options);
```

Learning About Java Classes and Methods by Exploring the Javadoc

The documentation generated by Sun Microsystems, Inc.'s Javadoc can be a powerful resource when using the MATLAB Builder JA product. The Javadoc can be browsed from any MATLAB Help browser or The MathWorks web site by entering the name of the class or method you want to learn more about in the search field.

Javadoc contains, among other information:

- Signatures that diagram method and class usage
- Parameters passed in, return values expected, and exceptions that can be thrown
- Examples demonstrating typical usage of the class or method

4

Sample Java Applications

Note Remember to double-quote all parts of the `java` command paths that contain spaces. To test directly against the MCR when executing `java`, substitute *mcrroot* for *matlabroot*, where *mcrroot* is the location where the MCR is installed on your system.

Plot Example

Purpose

The purpose of the example is to show you how to do the following:

- Use the MATLAB Builder JA product to convert a MATLAB function (`drawplot`) to a method of a Java class (`plotter`) and wrap the class in a Java component (`plotdemo`).

- Access the component in a Java application (`createplot.java`) by instantiating the `plotter` class and using the `MWArray` class library to handle data conversion.

Note For complete reference information about the `MWArray` class hierarchy, see the `com.mathworks.toolbox.javabuilder` package.

- Build and run the `createplot.java` application.

The `drawplot` function displays a plot of input parameters `x` and `y`.

Procedure

1 If you have not already done so, copy the files for this example as follows:

   a Copy the following directory that ships with MATLAB to your work directory:

       matlabroot\toolbox\javabuilder\Examples\PlotExample

   b At the MATLAB command prompt, `cd` to the new `PlotExample` subdirectory in your work directory.

2 If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 6-3.

3 Write the `drawplot` function as you would any MATLAB function.

The following code defines the `drawplot` function:

```
function drawplot(x,y)
plot(x,y);
```

This code is already in your work directory in `PlotExample\PlotDemoComp\drawplot.m`.

4 While in MATLAB, issue the following command to open the Deployment Tool Window:

```
deploytool
```

5 In MATLAB, Select **File > New Deployment Project**.

6 In the New Deployment Project dialog box, select **MATLAB Builder JA** and **Java Package**.

7 Select `plotdemo` as the name of the project and click **OK**.

8 In the Deployment Tool, select **plotdemo.class** and right-click. Select **Rename** and type `plotter`.

9 Select **Generate Verbose Output**.

10 Add the `drawplot.m` file to the project.

11 Save the project.

12 Build the component.

13 Write source code for an application that accesses the component.

The sample application for this example is in *matlabroot*`\toolbox\javabuilder\Examples\PlotExample\PlotDemoJavaApp\createplot.java`.

The program graphs a simple parabola from the equation $y = x^2$.

The program listing is shown here.

createplot.java

```
/* createplot.java
 * This file is used as an example for the MATLAB
 * Builder for Java product.
 *
 * Copyright 2001-2007 The MathWorks, Inc.
 */


/* Necessary package imports */
import com.mathworks.toolbox.javabuilder.*;
import plotdemo.*;


/*
 * createplot class demonstrates plotting x-y data into
 * a MATLAB figure window by graphing a simple parabola.
 */
class createplot
{
   public static void main(String[] args)
   {
      MWNumericArray x = null;   /* Array of x values */
      MWNumericArray y = null;   /* Array of y values */
      plotter thePlot = null;    /* Plotter class instance */
      int n = 20;                /* Number of points to plot */

      try
      {
         /* Allocate arrays for x and y values */
         int[] dims = {1, n};
         x = MWNumericArray.newInstance(dims,
            MWClassID.DOUBLE, MWComplexity.REAL);
         y = MWNumericArray.newInstance(dims,
            MWClassID.DOUBLE, MWComplexity.REAL);

         /* Set values so that y = x^2 */
         for (int i = 1; i <= n; i++)
         {
```

```
            x.set(i, i);
            y.set(i, i*i);
        }

        /* Create new plotter object */
        thePlot = new plotter();

        /* Plot data */
        thePlot.drawplot(x, y);
        thePlot.waitForFigures();
    }

    catch (Exception e)
    {
        System.out.println("Exception: " + e.toString());
    }

    finally
    {
        /* Free native resources */
        MWArray.disposeArray(x);
        MWArray.disposeArray(y);
        if (thePlot != null)
            thePlot.dispose();
    }
  }
}
```

The program does the following:

- Creates two arrays of double values, using `MWNumericArray` to represent the data needed to plot the equation.

- Instantiates the `plotter` class as `thePlot` object, as shown:

    ```
    thePlot = new plotter();
    ```

- Calls the `drawplot` method to plot the equation using the MATLAB `plot` function, as shown:

    ```
    thePlot.drawplot(x,y);
    ```

- Uses a `try-catch` block to catch and handle any exceptions.

14 Compile the `createplot` application using `javac`. When entering this command, ensure there are no spaces between path names in the *matlabroot* argument. For example, there should be no space between `javabuilder.jar;` and `.\distrib\plotdemo.jar` in the following example. `cd` to your work directory. Ensure `createplot.java` is in your work directory

- On Windows, execute this command:

```
javac -classpath
  .;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
  .\distrib\plotdemo.jar createplot.java
```

- On UNIX, execute this command:

```
javac -classpath
  .:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
  ./distrib/plotdemo.jar createplot.java
```

15 Run the application.

To run the `createplot.class` file, do one of the following:

- On Windows, type:

```
java -classpath
  .;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
  .\distrib\plotdemo.jar
  createplot
```

- On UNIX, type:

```
java -classpath
  .:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
  ./distrib/plotdemo.jar
   createplot
```

> **Note** The supported JRE version is 1.6.0. To find out what JRE
> you are using, refer to the output of `'version -java'` in MATLAB
> or refer to the `jre.cfg` file in *matlabroot*/sys/java/jre/*arch* or
> *mcrroot*/sys/java/jre/*arch*.

> **Note** If you are running on Solaris™ 64-bit platforms, you must add the
> `-d64` flag in the Java command. See "Limitations and Restrictions" on page
> 6-2 for more specific information.

The createplot program should display the output.
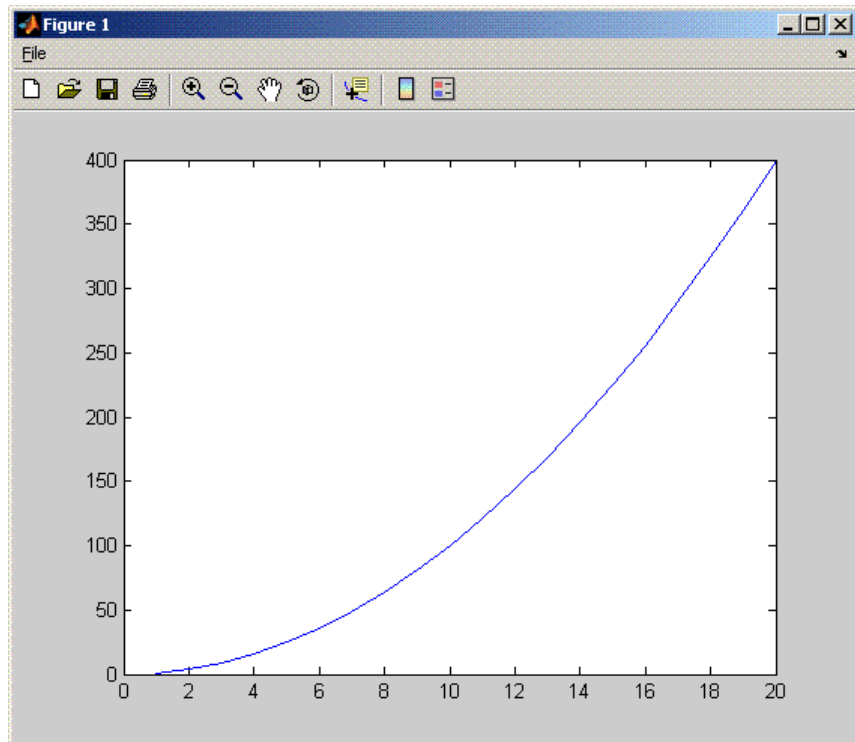
Spectral Analysis Example

Purpose

The purpose of the example is to show you the following:

- How to use the MATLAB Builder JA product to create a component (`spectralanalysis`) containing a class that has a private method that is automatically encapsulated

- How to access the component in a Java application (`powerspect.java`), including use of the `MWArray` class hierarchy to represent data

> **Note** For complete reference information about the `MWArray` class hierarchy, see the `com.mathworks.toolbox.javabuilder` package.

- How to build and run the application

The component `spectralanalysis` analyzes a signal and graphs the result. The class, `fourier`, performs a Fast Fourier Transform (FFT) on an input data array. A method of this class, `computefft`, returns the results of that FFT as two output arrays—an array of frequency points and the power spectral density. The second method, `plotfft`, graphs the returned data. These two methods, `computefft` and `plotfft`, encapsulate MATLAB functions.

The MATLAB code for these two methods is in `computefft.m` and `plotfft.m`, which can be found in *matlabroot*\toolbox\javabuilder\Examples\SpectraExample\SpectraDemoComp.

computefft.m

```
function [fftData, freq, powerSpect] = ComputeFFT(data, interval)
%   COMPUTEFFT Computes the FFT and power spectral density.
%   [FFTDATA, FREQ, POWERSPECT] = COMPUTEFFT(DATA, INTERVAL)
%   Computes the FFT and power spectral density of the input data.
%   This file is used as an example for MATLAB Builder NE
%   product.
%   Copyright 2001-2007 The MathWorks, Inc.
```

```
if (isempty(data))
   fftdata = [];
   freq = [];
   powerspect = [];
   return;
end
if (interval <= 0)
   error('Sampling interval must be greater then zero');
   return;
end
fftData = fft(data);
freq = (0:length(fftData)-1)/(length(fftData)*interval);
powerSpect = abs(fftData)/(sqrt(length(fftData)));
```

plotfft.m

```
function PlotFFT(fftData, freq, powerSpect)
%PLOTFFT Computes and plots the FFT and power spectral density.
%   [FFTDATA, FREQ, POWERSPECT] = PLOTFFT(DATA, INTERVAL)
%   Computes the FFT and power spectral density of the input data.
%   This file is used as an example for MATLAB Builder NE
%   product.
%   Copyright 2001-2007 The MathWorks, Inc.
len = length(fftData);
    if (len <= 0)
        return;
    end
    plot(freq(1:floor(len/2)), powerSpect(1:floor(len/2)))
    xlabel('Frequency (Hz)'), grid on
    title('Power spectral density')
```

Procedure

1 If you have not already done so, copy the files for this example as follows:

   a Copy the following directory that ships with MATLAB to your work directory:

      matlabroot\toolbox\javabuilder\Examples\SpectraExample

b At the MATLAB command prompt, cd to the new SpectraExample subdirectory in your work directory.

2 If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 6-3.

3 Write the M-code that you want to access.

This example uses computefft.m and plotfft.m, which are already in your work directory in SpectraExample\SpectraDemoComp.

4 While in MATLAB, issue the following command to open the Deployment Tool window:

deploytool

5 In MATLAB, Select **File > New Deployment Project**.

6 In the New Deployment Project dialog box, select **MATLAB Builder JA** and **Java Package**.

7 Select spectralanalysis as the name of the project and click **OK**.

8 In the Deployment Tool, select **spectralanalysis.class** and right-click. Select **Rename** and type fourier.

9 Select **Generate Verbose Output**.

10 Add the plotfft.m M-file to the project.

> **Note** In this example, the application that uses the fourier class does not need to call computefft directly. The computefft method is required only by the plotfft method. Thus, when creating the component, you do not need to add the computefft function, although doing so does no harm.

11 Save the project. Make note of the project directory because you will refer to it later when you build the program that will use it.

12 Build the component.

13 Write source code for an application that accesses the component.

The sample application for this example is in
`SpectraExample\SpectraDemoJavaApp\powerspect.java`.

The program listing is shown here.

powerspect.java

```
/* powerspect.java
 * This file is used as an example for the MATLAB
 * Builder for Java product.
 *
 * Copyright 2001-2007 The MathWorks, Inc.
 */


/* Necessary package imports */
import com.mathworks.toolbox.javabuilder.*;
import spectralanalysis.*;


/*
 * powerspect class computes and plots the power
 * spectral density of an input signal.
 */
class powerspect
{
   public static void main(String[] args)
   {
      double interval = 0.01;    /* Sampling interval */
      int nSamples = 1001;       /* Number of samples */
      MWNumericArray data = null; /* Stores input data */
      Object[] result = null;    /* Stores result */
      fourier theFourier = null;  /* Fourier class instance */

      try
      {
         /*
          * Construct input data as sin(2*PI*15*t) +
          * sin(2*PI*40*t) plus a random signal.
          *    Duration = 10
```

```
 *     Sampling interval = 0.01
 */
int[] dims = {1, nSamples};
data = MWNumericArray.newInstance(dims, MWClassID.DOUBLE,
                                       MWComplexity.REAL);
for (int i = 1; i <= nSamples; i++)
{
   double t = (i-1)*interval;
   double x = Math.sin(2.0*Math.PI*15.0*t) +
      Math.sin(2.0*Math.PI*40.0*t) +
      Math.random();
   data.set(i, x);
}

/* Create new fourier object */
theFourier = new fourier();
theFourier.waitForFigures();

/* Compute power spectral density and plot result */
result = theFourier.plotfft(3, data,
   new Double(interval));
}

catch (Exception e)
{
   System.out.println("Exception: " + e.toString());
}

finally
{
   /* Free native resources */
   MWArray.disposeArray(data);
   MWArray.disposeArray(result);
   if (theFourier != null)
      theFourier.dispose();
}
}
}
```

The program does the following:

- Constructs an input array with values representing a random signal
  with two sinusoids at 15 and 40 Hz embedded inside of it

- Creates an `MWNumericArray` array that contains the data, as shown:

  ```
  data = MWNumericArray.newInstance(dims, MWClassID.DOUBLE, MWComplexity.REAL);
  ```

- Instantiates a `fourier` object

- Calls the `plotfft` method, which calls `computeftt` and plots the data

- Uses a `try-catch` block to handle exceptions

- Frees native resources using `MWArray` methods

14 Compile the `powerspect.java` application using `javac`. When entering
this command, ensure there are no spaces between path names in the
matlabroot argument. For example, there should be no space between
`javabuilder.jar;` and `.\distrib\spectralanalysis.jar` in the
following example.

Open a Command Prompt window and `cd` to the
matlabroot`\spectralanalysis` directory. `cd` to your work directory.
Ensure `powerspect.java` is in your work directory

- On Windows, execute the following command:

  ```
  javac -classpath
  .;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
  .\distrib\spectralanalysis.jar powerspect.java
  ```

- On UNIX, execute the following command:

  ```
  javac -classpath
  .:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
  ./distrib/spectralanalysis.jar powerspect.java
  ```

> **Note** For *matlabroot* substitute the MATLAB root directory on your system. Type matlabroot to see this directory name.

15 Run the application.

- On Windows, execute the powerspect class file:

```
java -classpath
.;matlabroot\toolbox\javabuilder\jar\javabuilder.jar
.\distrib\spectralanalysis.jar
powerspect
```

- On UNIX, execute the powerspect class file:

```
java -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
./distrib/spectralanalysis.jar
powerspect
% where <Arch> = glux86 gluxa64 sol64
```

> **Note** The supported JRE version is 1.6.0. To find out what JRE you are using, refer to the output of 'version -java' in MATLAB or refer to the jre.cfg file in *matlabroot*/sys/java/jre/*arch* or *mcrroot*/sys/java/jre/*arch*.

> **Note** If you are running on Solaris 64-bit platforms, you must add the -d64 flag in the Java command. See "Limitations and Restrictions" on page 6-2 for more specific information.

The powerspect program should display the output:

Matrix Math Example

Purpose

The purpose of the example is to show you the following:

- How to assign more than one MATLAB function to a component class.

- How to manually handle native memory management.

- How to access the component in a Java application (`getfactor.java`) by instantiating `Factor` and using the `MWArray` class library to handle data conversion.

Note For complete reference information about the `MWArray` class hierarchy, see the `com.mathworks.toolbox.javabuilder` package.

- How to build and run the `MatrixMathDemoApp` application

This example builds a Java component to perform matrix math. The example creates a program that performs Cholesky, LU, and QR factorizations on a simple tridiagonal matrix (finite difference matrix) with the following form:

```
A = [ 2 -1  0  0  0
     -1  2 -1  0  0
      0 -1  2 -1  0
      0  0 -1  2 -1
      0  0  0 -1  2 ]
```

You supply the size of the matrix on the command line, and the program constructs the matrix and performs the three factorizations. The original

matrix and the results are printed to standard output. You may optionally perform the calculations using a sparse matrix by specifying the string "sparse" as the second parameter on the command line.

MATLAB Functions to Be Encapsulated

The following code defines the MATLAB functions used in the example:

cholesky.m

```
function [L] = cholesky(A)
%CHOLESKY Cholesky factorization of A.
%   L = CHOLESKY(A) returns the Cholesky factorization of A.
%   This file is used as an example for the MATLAB
%   Builder for Java product.

%   Copyright 2001-2007 The MathWorks, Inc.

L = chol(A);
```

ludecomp.m

```
function [L,U] = ludecomp(A)
%LUDECOMP LU factorization of A.
%   [L,U] = LUDECOMP(A) returns the LU factorization of A.
%   This file is used as an example for the MATLAB
%   Builder for Java product.

%   Copyright 2001-2007 The MathWorks, Inc.

[L,U] = lu(A);
```

qrdecomp.m

```
function [Q,R] = qrdecomp(A)
%QRDECOMP QR factorization of A.
%   [Q,R] = QRDECOMP(A) returns the QR factorization of A.
%   This file is used as an example for the MATLAB
%   Builder for Java product.

%   Copyright 2001-2007 The MathWorks, Inc.
```

```
[Q,R] = qr(A);
```

Understanding the getfactor Program

The `getfactor` program takes one or two arguments from standard input. The first argument is converted to the integer order of the test matrix. If the string `sparse` is passed as the second argument, a sparse matrix is created to contain the test array. The Cholesky, LU, and QR factorizations are then computed and the results are displayed to standard output.

The main method has three parts:

- The first part sets up the input matrix, creates a new factor object, and calls the `cholesky`, `ludecomp`, and `qrdecomp` methods. This part is executed inside of a `try` block. This is done so that if an exception occurs during execution, the corresponding `catch` block will be executed.

- The second part is the `catch` block. The code prints a message to standard output to let the user know about the error that has occurred.

- The third part is a `finally` block to manually clean up native resources before exiting.

Procedure

Step-by-Step Procedure

1 If you have not already done so, copy the files for this example as follows:

   a Copy the following directory that ships with MATLAB to your work directory:

   matlabroot\toolbox\javabuilder\Examples\MatrixMathExample

   b At the MATLAB command prompt, `cd` to the new `MatrixMathExample` subdirectory in your work directory.

2 If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 6-3.

3 Write the MATLAB functions as you would any MATLAB function.

The code for the `cholesky`, `ludecomp`, and `qrdecomp` functions is already in your work directory in `MatrixMathExample\MatrixMathDemoComp\`.

4 While in MATLAB, issue the following command to open the Deployment Tool window:

`deploytool`

5 In MATLAB, select **File > New Deployment Project**.

6 In the New Deployment Project dialog box, select **MATLAB Builder JA** and **Java Package**.

7 Select `factormatrix` as the name of the project and click **OK**.

8 In the Deployment Tool, select **factormatrix.class** and right-click. Select **Rename** and type `factor`.

9 Select **Generate Verbose Output**.

10 Add the `cholesky.m`, `ludecomp.m`, and `qrdecomp.m` M-files to the project.

11 Save the project.

12 Build the component by clicking the build icon on the toolbar in Deployment Tool.

13 Write source code for an application that accesses the component.

The sample application for this example is in `MatrixMathExample\MatrixMathDemoJavaApp\getfactor.java`.

The program listing is shown here.

getfactor.java

```
/* getfactor.java
 * This file is used as an example for the MATLAB
 * Builder for Java product.
 *
```

```
 * Copyright 2001-2007 The MathWorks, Inc.
 */

/* Necessary package imports */
import com.mathworks.toolbox.javabuilder.*;
import factormatrix.*;

/*
 * getfactor class computes cholesky, LU, and QR
 * factorizations of a finite difference matrix
 * of order N. The value of N is passed on the
 * command line. If a second command line arg
 * is passed with the value of "sparse", then
 * a sparse matrix is used.
 */
class getfactor
{
    public static void main(String[] args)
    {
        MWNumericArray a = null;    /* Stores matrix to factor */
        Object[] result = null;     /* Stores the result */
        factor theFactor = null;    /* Stores factor class instance */

        try
        {
            /* If no input, exit */
            if (args.length == 0)
            {
                System.out.println("Error: must input a positive integer");
                return;
            }

            /* Convert input value */
            int n = Integer.valueOf(args[0]).intValue();

            if (n <= 0)
            {
                System.out.println("Error: must input a positive integer");
                return;
            }
```

```
/*
 * Allocate matrix. If second input is "sparse"
 * allocate a sparse array
 */
int[] dims = {n, n};

if (args.length > 1 && args[1].equals("sparse"))
   a = MWNumericArray.newSparse(dims[O], dims[1],n+2*(n-1), MWClassID.DOUBLE, MWComplexity.REAL);
else
   a = MWNumericArray.newInstance(dims,MWClassID.DOUBLE, MWComplexity.REAL);

/* Set matrix values */
int[] index = {1, 1};

for (index[0] = 1; index[0] <= dims[0]; index[0]++)
{
   for (index[1] = 1; index[1] <= dims[1]; index[1]++)
   {
      if (index[1] == index[0])
         a.set(index, 2.0);
      else if (index[1] == index[0]+1 || index[1] == index[0]-1)
         a.set(index, -1.0);
   }
}

/* Create new factor object */
theFactor = new factor();

/* Print original matrix */
System.out.println("Original matrix:");
System.out.println(a);

/* Compute cholesky factorization and print results. */
result = theFactor.cholesky(1, a);
System.out.println("Cholesky factorization:");
System.out.println(result[O]);
MWArray.disposeArray(result);

/* Compute LU factorization and print results. */
```

```
        result = theFactor.ludecomp(2, a);
        System.out.println("LU factorization:");
        System.out.println("L matrix:");
        System.out.println(result[0]);
        System.out.println("U matrix:");
        System.out.println(result[1]);
        MWArray.disposeArray(result);

        /* Compute QR factorization and print results. */
        result = theFactor.qrdecomp(2, a);
        System.out.println("QR factorization:");
        System.out.println("Q matrix:");
        System.out.println(result[0]);
        System.out.println("R matrix:");
        System.out.println(result[1]);
    }

    catch (Exception e)
    {
        System.out.println("Exception: " + e.toString());
    }

    finally
    {
        /* Free native resources */
        MWArray.disposeArray(a);
        MWArray.disposeArray(result);
        if (theFactor != null)
            theFactor.dispose();
    }
  }
}
```

The statement:

```
    theFactor = new factor();
```

creates an instance of the class `factor`.

The following statements call the methods that encapsulate the MATLAB functions:

```
result = theFactor.cholesky(1, a);
...
result = theFactor.ludecomp(2, a);
...
result = theFactor.qrdecomp(2, a);
...
```

14 Compile the `getfactor` application using `javac`. When entering this command, ensure there are no spaces between path names in the *matlabroot* argument. For example, there should be no space between `javabuilder.jar;` and `.\distrib\factormatrix.jar` in the following example.

`cd` to the *matlabroot*`\factormatrix` directory. Ensure `getfactor.java` is in this directory.

- On Windows, execute the following command:

```
javac -classpath
.;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
.\distrib\factormatrix.jar getfactor.java
```

- On UNIX, execute the following command:

```
javac -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
./distrib/factormatrix.jar getfactor.java
```

15 Run the application.

Run `getfactor` using a nonsparse matrix

- On Windows, execute the `getfactor` class file as follows:

```
java -classpath
.;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
.\distrib\factormatrix.jar
getfactor 4
```

- On UNIX, execute the getfactor class file as follows:

```
java -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
./distrib/factormatrix.jar
getfactor 4
% where <Arch> = glux86 gluxa64 sol64
```

Note The supported JRE version is 1.6.0. To find out what JRE you are using, refer to the output of 'version -java' in MATLAB or refer to the jre.cfg file in *matlabroot*/sys/java/jre/<arch> or *mcrroot*/sys/java/jre/<arch>.

Note If you are running on Solaris 64-bit platforms, you must add the -d64 flag in the Java command. See "Limitations and Restrictions" on page 6-2 for more specific information.

Output for the Matrix Math Example

```
Original matrix:
     2    -1     0     0
    -1     2    -1     0
     0    -1     2    -1
     0     0    -1     2
Cholesky factorization:
    1.4142   -0.7071        0        0
         0    1.2247   -0.8165        0
         0         0    1.1547   -0.8660
         0         0        0    1.1180


LU factorization:
L matrix:
    1.0000        0        0        0
   -0.5000    1.0000        0        0
         0   -0.6667    1.0000        0
```

```
            0         0   -0.7500    1.0000


U matrix:
       2.0000   -1.0000         0         0
            0    1.5000   -1.0000         0
            0         0    1.3333   -1.0000
            0         0         0    1.2500


QR factorization:
Q matrix:
      -0.8944   -0.3586   -0.1952    0.1826
       0.4472   -0.7171   -0.3904    0.3651
            0    0.5976   -0.5855    0.5477
            0         0    0.6831    0.7303


R matrix:
      -2.2361    1.7889   -0.4472         0
            0   -1.6733    1.9124   -0.5976
            0         0   -1.4639    1.9518
            0         0         0    0.9129
```

To run the same program for a sparse matrix, use the same command and add the string sparse to the command line:

```
java (... same arguments) getfactor 4 sparse
```

Output for a Sparse Matrix

```
Original matrix:
    (1,1)        2
    (2,1)       -1
    (1,2)       -1
    (2,2)        2
    (3,2)       -1
    (2,3)       -1
    (3,3)        2
    (4,3)       -1
```

```
     (3,4)         -1
     (4,4)          2


Cholesky factorization:
     (1,1)       1.4142
     (1,2)      -0.7071
     (2,2)       1.2247
     (2,3)      -0.8165
     (3,3)       1.1547
     (3,4)      -0.8660
     (4,4)       1.1180


LU factorization:
L matrix:
     (1,1)       1.0000
     (2,1)      -0.5000
     (2,2)       1.0000
     (3,2)      -0.6667
     (3,3)       1.0000
     (4,3)      -0.7500
     (4,4)       1.0000


U matrix:
     (1,1)       2.0000
     (1,2)      -1.0000
     (2,2)       1.5000
     (2,3)      -1.0000
     (3,3)       1.3333
     (3,4)      -1.0000
     (4,4)       1.2500


QR factorization:
Q matrix:
     (1,1)       0.8944
     (2,1)      -0.4472
     (1,2)       0.3586
```

```
                (2,2)        0.7171
                (3,2)       -0.5976
                (1,3)        0.1952
                (2,3)        0.3904
                (3,3)        0.5855
                (4,3)       -0.6831
                (1,4)        0.1826
                (2,4)        0.3651
                (3,4)        0.5477
                (4,4)        0.7303


        R matrix:
                (1,1)        2.2361
                (1,2)       -1.7889
                (2,2)        1.6733
                (1,3)        0.4472
                (2,3)       -1.9124
                (3,3)        1.4639
                (2,4)        0.5976
                (3,4)       -1.9518
                (4,4)        0.9129
```

Phonebook Example

In this section...
"Purpose" on page 4-28
"Procedure" on page 4-28

Purpose

The makephone function takes a structure array as an input, modifies it, and supplies the modified array as an output.

Note For complete reference information about the MWArray class hierarchy, see the com.mathworks.toolbox.javabuilder package.

Procedure

1 If you have not already done so, copy the files for this example as follows:

   a Copy the following directory that ships with MATLAB to your work directory:

   matlabroot\toolbox\javabuilder\Examples\PhoneExample

   b At the MATLAB command prompt, cd to the new PhoneExample subdirectory in your work directory.

2 If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 6-3.

3 Write the makephone function as you would any MATLAB function.

   The following code defines the makephone function:

```
function book = makephone(friends)
%MAKEPHONE Add a structure to a phonebook structure
%   BOOK = MAKEPHONE(FRIENDS) adds a field to its input structure.
```

```
%   The new field EXTERNAL is based on the PHONE field of the original.
%   This file is used as an example for MATLAB
%   Builder for Java.

%   Copyright 2006-2007 The MathWorks, Inc.

book = friends;
for i = 1:numel(friends)
    numberStr = num2str(book(i).phone);
    book(i).external = ['(508) 555-' numberStr];
end
```

This code is already in your work directory in
PhoneExample\PhoneDemoComp\makephone.m.

4 While in MATLAB, issue the following command to open the Deployment
Tool window:

deploytool

5 In MATLAB, select **File > New Deployment Project**.

6 In the New Deployment Project dialog box, select **MATLAB Builder JA**
and **Java Package**.

7 Select phonebookdemo as the name of the project and click **OK**.

8 In the Deployment Tool, select **phonebookdemo.class** and right-click.
Select **Rename** and type phonebook.

9 Select **Generate Verbose Output**.

10 Add the makephone.m file to the project.

11 Save the project.

12 Build the component.

13 Write source code for an application that accesses the component.

The sample application for this example is in
matlabroot\toolbox\javabuilder\Examples\PhoneExample\
PhoneDemoJavaApp\getphone.java.

The program defines a structure array containing names and phone
numbers, modifies it using a MATLAB function, and displays the resulting
structure array.

The program listing is shown here.

getphone.java

```
/* getphone.java
%   This file is used as an example for MATLAB
%   Builder for Java.
*
* Copyright 2001-2007 The MathWorks, Inc.
*/


/* Necessary package imports */
import com.mathworks.toolbox.javabuilder.*;

import phonebookdemo.*;

/*
* getphone class demonstrates the use of the MWStructArray class
*/
class getphone
{
    public static void main(String[] args)
    {
        phonebook thePhonebook = null;     /* Stores magic class instance */
        MWStructArray friends = null; /* Sample input data */
        Object[] result = null;    /* Stores the result */
        MWStructArray book = null; /* Output data extracted from result */

        try
        {
            /* Create new magic object */
            thePhonebook = new phonebook();
```

```
    /* Create an MWStructArray with two fields */
    String[] myFieldNames = {"name", "phone"};
    friends = new MWStructArray(2,2,myFieldNames);

    /* Populate struct with some sample data --- friends and phone numbers */
    friends.set("name",1,new MWCharArray("Jordan Robert"));
    friends.set("phone",1,3386);
    friends.set("name",2,new MWCharArray("Mary Smith"));
    friends.set("phone",2,3912);
    friends.set("name",3,new MWCharArray("Stacy Flora"));
    friends.set("phone",3,3238);
    friends.set("name",4,new MWCharArray("Harry Alpert"));
    friends.set("phone",4,3077);

    /* Show some of the sample data */
    System.out.println("Friends: ");
    System.out.println(friends.toString());

    /* Pass it to an M-function that determines external phone number */
    result = thePhonebook.makephone(1, friends);
    book = (MWStructArray)result[0];
    System.out.println("Result: ");
    System.out.println(book.toString());

    /* Extract some data from the returned structure */
    System.out.println("Result record 2:");
    System.out.println(book.getField("name",2));
    System.out.println(book.getField("phone",2));
    System.out.println(book.getField("external",2));

    /* Print the entire result structure using the helper function below */
    System.out.println("");
    System.out.println("Entire structure:");
    dispStruct(book);
}
catch (Exception e)
{
    System.out.println("Exception: " + e.toString());
}
```

```
    finally
    {
        /* Free native resources */
        MWArray.disposeArray(result);
        MWArray.disposeArray(friends);
        MWArray.disposeArray(book);
        if (thePhonebook != null)
            thePhonebook.dispose();
    }
}


public static void dispStruct(MWStructArray arr) {
 System.out.println("Number of Elements: " + arr.numberOfElements());
 //int numDims = arr.numberOfDimensions();
 int[] dims = arr.getDimensions();
 System.out.print("Dimensions: " + dims[0]);
 for (int i = 1; i < dims.length; i++)
 {
  System.out.print("-by-" + dims[i]);
 }
 System.out.println("");
 System.out.println("Number of Fields: " + arr.numberOfFields());
 System.out.println("Standard MATLAB view:");
 System.out.println(arr.toString());
 System.out.println("Walking structure:");
    java.lang.String[] fieldNames = arr.fieldNames();
    for (int element = 1; element <= arr.numberOfElements(); element++) {
        System.out.println("Element " + element);
        for (int field = 0; field < arr.numberOfFields(); field++) {
         MWArray fieldVal = arr.getField(fieldNames[field], element);
         /* Recursively print substructures, give string display of other classes */
         if (fieldVal instanceof MWStructArray)
         {
             System.out.println("   " + fieldNames[field] + ": nested structure:");
             System.out.println("+++ Begin of \"" + fieldNames[field] + "\" nested structure");
          dispStruct((MWStructArray)fieldVal);
             System.out.println("+++ End of \"" + fieldNames[field] + "\" nested structure");
         } else {
             System.out.print("   " + fieldNames[field] + ": ");
```

```
            System.out.println(fieldVal.toString());
        }
    }
  }
 }
}
```

The program does the following:

- Creates a structure array, using MWStructArray to represent the example phonebook data.

- Instantiates the plotter class as thePhonebook object, as shown:

  thePhonebook = new phonebook();

- Calls the makephone method to create a modified copy of the structure by adding an additional field, as shown:

  result = thePhonebook.makephone(1, friends);

- Uses a try-catch block to catch and handle any exceptions.

14 Compile the getphone application using javac. When entering this command, ensure there are no spaces between path names in the *matlabroot* argument. For example, there should be no space between javabuilder.jar; and .\distrib\phonebookdemo.jar in the following example. cd to your work directory. Ensure getphone.java is in your work directory

- On Windows, execute this command:

  ```
  javac -classpath
    .;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
    .\distrib\phonebookdemo.jar getphone.java
  ```

- On UNIX, execute this command:

  ```
  javac -classpath
    .:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
    ./distrib/phonebookdemo.jar getphone.java
  ```

15 Run the application.

To run the `getphone.class` file, do one of the following:

- On Windows, type:

```
java -classpath
  .;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
  .\distrib\phonebookdemo.jar
  getphone
```

- On UNIX, type:

```
java -classpath
  .:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
  ./distrib/phonebookdemo.jar
   getphone
```

Note The supported JRE version is 1.6.0. To find out what JRE you are using, refer to the output of `'version -java'` in MATLAB or refer to the `jre.cfg` file in *matlabroot*/sys/java/jre/*arch* or *mcrroot*/sys/java/jre/*arch*.

Note If you are running on Solaris 64-bit platforms, you must add the `-d64` flag in the Java command. See "Limitations and Restrictions" on page 6-2 for more specific information.

The `getphone` program should display the output:

```
Friends:
2x2 struct array with fields:
    name
    phone
Result:
2x2 struct array with fields:
    name
    phone
```

```
        external
Result record 2:
Mary Smith
3912
(508) 555-3912

Entire structure:
Number of Elements: 4
Dimensions: 2-by-2
Number of Fields: 3
Standard MATLAB view:
2x2 struct array with fields:
    name
    phone
    external
Walking structure:
Element 1
   name: Jordan Robert
   phone: 3386
   external: (508) 555-3386
Element 2
   name: Mary Smith
   phone: 3912
   external: (508) 555-3912
Element 3
   name: Stacy Flora
   phone: 3238
   external: (508) 555-3238
Element 4
   name: Harry Alpert
   phone: 3077
   external: (508) 555-3077
```

Buffered Image Creation Example

Purpose

This example demonstrates how to create a buffered image from a graphical representation of the surf(peaks) function in MATLAB.

The hardcopy function is used to output the figure window as an array:

```
function w = getSurfsFigure
 f = figure;
 set(f,'Visible', 'off');

 f = surf(peaks);
 w = hardcopy(gcf,'-dOpenGL','-r0');
end
```

Note There is minimal error handling in this example. Integrate this code with whatever logging is currently in place for your Java layer.

Procedure

1 Create a Java object from the function above by doing the following:

a Start the Deployment Tool by entering deploytool from the MATLAB command prompt.

b Select **File > New > Deployment Project** from the MATLAB interface.

c Select a **MATLAB Builder JA Project** and **Java Package**, then give the project a name and location. Click **OK**.

d Click **Settings** in the Deployment Tool and enter the package name as com.mathworks.deploy.peaks. Click **OK**.

e In the Deployment Tool, change the *project_name* **class** name to Peaks by right-clicking the class folder and selecting **Rename**.

f Add the function getSurfsFigure.m to the Peaks class by dragging the function from the MATLAB Current Directory browser to the **Peaks** class folder in the Deployment Tool.

g Click ⚒ on the Deployment Tool toolbar and build your Java object. Be sure to select the **Include MCR** build option.

2 From the output **distrib** directory of your build, copy `peaks.jar` to the directory where you are building your application.

3 Create the `SurfPeaks.java` code:

```
//imported classes from JRE
import java.awt.Image;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.ImageIcon;

//imported classes from javabuilder.jar
import com.mathworks.toolbox.javabuilder.Images;
import com.mathworks.toolbox.javabuilder.MWArray;
import com.mathworks.toolbox.javabuilder.MWNumericArray;
import com.mathworks.toolbox.javabuilder.MWException;

//Import for the Deployment Project peaks.jar
import com.mathworks.deploy.peaks.Peaks;

//The goal of this class is to show how you would deal with
//  static images coming from a java deployment of a MATLAB figure.
//This can be run as a standalone java application.
//
//The MATLAB function being deployed is surf(peaks).
//The hardcopy function is used to
//  output the figure window as an array.
//M Code:
//*******************************************
//    function w = getSurfsFigure
//        f = figure;
//        surf(peaks);
//        w = hardcopy(gcf,'-dOpenGL','-r0');
//        close(f);
//    end
//*******************************************
//
```

```
//For this example you must have the deployment project
// jar and the javabuilder.jar on your classpath.
//
//Note:
//For this example there is minimal error handling.
//Typically you would want to integrate this with whatever
// logging is currently in place for your java layer.
public class SurfPeaks
{
    //This initializes and executes the JFrame.
    public static void main(String args[])
    {
        ImageIcon icon = new ImageIcon(getSurfImage());
        JLabel label =  new JLabel(icon);
        JFrame frame = new JFrame();
        frame.setSize(icon.getIconWidth(),icon.getIconHeight());
        frame.setContentPane(label);
        frame.setVisible(true);
    }

    //This method is basically our "business logic" method.
    //It is responsible for instantiating our MATLAB deployment,
    //passing in any needed inputs, and dealing with any outputs.
    //In this example we have no inputs, and the only output is the
    //figure in hardcopy format (three dimensional MWNumericArray).
    private static Image getSurfImage()
    {
        try
        {
            //Our deployment uses native resources and
            //should be disposed of as soon as possible.
            Peaks matlabModel = new Peaks();
            try
            {
                //If we had any inputs to our method
                //they would be passed in here.
                Object[] results = matlabModel.getSurfsFigure(1);

                //This array uses native resources and
                //should be disposed of as soon as possible.
```

```
                    MWArray mwArray = (MWArray)results[0];
                    try
                    {
                        //Since we want this method to return only
                        //     non MATLAB data
                        //  we convert the MATLAB figure to a
                        //     buffered image and return it.
                        return Images.renderArrayData
                                ((MWNumericArray)mwArray);
                    }
                    finally
                    {
                        MWArray.disposeArray(mwArray);
                    }
                }
                finally
                {
                    matlabModel.dispose();
                }
            }
            catch(MWException mwe)
            {
                mwe.printStackTrace();
                return null;
            }
        }
    }
```

4 Compile the program using `javac` and the following command:

```
javac -classpath javabuilder.jar;peaks.jar SurfPeaks.java
```

Ensure that `javabuilder.jar` and `peaks.jar` (compiled earlier) are both in the directory you compile from (or define their full paths).

5 Run `SurfPeaks.class` using the following `java` command. Ensure that `javabuilder.jar` and `peaks.jar` (compiled earlier) are both in the directory you compiled in (or define their full paths).

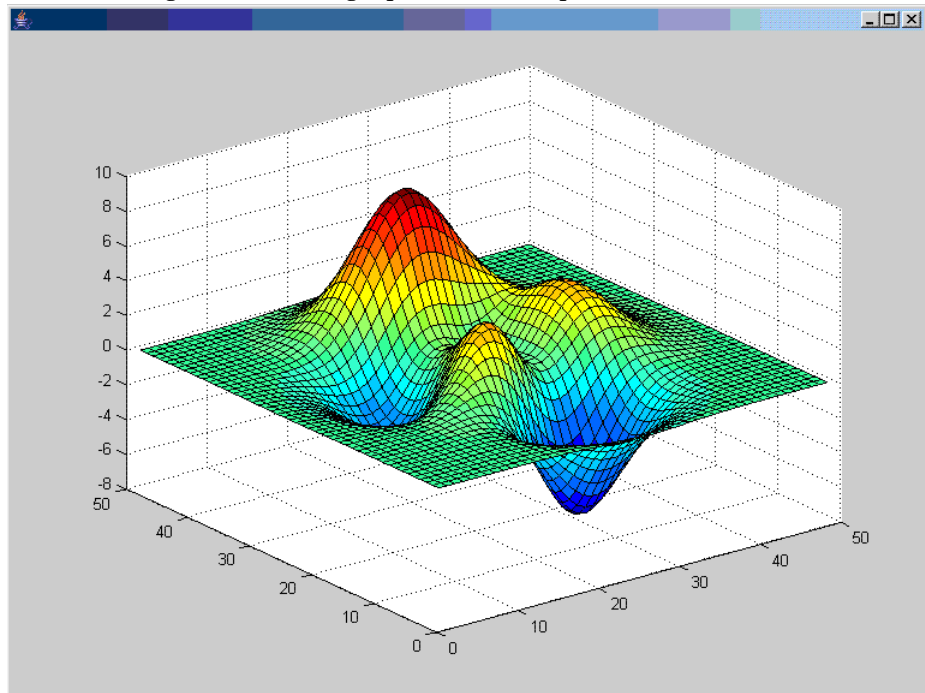```
java -classpath javabuilder.jar;peaks.jar;. SurfPeaks
```

**Note** If you are running on Solaris 64-bit platforms, you must add the -d64 flag in the Java command. See "Limitations and Restrictions" on page 6-2 for more specific information.

The following Surf Peaks graphic should open:

# Optimization Example

## Purpose

This example shows how to:

- Use the MATLAB Builder JA product to create a component (`OptimDemo`) that applies MATLAB optimization routines to objective functions implemented as Java objects.

- Access the component in a Java application (`PerformOptim.java`), including use of the `MWJavaObjectRef` class to create a reference to a Java object (`BananaFunction.java`) and pass it to the component.

  **Note** For complete reference information about the `MWArray` class hierarchy, see the `com.mathworks.toolbox.javabuilder` Javadoc package in the MATLAB Help or on the Web.

- Build and run the application.

## OptimDemo Component

The component `OptimDemo` finds a local minimum of an objective function and returns the minimal location and value. The component uses the MATLAB optimization function fminsearch, and this example optimizes the Rosenbrock banana function used in the fminsearch documentation. The class, Optimizer, performs an unconstrained nonlinear optimization on an objective function implemented as a Java object. A method of this class, doOptim, accepts an initial guess and Java object that implements the objective function, and returns the location and value of a local minimum. The second method, displayObj, is a debugging tool that lists the characteristics of a Java object. These two methods, doOptim and

displayObj, encapsulate MATLAB functions. The MATLAB code for these two methods is in doOptim.m and displayObj.m, which can be found in *matlabroot*\toolbox\javabuilder\Examples\ObjectRefExample\ObjectRefDemoComp.

## Procedure

**1** If you have not already done so, copy the files for this example as follows:

    **a** Copy the following directory that ships with MATLAB to your work directory:
    *matlabroot*\toolbox\javabuilder\Examples\ObjectRefExample

    **b** At the MATLAB command prompt, cd to the new ObjectRefExample subdirectory in your work directory.

**2** If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 6-3.

**3** Write the M-code that you want to access. This example uses doOptim.m and displayObj.m, which are already in your work directory in ObjectRefExample\ObjectRefDemoComp.

For reference, the code of doOptim.m is displayed here:

```
function [x,fval] = doOptim(h, xO)
%DOOPTIM Optimize a Java objective function
%    This file is used as an example for the
%    MATLAB Builder JA product.

% FMINSEARCH can't operate directly on Java objective functions,
% so you must create an anonymous function with the correct
% signature to wrap the Java object.
% Here, we assume our object has a method evaluateFunction()
% that takes an array of doubles and returns a double.
% This could become an Interface,
% and we could check that the object implements that Interface.
mWrapper = @(x) h.evaluateFunction(x);

% Compare two ways of evaluating the objective function
```

```
% These eventually call the same Java method, and return the
% same results.
directEval = h.evaluateFunction(x0)
wrapperEval = mWrapper(x0)

[x,fval] = fminsearch(mWrapper,x0)
```

For reference, the code of displayObj.m is displayed here:

```
function className = displayObj(h)
%DISPLAYOBJ Display information about a Java object
%   This file is used as an example for the
%   MATLAB Builder JA product.

h
className = class(h)
whos('h')
methods(h)
```

4 While in MATLAB, issue the following command to open the Deployment
Tool window:

```
deploytool
```

5 In MATLAB, select **File > New Deployment Project**.

6 In the New Deployment Project dialog, select **MATLAB Builder JA** and
Java Package.

7 Select OptimDemo as the name of the project and click **OK**.

8 In the Deployment Tool, select **OptimDemo.class** and right-click. Select
Rename and type Optimizer.

9 Select **Generate Verbose Output**.

10 Add the doOptim.m and displayObj.m files to the project.

11 Save the project. Make note of the project directory because you will refer
to it later when you build the program that will use it.

12 Build the component.

13 Write source code for a class that implements an object function to optimize. The sample application for this example is in `ObjectRefExample\ObjectRefDemoJavaApp\BananaFunction.java`. The program listing is shown here:

```
/* BananaFunction.java
 * This file is used as an example for the MATLAB
 * Builder JA product.
 *
 * Copyright 2001-2007 The MathWorks, Inc.
 * $Revision: 1.1.4.59 $  $Date: 2008/07/02 04:58:21 $
 */

public class BananaFunction {
 public BananaFunction() {}
 public double evaluateFunction(double[] x)
 {
  /* Implements the Rosenbrock banana function described in
   * the FMINSEARCH documentation
   */
  double term1 = 100*java.lang.Math.pow((x[1]-Math.pow(x[0],2.0)),2.0);
  double term2 =  Math.pow((1-x[0]),2.0);
  return term1 + term2;
 }


}
```

The class implements the Rosenbrock banana function described in the `fminsearch` documentation.

14 Write source code for an application that accesses the component. The sample application for this example is in `ObjectRefExample\ObjectRefDemoJavaApp\PerformOptim.java`. The program listing is shown here:

```
/* PerformOptim.java
 * This file is used as an example for the MATLAB
 * Builder JA product.
 *
 * Copyright 2001-2007 The MathWorks, Inc.
```

```
 * $Revision: 1.1.4.59 $  $Date: 2008/07/02 04:58:21 $
 */

/* Necessary package imports */
import com.mathworks.toolbox.javabuilder.*;
import OptimDemo.*;

/*
 * Demonstrates the use of the MWJavaObjectRef class
 * Takes initial point for optimization as two arguments:
 *     PerformOptim -1.2 1.0
 */
class PerformOptim
{
 public static void main(String[] args)
 {
  Optimizer theOptimizer = null;  /* Stores component
                                              instance */
  MWJavaObjectRef origRef = null;  /* Java object reference to
                                            be passed to component */
  MWJavaObjectRef outputRef = null; /* Output data extracted
                                          from result */
  MWNumericArray x0 = null; /* Initial point for optimization */
  MWNumericArray x = null; /* Location of minimal value */
  MWNumericArray fval = null; /* Minimal function value */
  Object[] result = null; /* Stores the result */


  try
  {
   /* If no input, exit */
   if (args.length < 2)
   {
    System.out.println("Error: must input initial x0_1
                                          and x0_2 position");
    return;
   }

   /* Instantiate a new Builder component object */
   /* This should only be done once per application instance */
```

```
theOptimizer = new Optimizer();

try {
 /* Initial point --- parse data from text fields */
 double[] x0Data = new double[2];
 x0Data[0] = Double.valueOf(args[0]).doubleValue();
 x0Data[1] = Double.valueOf(args[1]).doubleValue();
 x0 = new MWNumericArray(x0Data, MWClassID.DOUBLE);
 System.out.println("Using x0 =");
 System.out.println(x0);

 /* Create object reference to objective function object */
 BananaFunction objectiveFunction = new BananaFunction();
 origRef = new MWJavaObjectRef(objectiveFunction);

 /* Pass Java object to an M-function that lists its
                                        methods, etc */
 System.out.println("*********************************");
 System.out.println("** Properties of Java object   **");
 System.out.println("*********************************");
 result = theOptimizer.displayObj(1, origRef);
 MWArray.disposeArray(result);
 System.out.println("** Finished DISPLAYOBJ **********");

 /* Call the Java component to optimize the function */
 /* using the MATLAB function FMINSEARCH */
 System.out.println("****************************************");
 System.out.println("** Unconstrained nonlinear optimization*");
 System.out.println("****************************************");
 result = theOptimizer.doOptim(2, origRef, x0);
 try {
  System.out.println("** Finished DOOPTIM ****** *********");
  x = (MWNumericArray)result[0];
  fval = (MWNumericArray)result[1];

  /* Display the results of the optimization */
  System.out.println("Location of minimum: ");
  System.out.println(x);
  System.out.println("Function value at minimum: ");
  System.out.println(fval.toString());
```

```
 }
 finally
 {
  MWArray.disposeArray(result);
 }
}
finally
{
 /* Free native resources */
 MWArray.disposeArray(origRef);
 MWArray.disposeArray(outputRef);
 MWArray.disposeArray(xO);
}
}
catch (Exception e)
{
 System.out.println("Exception: " + e.toString());
}

finally
{
 /* Free native resources */
 if (theOptimizer != null)
  theOptimizer.dispose();
}
}
}
```

The program does the following:

- Instantiates an object of the BananaFunction class above to be optimized.

- Creates an MWJavaObjectRef that references the BananaFunction object, as shown: origRef = new MWJavaObjectRef(objectiveFunction);

- Instantiates an Optimizer object

- Calls the displayObj method to verify that the Java object is being passed correctly

- Calls the doOptim method, which uses fminsearch to find a local minimum of the objective function

- Uses a `try/catch` block to handle exceptions

- Frees native resources using `MWArray` methods

15 Compile the `PerformOptim.java` application and `BananaFunction.java` helper class using `javac`. When entering this command, ensure there are no spaces between path names in the *matlabroot* argument. For example, there should be no space between `javabuilder.jar;` and `.\distrib\OptimDemo.jar` in the following example.

a Open a Command Prompt window and `cd` to the *matlabroot*`\ObjectRefExample` directory.

b Compile the application according to which operating system you are running on:

- On Windows, execute this command:

```
javac -classpath
.;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
.\distrib\OptimDemo.jar BananaFunction.java
javac -classpath
.;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
.\distrib\OptimDemo.jar PerformOptim.java
```

- On UNIX, execute this command:

```
javac -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
./distrib/OptimDemo.jar BananaFunction.java
javac -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
./distrib/OptimDemo.jar PerformOptim.java
```

16 Execute the `PerformOptim` class file as follows:

On Windows, type:

```
java -classpath
.;matlabroot\toolbox\javabuilder\jar\javabuilder.jar
.\distrib\OptimDemo.jar
PerformOptim -1.2 1.0
```

On UNIX, type:

```
java -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
./distrib/OptimDemo.jar
PerformOptim -1.2 1.0
```

Note Valid architectures on UNIX are `glnx86`, `glnxa64`, `mac`, `maci`, and `sol64`.

Note The supported JRE version is 1.6.0. To find out what JRE you are using, refer to the output of `version -java` in MATLAB or refer to the `jre.cfg` file in *matlabroot*/sys/java/jre/arch or *mcrroot*/sys/java/jre/arch.

Note If you are running on Solaris 64-bit platforms, you must add the `-d64` flag in the Java command. See "Limitations and Restrictions" on page 6-2 for more specific information.

When run successfully, the `PerformOptim` program should display the following output:

```
Using x0 =
-1.2000    1.0000
***************************************************
** Properties of Java object                     **
***************************************************


h =

BananaFunction@1766806
```

```
className =

BananaFunction

  Name       Size            Bytes  Class              Attributes

  h          1x1                    BananaFunction



Methods for class BananaFunction:


BananaFunction    getClass        notifyAll
equals            hashCode        toString
evaluateFunction  notify          wait

** Finished DISPLAYOBJ *****************************
****************************************************
** Performing unconstrained nonlinear optimization **
****************************************************

directEval =

   24.2000


wrapperEval =

   24.2000


x =

    1.0000    1.0000


fval =

   8.1777e-10
```

```
Optimization successful
** Finished DOOPTIM ********************************
Location of minimum:
1.0000    1.0000
Function value at minimum:
8.1777e-10
```

Deploying a Java Component Over the Web

- "Creating a Deployable Web Application" on page 5-2
- "Delivering Interactive Graphics Over the Web with WebFigures" on page 5-9
- "Creating Scalable Web Applications with RMI" on page 5-26

Creating a Deployable Web Application

In this section...
"Example Overview" on page 5-2
"Before You Begin" on page 5-2
"Downloading the Demo Files" on page 5-3
"Build Your Java Component" on page 5-4
"Compiling Your Java Code" on page 5-5
"Generating the Web Archive (WAR) File " on page 5-5
"Running the Web Deployment Demo" on page 5-6
"Using the Web Application" on page 5-6

Example Overview

This example demonstrates how to display a plot created by a Java servlet calling a component created with the MATLAB Builder JA product over a Web interface. This example uses MATLAB `varargin` and `varargout` for optional input and output to the `varargexample.m` function. For more information about `varargin` and `varargout`, see "How Does the MATLAB® Builder JA Product Handle Data?" on page 2-5.

Before You Begin

This section describes what you need to know and do before you create the Web deployment example.

- "Ensure You Have the Required Products" on page 5-2

- "Ensure Your Web Server Is Java Compliant" on page 5-3

- "Install the javabuilder.jar Library" on page 5-3

Ensure You Have the Required Products

The following products must be installed at their recommended release levels.

MATLAB, MATLAB Compiler, MATLAB Builder JA. This example was tested with R2007b.

Java Development Kit (JDK). Ensure you have Sun JDK v1.6.0 installed on your system. You can download it from Sun Microsystems, Inc.

Ensure Your Web Server Is Java Compliant

In order to run this example, your Web server must be capable of running accepted Java frameworks like J2EE. Running the WebFigures demo ("Delivering Interactive Graphics Over the Web with WebFigures" on page 5-9) also requires the ability to run servlets in WARs (Web Archives).

Install the javabuilder.jar Library

Ensure that the javabuilder.jar library (*matlabroot*/toolbox/javabuilder/jar/javabuilder.jar) has been installed into your Web server's common library directory.

Downloading the Demo Files

Download the demo files from the File Exchange at MATLAB Central. With **File Exchange** selected in the Search drop-down box, enter the keyword java_web_vararg_demo and click **Go**.

Contents of the Demo Files

The demo files contain the following three directories:

- mcode — Contains all of the MATLAB source code.

- JavaCode — Contains the required Java files and libraries.

- compile — Contains some helpful MATLAB functions to compile and clean up the demo.

> **Note** As an alternative to compiling the demo code manually and creating the application WAR (Web Archive) manually, you can run `compileVarArgServletDemo.m` in the compile directory. If you choose this option and want to change the locations of the output files, edit the values in `getVarArgServletDemoSettings.m`.
>
> If you choose to run `compileVarArgServletDemo.m`, consult the `readme` file in the download for additional information and then skip to "Running the Web Deployment Demo" on page 5-6

Build Your Java Component

Build your Java component by compiling your code into a deployable Java component `.jar` file.

> **Note** For a more detailed explanation of building a Java component, including further details on setting up your Java environment, the **src** and **distrib** directories, and other information, see Chapter 1, "Getting Started"

1 Start `deploytool` from the MATLAB command line.

2 Select **New Project > MATLAB Builder JA Project**.

3 Specify the project name as `vararg_java` and click **OK**.

4 In the Deployment Tool, right-click **vararg_javaclass** and select **Add File**.

5 Using the MATLAB Current Directory browser, navigate to the demo directory `mcode` and add the `varargexample.m` M-file to the class `vararg_javaclass` by dragging it to the **vararg_javaclass** folder in the Deployment Tool GUI.

6 Click the 🏭 icon on the Deployment Tool toolbar to build your project, creating `vararg_java.jar` in the `vararg_java` `distrib` output directory.

Compiling Your Java Code

Use `javac` to compile the Java source file `VarArgServletClass.java` from demo directory `JavaCode\src\VarArg`.

`javac.exe` should be located in the bin directory of your JDK installation.

Ensure your `classpath` is set to include:

- `javabuilder.jar` (shipped with the MATLAB Builder JA product)
- `servlet-api.jar` (in the demo directory `JavaCode\lib`)

For more details about using `javac`, see the Sun Microsystems, Inc. Web site.

Generating the Web Archive (WAR) File

Web archive or WAR files are a type of Java Archive used to deploy J2EE and JSP servlets. To run this example you will need to use the `jar` command to generate the final WAR file that runs the application. To do this, follow these steps:

1 Copy the JAR file created using the MATLAB Builder JA product into the `JavaCode\build\WEB-INF\lib` demo directory.

2 Copy the compiled Java class to the `JavaCode\build\WEB-INF\classes` demo directory.

3 Use the `jar` command to generate the final WAR as follows:

```
jar cf VarArgServlet.war -C build .
```

Caution   Don't omit the `.` parameter above, which denotes the current working directory.

For more information about the `jar` command, refer to the Sun Microsystems, Inc. Web site.

Running the Web Deployment Demo

When you're ready to run the application, do the following:

1 Install the `VarArgServlet.war` file into your Web server's `webapps` directory.

2 Run the application by entering `http://localhost:`*port_number*`/VarArgServlet` in the address field of your Web browser, where *port_number* is the port that your Web server is configured to use (usually `8080`).

Using the Web Application

To use the application, do the following on the `http://localhost/VarArgServlet` Web page:

1 Enter any amount of numbers to plot in the **Data to Plot** field.

2 Select **Line Color** and **Border Color** using the **Optional Input** drop-down lists. Note that these optional inputs are passed as `varargin` to the compiled M-code.

3 Select additional information you want to output, such as mean and standard deviation, by clicking an option in the **Optional Output** area. Note that these optional outputs are set as `varargout` from the compiled M-code.

4 Click **Display Plot**. Example output is shown below using the **Mean** optional output.
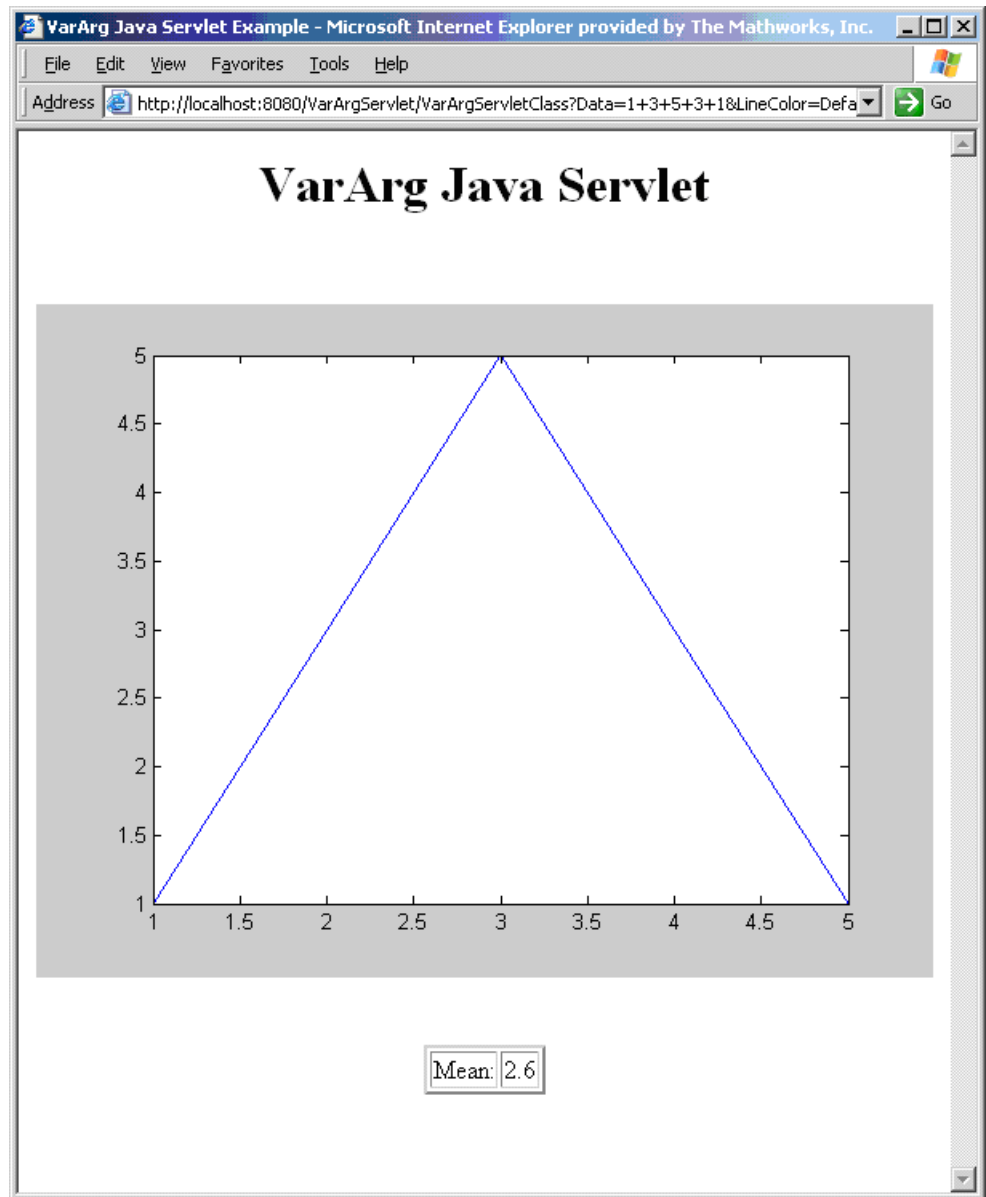
Delivering Interactive Graphics Over the Web with WebFigures

In this section...
"Before You Begin" on page 5-9
"Downloading the Example Files" on page 5-9
"WebFigures Feature" on page 5-9
"Preparing to Implement WebFigures" on page 5-10
"Implementing WebFigures" on page 5-16
"End-User Interaction with WebFigures" on page 5-24

Before You Begin

See the VarArg Web demo section "Before You Begin" on page 5-2 for information on properly setting up your Java environment before you run the example in this section.

Downloading the Example Files

You will need several files to run the example in this section. Download them from the File Exchange at MATLAB Central. With **File Exchange** selected in the Search drop-down box, enter the keyword java_web_figures_demo and click **Go**.

WebFigures Feature

You can provide interactive Web graphics using the WebFigures feature of the MATLAB Builder JA product. WebFigures provides thin client delivery of interactive Web graphics through a single Web application server running a single JVM on a single physical machine.

To implement WebFigures, use the Java WebFigure class package. This class encapsulates an individual figure and is a serializable, data-semantics class that can be attached to the following context scopes within J2EE:

- Application

- Page

- Session

Preparing to Implement WebFigures

To prepare to implement WebFigures, perform the following steps:

1 "Modifying the WebFigure Creation Code" on page 5-10

2 "Modifying the Code to Attach the WebFigure to the J2EE Context Scope" on page 5-11

3 "Embedding the WebFigure as Part of the Response HTML Page" on page 5-12

4 "Enabling User Interaction by Creating Mapping in a Web Deployment Descriptor File" on page 5-15

5 "Enabling Disposal of the WebFigure" on page 5-15

Modifying the WebFigure Creation Code

Add the code that will allow the WebFigure's creation by doing the following:

1 Add the `webfigure` statement to your existing M code. The syntax of the statement is:

```
webfigure_handle = webfigure(figure_handle);
```

When `webfigure` is executed, a "snapshot," or picture, is taken of an open MATLAB figure. The example code below opens a new figure window, plots some data in the window, creates the `WebFigure` object, and closes the figure. Note the `WebFigure` is valid even after the MATLAB figure has been closed.

```
function w = getplot
f = figure;
plot(1:10);
w = webfigure(f);
close(f);
```

2 Add the reference to the `WebFigure` object in your `MWArray` interface Java code so it can be received by the controller servlet or scriptlet as follows:

```
// generate the plot
Object[] results = matlabModel.getplot(1);

//unpack the WebFigure
MWJavaObjectRef ref = (MWJavaObjectRef)results[0];
WebFigure f = (WebFigure)ref.get();
```

When this code is run, the `WebFigure` is created and is ready for further manipulation by the Java application's controller code.

Modifying the Code to Attach the WebFigure to the J2EE Context Scope

Change the Java application's controller code to allow the figure to be accessible to the view layer. You do this by modifying the controller code so the figure is attached to a J2EE context scope, such as the session. Attaching to a session assumes that the figure will be interacted with by one user only. Attach the figure to a broader context scope, if needed, depending on the scope of your user audience. Alternate scopes are page or application.

Note For more information on Web-specific terms and concepts such as context scopes, see the Sun Microsystems, Inc. Web site.

Attach the figure to a context scope using the following syntax:

```
context_scope_object.setAttribute(name,
webfigure_object_variable);
```

For example, to attach to the session context scope from a servlet's service method, the following statement should be added to your controller code:

```
request.getSession().setAttribute("UserPlot", f);
```

When this statement is run, the `WebFigure` will have a name (`UserPlot`) which will be referred to by the view layer in the next step.

5-11

Embedding the WebFigure as Part of the Response HTML Page

To enable the user to interact with the figure, it must be embedded in an HTML page. To do this, allow the controller code to delegate displaying a response to the view layer.

Two methods are available for embedding the WebFigure, depending on whether the view layer is implemented as either a servlet or a JSP page.

See the following table for a listing of the required and optional parameters used in the constructor and the `getHTMLEmbedString` method.

Parameter	Definition
WebFigures constructor parameters	
root	URL where the WebFigures servlet is mapped in the deployment descriptor (web.xml)
servletContext	Reference to the servlet context for the current servlet
getHTMLEmbedString method parameters	
webfigure	Handle to the WebFigure being displayed
name	The attribute name used to store the WebFigure object
scope	The scope where the attribute is set; possible values can be session, page, or application
width	The width of the figure window in the HTML page. It can be specified in pixels or as a percentage of the width of the containing HTML element **Note** If a value is not provided (for JSP) or if it is null (for servlet), the MATLAB figure window dimensions will be used when rendering the figure on the Web page.

Parameter	Definition
height	The height of the figure window in the HTML page. It can be specified in pixels or as a percentage of the height of the containing HTML element **Note** If a value is not provided (for JSP) or if it is null (for servlet), the MATLAB figure window dimensions will be used when rendering the figure on the Web page.
style	Element-level CSS properties (from the style tag) used to customize the look of the figure interface on the HTML page

Embedding from a Servlet. To embed from a servlet, create an instance of the class WebFigures using the servlet API. Supply an argument to the WebFigures constructor that equates to the URL where the WebFigures servlet is mapped.

```
// The argument to the WebFigures constructor is the URL where the
// WebFigures servlet is mapped (relative to the Web application
                                 and servlet context)
WebFigures webFigures = new WebFigures("WebFigures", f,
                                       getServletContext());
responseWriter.print(webFigures.getHtmlEmbedString(wb, name,
                                                    scope, width,
                                                    height, options));
```

The following graphic depicts an AJAX client, running WebFigures, communicating with a view layer implemented as a servlet.

Client Browser   **Web Server**
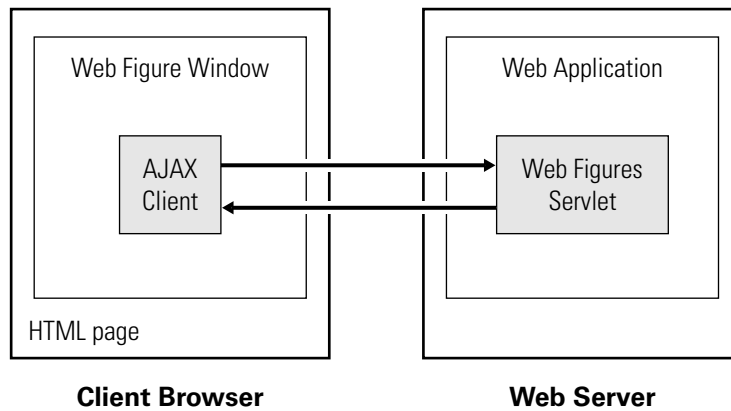
Embedding from a JSP Page. To embed from the JSP page, you must import the webfigures tag library and use the web-figure tag. Do this by adding the tags displayed in the bold lines in the following example:

```
<%@ taglib prefix="wf" uri="/WEB-INF/webfigures.tld" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>


<html>
    <head><title>MATLAB WebFigures Demo</title></head>
    <body>
        <table border=0 cellspacing=2 cellpadding=0 style="width:100%;height:100%">
        <tr><td>Use the controls below to interact with the surface plot.</td></tr>
        <tr><td height=100%>
        <wf:web-figure name="UserPlot" scope="session" root="WebFigures"
                        width="100%" height="100%"/>
        <td></tr>
        </table>
    </body>
</html>
```

By default, the following client settings have these values when the WebFigure is displayed for the first time:

- zoom-to-fit is true — The figure is automatically resized to fit the entire view port when the embedded HTML element is resized.

- `camera angle` is the same as in the original MATLAB figure when the WebFigure was created.

Enabling User Interaction by Creating Mapping in a Web Deployment Descriptor File

Users send requests from the client to the WebFigures servlet, which must be mapped to a URL in the Web application using WebFigures. Do this by creating or customizing the Web deployment descriptor file `/WEB-INF/web.xml`.

Use the following example template of the descriptor file, or use a similar template provided with your IDE:

```
<web-app>
    <servlet>
        <servlet-name>WebFigures</servlet-name>
        <servlet-class>
            com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WebFigures</servlet-name>
        <url-pattern>/WebFigures/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The `url-pattern` tag specifies that all URLs in `/Webfigures` are to be mapped to the `WebFiguresServlet` (accomplished by the `*` wildcard character).

Note The `WebFiguresServlet` does not have to be mapped onto `/WebFigures`. Choose the `url-pattern` most appropriate for your application.

Enabling Disposal of the WebFigure

To free up resources, you must provide a method to dispose of the created WebFigure. Do this by binding the `WebFigure` to the lifetime of an HTTP session context so that it will automatically be disposed of when the session

expires (via timeout, server shutdown, or explicit invalidation in Controller logic).

This is performed through use of a new utility class, `com.mathworks.toolbox.javabuilder.web.MWHttpSessionBinder`.

For example:

```
// bind the figure's lifetime to the session
sessionContext.setAttribute("UserPlotBinder", new MWHttpSessionBinder(figure));
```

This method ensures that `dispose` will be called on the given object when the attribute `UserPlotBinder` is unbound from the session. When the session expires, all its attributes are unbound. To disable this behavior after having bound the figure to the session, the following code may be used:

```
MWHttpSessionBinder binder =
       (MWHttpSessionBinder)sessionContext.getAttribute("UserPlotBinder");
binder.setObject(null);
sessionContext.removeAttribute("UserPlotBinder");
```

Setting the binder's object property to `null` ensures that `dispose` will not be called when `removeAttribute` is called.

Implementing WebFigures

You are now ready to compile and run your WebFigures application. Ensure you have the following at hand:

- From the files prepared in "Preparing to Implement WebFigures" on page 5-10:

  - M-code (`getplot.m`)

  - A controller servlet (`ModelRunnerServlet.java`) **or** JSP page (`response.jsp`)

  - An HTML page that serves as the entry point to the application (`index.html`)

  - The Web descriptor file, `web.xml`

- Provided as part of MATLAB Builder JA WebFigures:

- WebFigures servlet
  (`com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet`)
  included in `javabuilder.jar`

- The WebFigures JSP tag library descriptor (`webfigures.tld`)

To compile and run your WebFigures application.

1 "Compilng the M-Code into a Java Object" on page 5-17

2 "Writing Code to Instantiate the Java Object and Create the WebFigure" on page 5-18

3 "Adapting the HTML to Display the WebFigure" on page 5-22

4 "Adding Servlet Mapping for WebFiguresServlet" on page 5-23

5 "Adding the index.html to Create the Entry Point for the Application" on page 5-24

Compilng the M-Code into a Java Object

Create the Java object from the plot function example outlined in "Modifying the WebFigure Creation Code" on page 5-10 by doing the following:

1 Start `deploytool` from the MATLAB command prompt.

2 Select **File > New Deployment Project**.

3 In the next screen, select **MATLAB Builder JA** in the left pane and **Java Package** in the right pane.

4 Enter any name for the project in the **Name** field, verify or change the **Location**, and click **OK**.

5 Click **Settings**. In the **Package Name** field, specify the package name as com_mathworks_examples_plot.

6 In Deployment Tool, right-click the *project_name***class** folder name, select **Rename Class** and enter the name `plotter`.

5-17

7 Add the file `getplot.m` to the **plotter** class by dragging the file from the MATLAB Current Directory browser to the class folder in the Deployment Tool.

8 Click the ▥ icon on the Deployment Tool toolbar to build your project, creating `plot.jar` in the **distrib** output directory.

9 Copy the `plot.jar` files to your Web application directory tree `WEB-INF/lib/`.

Writing Code to Instantiate the Java Object and Create the WebFigure

Add code to the `ModelRunnerServlet.java` controller servlet to create the Java object and the WebFigure in realtime.

1 Add the following code to the `super.init()` method to instantiate the object:

```
try {
    // create a new plotter object
    matlabModel = new plotter();
}
catch (MWException mcrInitError) {
    mcrInitError.printStackTrace();
}
```

2 Add the following code to the `doGet()` method to create the WebFigure object:

```
// find the plotter object associated with this session
WebFigure userPlot = (WebFigure)session.getAttribute("UserPlot");

// if this is the first time doGet has been called for this session,
//  create the plot and WebFigure object
if (null == userPlot) {
    try {
        // generate the plot
        Object[] results = matlabModel.getplot(/* nargout = */ 1);

        try {
```

```
                              // unpack the WebFigure
                              MWJavaObjectRef ref = (MWJavaObjectRef)results[O];
                              userPlot = (WebFigure)ref.get();

                              // store the figure in the session context
                              session.setAttribute("UserPlot", userPlot);

                              // bind the figure's lifetime to the session
                              session.setAttribute("UserPlotBinder",
                                                   new MWHttpSessionBinder(userPlot));
                        }
                        finally {
                              // free MCR-related resources held by the results
                              MWArray.disposeArray(results);
                        }
                  }
                  catch (MWException getplotError) {
                        getplotError.printStackTrace();
                  }
            }
```

The complete `ModelRunnerServlet.java` program should look like this:

```java
// imported classes from JRE
import java.io.IOException;

// imported classes from servlet-api.jar
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.RequestDispatcher;

// imported classes from javabuilder.jar
import com.mathworks.toolbox.javabuilder.webfigures.WebFigure;
import com.mathworks.toolbox.javabuilder.MWJavaObjectRef;
import com.mathworks.toolbox.javabuilder.MWException;
```

```
import com.mathworks.toolbox.javabuilder.MWArray;
import com.mathworks.toolbox.javabuilder.web.MWHttpSessionBinder;

// imported classes from plot.jar
import com.mathworks.examples.plot.Plotter;

public class ModelRunnerServlet extends HttpServlet
{
    private plotter matlabModel = null;

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);


        try {
            // create a new plotter object
            matlabModel = new plotter();
        }
        catch (MWException mcrInitError) {
            mcrInitError.printStackTrace();
        }
    }

    public void destroy()
    {
        super.destroy();
        // free MCR-related resources
        matlabModel.dispose();
    }

    protected void doGet(final HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        HttpSession session = request.getSession();
        ServletContext servletContext = session.getServletContext();

        // find the plotter object associated with this session
        WebFigure userPlot = (WebFigure)session.getAttribute("UserPlot");
```

```
            // if this is the first time doGet has been called for this session,
            //  create the plot and WebFigure object
            if (null == userPlot) {
                try {
                    // generate the plot
                    Object[] results = matlabModel.getplot(/* nargout = */ 1);

                    try {
                        // unpack the WebFigure
                        MWJavaObjectRef ref = (MWJavaObjectRef)results[0];
                        userPlot = (WebFigure)ref.get();

                        // store the figure in the session context
                        session.setAttribute("UserPlot", userPlot);

                        // bind the figure's lifetime to the session
                        session.setAttribute("UserPlotBinder",
                                                new MWHttpSessionBinder(userPlot));
                    }
                    finally {
                        // free MCR-related resources held by the results
                        MWArray.disposeArray(results);
                    }
                }
                catch (MWException getplotError) {
                    getplotError.printStackTrace();
                }
            }

            // forward the request to the View layer (response.jsp)
            RequestDispatcher dispatcher = request.getRequestDispatcher("/response.jsp");
            dispatcher.forward(request, response);
        }
    }
```

Adapting the HTML to Display the WebFigure

Customize the response.jsp code in webfigures.tld to display the WebFigure to the user by doing the following:

1 Copy webfigures.tld from *matlabroot*/toolbox/javabuilder/webfigures/webfigures.tld to the directory WEB-INF/ under the Web application's directory tree.

2 Add the following declaration to reference webfigures.tld in the file response.jsp:

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

3 Add the following web-figure tag to display the figure:

```
<wf:web-figure name="UserPlot" scope="session" root="WebFigures"
               width="100%" height="100%"/>
```

When finished, the response.jsp page code should look like this:

```
<%@ taglib prefix="wf" uri="/WEB-INF/webfigures.tld" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<html>
  <head><title>MATLAB WebFigures Demo</title></head>
  <body>
    <table border=0 cellspacing=2 cellpadding=0 style="width:100%;height:100%">
      <tr><td>
          Use the console below to interact with the surface plot.
      </td></tr>
      <tr><td height=100%>
          <wf:web-figure name="UserPlot" scope="session" root="WebFigures"
                         width="100%" height="100%"/>
      </td></tr>
    </table>
  </body>
</html>
```

Adding Servlet Mapping for WebFiguresServlet

Add the following mapping to WEB-INF/web.xml so that the servlet can locate the WebFigure:

```
<servlet>
    <servlet-name>WebFigures</servlet-name>
    <servlet-class>
        com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>WebFigures</servlet-name>
    <url-pattern>/WebFigures/*</url-pattern>
</servlet-mapping>
```

Afterwards, the complete WEB-INF/web.xml file should look similar to this:

```
<?xml version="version_number" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>ModelRunnerServlet</servlet-name>
        <servlet-class>ModelRunnerServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ModelRunnerServlet</servlet-name>
        <url-pattern>/run_model</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>WebFigures</servlet-name>
        <servlet-class>
        com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WebFigures</servlet-name>
        <url-pattern>/WebFigures/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Adding the index.html to Create the Entry Point for the Application

Add the following index.html code:

```
<html>
<head><title>Welcome to the MATLAB WebFigures demo</title></head>
<body>Click <a href="run_model">here</a> to begin.</body>
</html>
```

End-User Interaction with WebFigures

An End-user can interact with a deployed WebFigure. They can use the dynamic menu interface to:

- Zoom (increase or decrease the size of the figure)

- Pan (change placement of figure on screen)

- Rotate (manipulate a figure to see all perspectives)

To use the dynamic menu interface, hover the mouse over the top of the WebFigure until you see three icons: a magnifying glass (zoom icon) a hand (pan icon) and a circle with a wraparound arrow (rotation icon). Click the icon that corresponds to the action you want to accomplish and use the mouse to manipulate the figure.

Note Currently, the WebFigures axis can only accommodate one subplot.

The following limitations currently exist from the end-user interface:

- Guide GUIs cannot be deployed as WebFigures

- Multiple sub-plotting in figures is available, but it is not possible to interact with all sub-plots.

Creating Scalable Web Applications with RMI

In this section...
"Using Remote Method Invocation (RMI)" on page 5-26
"Before You Begin" on page 5-27
"Running the Client and Server on a Single Machine" on page 5-27
"Running the Client and Server on Separate Machines" on page 5-31

Using Remote Method Invocation (RMI)

You can expand your application's throughput capacity by taking advantage of the MATLAB Builder JA product's use of RMI, the Java native remote procedure call (RPC) mechanism. The builder's implementation of RMI technology provides for automatic generation of interface code to enable components to be started in separate processes, on one or more computers, making your applications scalable and adaptable to future performance demands.

The following example uses RMI in the following ways:

- Running a client and server on a single machine
- Running a client on one machine and a server on another

Tip While running on UNIX, ensure you use : as the path separator in calls to java and javac. ; is used as a path separator only on Windows.
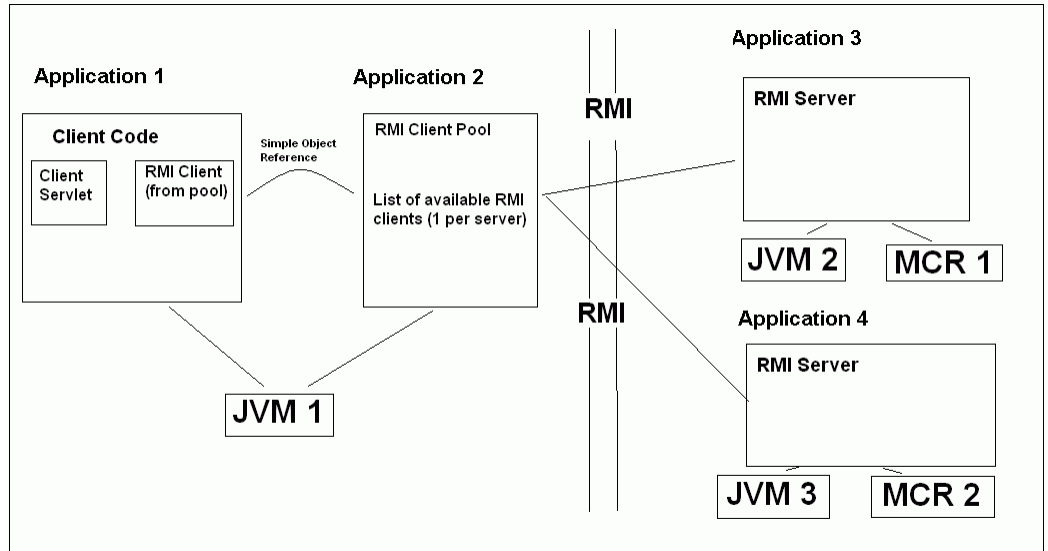
Before You Begin

See "Before You Begin" on page 5-2 for information on properly setting up your Java environment before you run the example in this section.

Running the Client and Server on a Single Machine

The following example shows how two separate processes can be run to initialize MATLAB struct arrays.

Note It is not necessary to have the MCR installed on the client side. Return values from the MCR can be automatically converted using the `marshalOutputs` Boolean in the `RemoteProxy` class. See the link in the "Documentation Set" from the product roadmap.

1 Compile the MATLAB Builder JA component by issuing the following at the MATLAB command prompt:

```
mcc -W 'java:dataTypesComp,dataTypesClass' createEmptyStruct.m
```

```
updateField.m -v
```

2 Compile the server Java code by issuing the following `javac` command. Ensure there are no spaces between `javabuilder.jar;` and *directory_containing_component.*

```
javac -classpath
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
directory_containing_component\dataTypesComp.jar
DataTypesServer.java
```

`DataTypesServer.java` can be found in *matlabroot*`\toolbox\javabuilder\Examples\RMIExamples\DataTypes\DataTypesDe`

3 Compile the client Java code by issuing the following `javac` command. Ensure there are no spaces between `javabuilder.jar;` and *directory containing component.*

```
javac -classpath
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
directory containing component\dataTypesComp.jar
DataTypesClient.java
```

4 Run the client and server as follows:

a Open two command windows on DOS or UNIX, depending on what platform you are using.

b If running Windows, ensure that *matlabroot*/bin/*arch* is defined on the system path. If running UNIX, ensure `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` are set properly.

c Run the server by issuing the following `java` command. Ensure there are no spaces between `dataTypesComp.jar;` and *matlabroot.*

```
java -classpath
.;directory_containing_component\dataTypesComp.jar;
matlabroot\toolbox\javabuilder\jar\javabuilder.jar

-Djava.rmi.server.codebase="file:///matlabroot/toolbox/javabuilder/jar/javabuilder.jar
 file:///directory_containing_component/dataTypesComp.jar" DataTypesServer
```

d Run the client by issuing the following *java* command. Ensure there are no spaces between `dataTypesComp.jar;` and *matlabroot*.

```
java -classpath
 .;directory_containing_component\dataTypesComp.jar;
matlabroot\toolbox\javabuilder\jar\javabuilder.jar
 DataTypesClient
```

`DataTypesClient.java` can be found in
matlabroot`\toolbox\javabuilder\Examples\RMIExamples\DataTypes\DataTypesDe`

5 If successful, the following output appears in the Command Window running the server:

```
Please wait for the server registration notification.
            Server registered and running successfully!!

            EVENT 1: Initializing the structure on server
                    and sending it to client:
                    Initialized empty structure:

                 Name: []
              Address: []

            ##################################

            EVENT 3: Partially initialized structure as received by server:

                 Name: []
              Address: [1x1 struct]

            Address field as initialized from the client:

               Street: '3, Apple Hill Drive'
                 City: 'Natick'
                State: 'MA'
                  Zip: '01760'

            ##################################
```

```
                    EVENT 4: Updating 'Name' field before
                            sending the structure back to the client:

                        Name: 'The MathWorks'
                     Address: [1x1 struct]


                ################################
```

If successful, the following output appears in the Command Window
running the client:

```
    Running the client application!!

                    EVENT 2: Initialized structure as received in client applications:

                         Name: []
                      Address: []

                    Updating the 'Address' field to :

                        Street: '3, Apple Hill Drive'
                          City: 'Natick'
                         State: 'MA'
                           Zip: '01760'

                ################################


                    EVENT 5: Final structure as received by client:

                         Name: 'The MathWorks'
                      Address: [1x1 struct]

                    Address field:

                        Street: '3, Apple Hill Drive'
                          City: 'Natick'
                         State: 'MA'
                           Zip: '01760'
```

#################################

Running the Client and Server on Separate Machines

To implement RMI with a client on one machine and a server on another, you must:

- Change how the server is bound to the system registry.

- Redefine how the client accesses the server.

After this is done, follow the steps in "Running the Client and Server on a Single Machine" on page 5-27.

6

Reference Information for Java

Requirements for the MATLAB Builder JA Product

In this section...
"System Requirements" on page 6-2
"Limitations and Restrictions" on page 6-2
"Settings for Environment Variables (Development Machine)" on page 6-3

System Requirements

System requirements and restrictions on use for the MATLAB Builder JA product are as follows:

- All requirements for the MATLAB Compiler product; see Installation and Configuration in the MATLAB Compiler documentation.

- Java Development Kit (JDK) 1.5 or later must be installed.

- Java Runtime Environment (JRE) that is used by MATLAB and MCR.

Note The supported JRE version is 1.6.0. To find out what JRE you are using, refer to the output of `'version -java'` in MATLAB or refer to the `jre.cfg` file in *matlabroot*/sys/java/jre/<arch> or *mcrroot*/sys/java/jre/<arch>.

Limitations and Restrictions

In general, limitations and restrictions on the use of the MATLAB Builder JA product are the same as those for the MATLAB Compiler product. See "Limitations and Restrictions" in the MATLAB Compiler documentation for details.

In addition, the MATLAB Builder JA product does not support MATLAB object data types (for example, Time Series objects).

On 64-bit Solaris computers, you need to include the `-d64` flag in the Java command used to run the application. This is because by default the Java Virtual Machine (JVM) starts in 32-bit client mode on 64-bit Solaris machines

and the MATLAB Compiler Runtime (MCR) requires a 64-bit JVM. For example when running "Deploying a Component with the Magic Square Example" on page 1-9, you should use the following command:

```
matlabroot/sys/java/jre/architecture/jre_directory/bin/java -classpath
  :matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
MagicDemoJavaApp/magicsquare/distrib/magicsquare.jar -d64 getmagic 5
```

Settings for Environment Variables (Development Machine)

Before starting to program, you must set the environment on your development machine to be compatible with the MATLAB Builder JA product.

Specify the following environment variables:

- "JAVA_HOME Variable" on page 6-3
- "Java CLASSPATH Variable" on page 6-4
- "Native Library Path Variables" on page 6-6

JAVA_HOME Variable

The MATLAB Builder JA product uses the JAVA_HOME variable to locate the Java Software Development Kit (SDK) on your system. It also uses this variable to set the versions of the javac.exe and jar.exe files it uses during the build process.

Note If you do not set JAVA_HOME, the builder assumes that \jdk\bin is on the system path.

Setting JAVA_HOME on Windows (Development Machine). If you are working on Windows, set your JAVA_HOME variable by entering the following command in your DOS command window. (In this example, your Java SDK is installed in directory C:\java\jdk\j2sdk1.6.0.)

```
set JAVA_HOME=C:\java\jdk\j2sdk1.6.0
```

Alternatively, you can add *jdk_directory*/bin to the system path. For example:

```
set PATH=%PATH%;c:\java\jdk\j2sdk1.6.0\bin
```

You can also set these variables globally using the Windows Control Panel. Consult your Windows documentation for instructions on setting system variables.

Note The supported JRE version is 1.6.0. To find out what JRE you are using, refer to the output of 'version -java' in MATLAB or refer to the jre.cfg file in *matlabroot*/sys/java/jre/<arch> or *mcrroot*/sys/java/jre/<arch>.

Setting JAVA_HOME on UNIX (Development Machine). If you are working on a UNIX system, set your JAVA_HOME variable by entering the following commands at the command prompt. (In this example, your Java SDK is installed in directory /java/jdk/j2sdk1.6.0.)

```
setenv JAVA_HOME /java/jdk/j2sdk1.6.0
```

Alternatively, you can add *jdk_directory*\bin to the system path.

Java CLASSPATH Variable

To build and run a Java application that uses a MATLAB Builder JA generated component, the system needs to find .jar files containing the MATLAB libraries and the class and method definitions that you have developed and built with the builder. To tell the system how to locate the .jar files it needs, specify a classpath either in the javac command or in your system environment variables.

Java uses the CLASSPATH variable to locate user classes needed to compile or run a given Java class. The class path contains directories where all the .class and/or .jar files needed by your program reside. These .jar files contain any classes that your Java class depends on.

When you compile a Java class that uses classes contained in the com.mathworks.toolbox.javabuilder package, you need to include a file

called `javabuilder.jar` on the Java class path. This file comes with the builder; you can find it in the following directory:

```
matlabroot/toolbox/javabuilder/jar % (development machine)
mcrroot/toolbox/javabuilder/jar % (end-user machine)
```

where *matlabroot* refers to the root directory into which the MATLAB installer has placed the MATLAB files, and `mcrroot` refers to the root directory under which MCR is installed. The builder automatically includes this `.jar` file on the class path when it creates the component. To use a class generated by the builder, you need to add this *matlabroot*/`toolbox/javabuilder/jar/javabuilder.jar` to the class path.

In addition, you need to add to the class path the `.jar` file created by the builder for your compiled `.class` files.

Example: Setting CLASSPATH on Windows. Suppose your MATLAB libraries are installed in `C:\`*matlabroot*`\bin\win32`, and your component `.jar` files are in `C:\mycomponent`.

Note For *matlabroot*, substitute the MATLAB root directory on your system. Type `matlabroot` to see this directory name.

To set your `CLASSPATH` variable on your development machine, enter the following command at the DOS command prompt:

```
set CLASSPATH=.;C:\matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
C:\mycomponent\mycomponent.jar
```

Alternatively, if the Java SDK is installed, you can specify the class path on the Java command line as follows. When entering this command, ensure there are no spaces between path names in the *matlabroot* argument. For example, there should be no space between `javabuilder.jar;` and `c:\mycomponent\mycomponent.jar` in the following example:

```
javac
   -classpath .;C:\matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
   C:\mycomponent\mycomponent.jar usemyclass.java
```

where usemyclass.java is the file to be compiled.

It is recommended that you globally add any frequently used class paths to the CLASSPATH system variable via the Windows Control Panel.

Example: Setting CLASSPATH on UNIX (Development Machine).
Suppose your UNIX environment is as follows:

- Your MATLAB libraries are installed in /*matlabroot*/bin/*arch*, (where *arch* is either glnx86, glnxa64, mac, or sol64, depending on the operating system of the machine.

- Your component .jar files are in /mycomponent.

To set your CLASSPATH variable, enter the following command at the prompt:

```
setenv CLASSPATH .:/matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
/mycomponent/mycomponent.jar
```

Like Windows, you can specify the class path directly on the Java command line. To compile usemyclass.java, type the following:

```
javac -classpath
.:/matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
/mycomponent/mycomponent.jar usemyclass.java
```

where usemyclass.java is the file to be compiled.

Native Library Path Variables
The operating system uses the native library path to locate native libraries that are needed to run your Java class. See the following list of variable names according to operating system:

Windows	PATH
Linux	LD_LIBRARY_PATH
Solaris	LD_LIBRARY_PATH
Macintosh	DYLD_LIBRARY_PATH

For information on how to set these path variables, see the following topics in "Standalone Applications" of the MATLAB Compiler documentation:

• See Testing the Application to set your path on a development machine

• See Running the Application to set your path on an end-user machine

Data Conversion Rules

In this section...
"Java to MATLAB Conversion" on page 6-8
"MATLAB to Java Conversion" on page 6-10
"Unsupported MATLAB Array Types" on page 6-11

Java to MATLAB Conversion

The following table lists the data conversion rules for converting Java data types to MATLAB types.

Note The conversion rules apply to scalars, vectors, matrices, and multidimensional arrays of the types listed.

The conversion rules apply not only when calling your own methods, but also when calling constructors and factory methods belonging to the MWArray classes.

When calling an MWArray class method constructor, supplying a specific data type causes the builder to convert to that type instead of the default.

Java to MATLAB Conversion Rules

Java Type	MATLAB Type
double	double
float	single
byte	int8
int	int32
short	int16
long	int64
char	char

Java to MATLAB Conversion Rules (Continued)

Java Type	MATLAB Type
boolean	logical
java.lang.Double	double
java.lang.Float	single
java.lang.Byte	int8
java.lang.Integer	int32
java.lang.Long	int64
java.lang.Short	int16
java.lang.Number	double **Note** Subclasses of java.lang.Number not listed above are converted to double.
java.lang.Boolean	logical
java.lang.Character	char
java.lang.String	char **Note** A Java string is converted to a 1-by-N array of char with N equal to the length of the input string. An array of Java strings (String[]) is converted to an M-by-N array of char, with M equal to the number of elements in the input array and N equal to the maximum length of any of the strings in the array. Higher dimensional arrays of String are converted similarly. In general, an N-dimensional array of String is converted to an N+1 dimensional array of char with appropriate zero padding where supplied strings have different lengths.

MATLAB to Java Conversion

The following table lists the data conversion rules for converting MATLAB data types to Java types.

> **Note** The conversion rules apply to scalars, vectors, matrices, and multidimensional arrays of the types listed.

MATLAB to Java Conversion Rules

MATLAB Type	Java Type (Primitive)	Java Type (Object)
cell	N/A	Object **Note** Cell arrays are constructed and accessed as arrays of MWArray.
structure	N/A	Object **Note** Structure arrays are constructed and accessed as arrays of MWArray.
char	char	java.lang.Character
double	double	java.lang.Double
single	float	java.lang.Float
int8	byte	java.lang.Byte
int16	short	java.lang.Short
int32	int	java.lang.Integer
int64	long	java.lang.Long

MATLAB to Java Conversion Rules (Continued)

MATLAB Type	Java Type (Primitive)	Java Type (Object)
uint8	byte	java.lang.ByteJava has no unsigned type to represent the uint8 used in MATLAB. Construction of and access to MATLAB arrays of an unsigned type requires conversion.
uint16	short	java.lang.shortJava has no unsigned type to represent the uint16 used in MATLAB. Construction of and access to MATLAB arrays of an unsigned type requires conversion.
uint32	int	java.lang.IntegerJava has no unsigned type to represent the uint32 used in MATLAB. Construction of and access to MATLAB arrays of an unsigned type requires conversion.
uint64	long	java.lang.LongJava has no unsigned type to represent the uint64 used in MATLAB. Construction of and access to MATLAB arrays of an unsigned type requires conversion.
logical	boolean	java.lang.Boolean
Function handle	Not supported	
Java class	Not supported	
User class	Not supported	

Unsupported MATLAB Array Types

Java has no unsigned types to represent the uint8, uint16, uint32, and uint64 types used in MATLAB. Construction of and access to MATLAB arrays of an unsigned type requires conversion.

Programming Interfaces Generated by the MATLAB Builder JA Product

In this section...
"APIs Based on MATLAB Function Signatures" on page 6-12
"Standard API" on page 6-13
"mlx API" on page 6-15
"Code Fragment: Signatures Generated for myprimes Example" on page 6-15

APIs Based on MATLAB Function Signatures

The builder generates two kinds of interfaces to handle MATLAB function signatures.

- A *standard* signature in Java

  This interface specifies input arguments for each overloaded method as one or more input arguments of class java.lang.Object or any subclass (including subclasses of MWArray). The standard interface specifies return values, if any, as a subclass of MWArray.

- mlx API

  This interface allows the user to specify the inputs to a function as an Object array, where each array element is one input argument. Similarly, the user also gives the mlx interface a preallocated Object array to hold the outputs of the function. The allocated length of the output array determines the number of desired function outputs.

  The mlx interface may also be accessed using java.util.List containers in place of Object arrays for the inputs and outputs. Note that if List containers are used, the output List passed in must contain a number of elements equal to the desired number of function outputs.

  For example, this would be incorrect usage:

  ```
  java.util.List outputs = new ArrayList(3);
  myclass.myfunction(outputs, inputs); // outputs contains O elements!
  ```

And this would be the correct usage:

```
java.util.List outputs = Arrays.asList(new Object[3]);
myclass.myfunction(outputs, inputs); // ok, list contains 3 elements
```

Typically you use the standard interface when you want to call MATLAB functions that return a single array. In other cases you probably need to use the mlx interface.

Standard API

The standard calling interface returns an array of one or more MWArray objects.

The standard API for a generic function with none, one, more than one, or a variable number of arguments, is shown in the following table.

Arguments	API to Use

Generic MATLAB function	<pre>function [Out1, Out2, ...,
	  varargout] =
	  foo(In1, In2, ...,
	  InN, varargin)</pre>

API if there are no input arguments	<pre>public Object[] foo(
	int numArgsOut
	)</pre>

API if there is one input argument	<pre>public Object[] foo(
	int numArgsOut,
	Object In1
	)</pre>

Arguments	API to Use
| API if there are two to *N* input arguments | ```
public Object[] foo(
int numArgsOut,
Object In1,
Object In2,
...
Object InN
)
``` |
| API if there are optional arguments, represented by the varargin argument | ```
public Object[] foo(
int numArgsOut,
 Object in1,
 Object in2,
 ...,
Object InN,
Object varargin
)
``` |

Details about the arguments for these samples of standard signatures are shown in the following table.

Argument	Description	Details About Argument
numArgsOut	Number of outputs	An integer indicating the number of outputs you want the method to return. To return no arguments, omit this argument.
		The value of *numArgsOut* must be less than or equal to the MATLAB function nargout.
		The *numArgsOut* argument must always be the first argument in the list.

Argument	Description	Details About Argument
In1, In2, ...InN	Required input arguments	All arguments that follow *numArgsOut* in the argument list are inputs to the method being called.
		Specify all required inputs first. Each required input must be of class MWArray or any class derived from MWArray.
varargin	Optional inputs	You can also specify optional inputs if your M-code uses the varargin input: list the optional inputs, or put them in an Object[] argument, placing the array last in the argument list.
Out1, Out2, ...OutN	Output arguments	With the standard calling interface, all output arguments are returned as an array of MWArrays.

mlx API

For a function with the following structure:

```
function [Out1, Out2, ..., varargout] =
  foo(In1, In2, ..., InN, varargin)
```

The builder generates the following API, as the mlx interface:

```
public void foo (List outputs, List inputs) throws MWException;
public void foo (Object[] outputs, Object[] inputs) throws MWException;
```

Code Fragment: Signatures Generated for myprimes Example

For a specific example, look at the myprimes method. This method has one input argument, so the builder generates three overloaded methods in Java.

When you add myprimes to the class myclass and build the component, the builder generates the myclass.java file. A fragment of myclass.java is listed to show overloaded implementations of the myprimes method in the Java code.

The standard interface specifies inputs to the function within the argument list and outputs as return values. The second implementation demonstrates the feval interface, the third implementation shows the interface to be used if there are no input arguments, and the fourth shows the implementation to be used if there is one input argument. Rather than returning function outputs as a return value, the feval interface includes both input and output arguments in the argument list. Output arguments are specified first, followed by input arguments.

```
/* mlx interface   List version */
public void myprimes(List lhs, List rhs) throws MWException
{
    (implementation omitted)
}
/* mlx interface   Array version */
public void myprimes(Object[] lhs, Object[] rhs) throws MWException
{
    (implementation omitted)
 }
/* Standard interface   no inputs*/
public Object[] myprimes(int nargout) throws MWException
   {
      (implementation omitted)
   }
/* Standard interface   one input*/
public Object[] myprimes(int nargout, Object n) throws MWException
   {
      (implementation omitted)
   }
```

See "APIs Based on MATLAB Function Signatures" on page 6-12 for details about the interfaces.

MWArray Class Specification

For complete reference information about the `MWArray` class hierarchy, see `com.mathworks.toolbox.javabuilder.MWArray`, which is in the *matlabroot*`/help/toolbox/javabuilder/MWArrayAPI/` directory.

Note For *matlabroot*, substitute the MATLAB root directory on your system. Type `matlabroot` to see this directory name.

Function Reference

deploytool

Purpose   Open GUI for MATLAB Builder JA and MATLAB Compiler products

Syntax   `deploytool`

Description   The `deploytool` command opens the Deployment Tool window, which is the graphical user interface (GUI) for the MATLAB Builder JA and MATLAB Compiler products.

Purpose	Invoke MATLAB Compiler

Syntax

```
mcc [- W | - S] 'java:component_name,class_name,
file1[file2...fileN]
[class{class_name:file1 [,file2,...,fileN]},...]
[-d output_dir_path]

mcc - B 'bundlefile'[:arg1, arg2, ..., argN]
```

Description Use the mcc command to invoke the MATLAB Compiler product either from the MATLAB command prompt (MATLAB mode) or the DOS or UNIX command line (standalone mode).

mcc prepares M-file(s) for deployment outside of the MATLAB environment.

Options

Note For a complete list of all mcc command options, see mcc in the MATLAB Compiler User's Guide documentation.

-W
> The -W option is used when running mcc with the MATLAB Builder JA product to create a class encapsulating one or more M-files.

-W String Elements	**Description**
java:	Keyword that tells the compiler the type of component to create, followed by a colon. Specify java: to create a Java component.
component_name	Specifies the name of the component and its namespace, which is a period-separated list, such as companyname.groupname.component.

-W String Elements	Description
class_name	Specifies the name of the Java class to be created. The default *class_name* is the last item in the list specified as *component_name*. *file1* [*file2...fileN*] are M-files to be encapsulated as methods in *class_name*.
[class{*class_name*:*file1* [,*file2*,...,*fileN*]},...]	Optional. Specifies additional classes that you want to include in the component. To use this option, you specify the class name, followed by a colon, and then the names of the files you want to include in the class. You can include this multiple times to specify multiple classes.
[-d *output_dir_path*]	Optional. Tells the builder to create a directory and copy the output files to it. If you use mcc instead of the Deployment Tool, the *project_directory*\src and *project_directory*\distrib directories are not automatically created.

> **Note** The -S option lets you control how each Java class uses the MCR.
>
> It tells the builder to create a single MCR when the first Java class is instantiated. This MCR is reused and shared among all subsequent class instances within the component, resulting in more efficient memory usage and eliminating the MCR startup cost in each subsequent class instantiation.
>
> By default, a new MCR instance is created for each instance of each Java class in the component. Use -S to change the default.
>
> When using -S, note that all class instances share a single MATLAB workspace and share global variables in the M-files used to build the component. This makes properties of a Java class behave as static properties instead of instance-wise properties.

The -B option is used to simplify the command-line input.

-B

    The -B option tells the builder to replace a specified file with the command-line information it contains.

-B String Elements	Description
java:	Keyword that tells the compiler the type of component to create, followed by a colon. Specify java: to create a Java component.

-B String Elements	Description
bundlefile	Specifies the name of the file containing predefined *component* and *class* information.
[:*arg1*, *arg2*, ..., *argN*]	Files or other arguments used by the *bundlefile*.

Examples   **Using -W with One Class**

```
mcc -W 'java:com.mycompany.mycomponent,myclass'
    foo.m bar.m
```

The example creates a Java component that has a fully qualified package name, com.mycompany.mycomponent. The component contains a single Java class, myclass, which contains methods foo and bar.

To use myclass, place the following statement in your code:

```
import com.mycompany.mycomponent.myclass;
```

Using -W with Additional Classes

```
mcc -W 'java:com.mycompany.mycomponent,myclass'
    foo.m bar.m class{myclass2:foo2.m,bar2.m}
```

The example creates a Java component named mycomponent with two classes:

myclass has methods foo and bar.

myclass2 has methods foo2 and bar2.

Using -B to Simplify Command Input

Suppose myoptions file contains

```
-W 'java:mycomponent,myclass'
```

In this case,

```
mcc -B 'myoptions' foo.m bar.m
```

produces the same results as

```
mcc -W 'java:[mycomponent,myclass]' foo.m bar.m
```

Using -S to Initialize a Single MCR

```
mcc -S 'java:mycomponent,myclass' foo.m bar.m
```

The example creates a Java component called `mycomponent` containing a single Java class named `myclass` with methods `foo` and `bar`. (See the first example in this table)

If and when multiple instances of `myclass` are instantiated in an application, only one MCR is initialized, and it is shared by all instances of `myclass`.

Note All of these command-line examples produce the `mycomponent.jar` file (component `jar` file).

Notice that the component name used to create these files is derived from the last item on the period-separated list that specifies the fully qualified name of the class.

---

mcc

The MATLAB Application Deployment Web Example Guide

For More Examples

For more deployment examples, see the *MATLAB® Application Deployment Web Example Guide.*

Examples

Use this list to find examples in the documentation.

Handling Data

Handling Errors

Handling Memory

COM Components

Sample Applications (Java)

Index