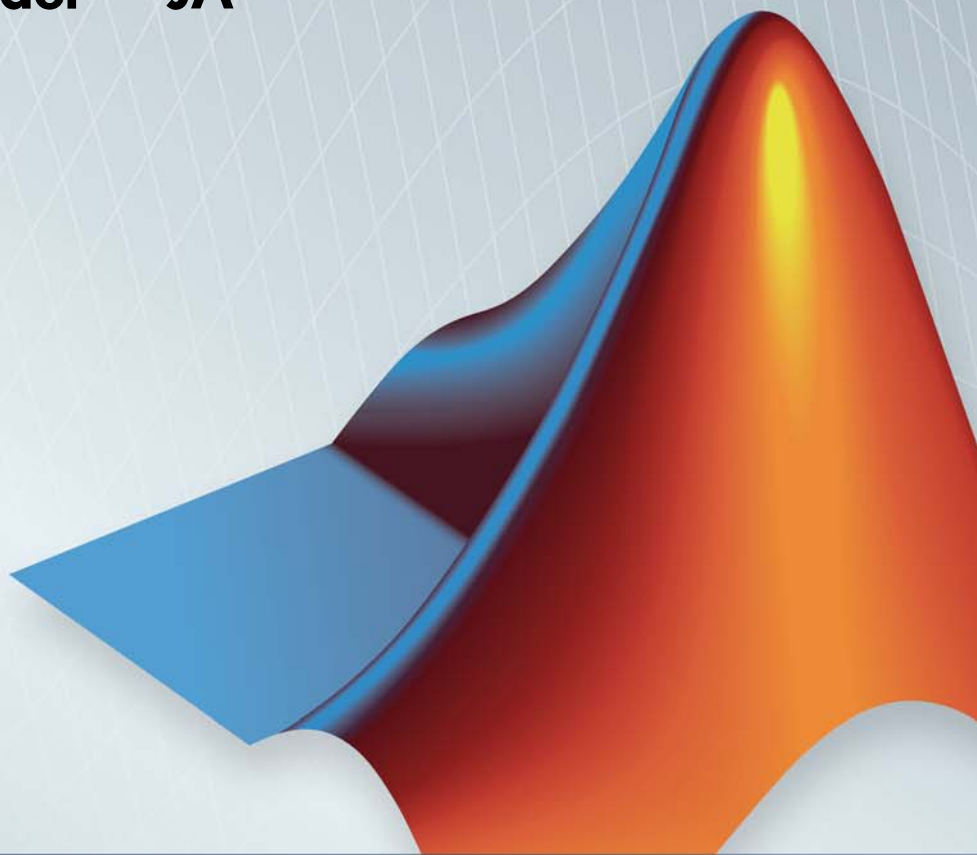


MATLAB® Builder™ JA

User's Guide

R2013b



MATLAB®



How to Contact 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 2006–2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2006	Online only	New for Version 1.0 (Release 2006b)
March 2007	Online only	Revised for Version 1.1 (Release 2007a)
September 2007	Online only	Revised for Version 2.0 (Release 2007b)
March 2008	Online only	Revised for Version 2.0.1 (Release 2008a)
October 2008	Online only	Revised for Version 2.0.2 (Release 2008b)
March 2009	Online only	Revised for Version 2.0.3 (Release 2009a)
September 2009	Online only	Revised for Version 2.0.4 (Release 2009b)
March 2010	Online only	Revised for Version 2.1 (Release 2010a)
September 2010	Online only	Revised for Version 2.2 (Release 2010b)
January 2011	Online only	Revised for Version 2.2.1 (Release 2010bSP1)
April 2011	Online only	Revised for Version 2.2.2 (Release 2011a)
September 2011	Online only	Revised for Version 2.2.3 (Release 2011b)
March 2012	Online only	Revised for Version 2.2.4 (Release 2012a)
September 2012	Online only	Revised for Version 2.2.5 (Release 2012b)
March 2013	Online only	Revised for Version 2.2.6 (Release 2013a)
September 2013	Online only	Revised for Version 2.3 (Release 2013b)

Getting Started

1

MATLAB Builder JA Product Description	1-2
Key Features	1-2
Determine Appropriate Tasks for MATLAB Compiler and Builder Products	1-3
Roles in the Java Application Deployment Process ...	1-5
Configure Your Environment	1-7
Install the Required JDK	1-7
Set JAVA_HOME	1-8
Set the CLASSPATH	1-8
Configure the Native Library Path Variables	1-9
Create a Java Package from MATLAB Code	1-10
Integrate a Java Package into an Application	1-16

Overview

2

Product Overview	2-2
MATLAB Compiler Extension	2-2
How MATLAB Builder JA Works	2-2
How the MATLAB Compiler and MATLAB Builder JA Products Work Together	2-4
How Does Java Package Deployment Work?	2-4
Limitations of Support	2-4

Application Deployment Products and the Compiler	
Apps	2-5
What Is the Difference Between the Compiler Apps and the mcc Command Line?	2-5
How Does MATLAB Compiler Software Build My Application?	2-5
Dependency Analysis Function	2-8
MEX-Files, DLLs, or Shared Libraries	2-9
Component Technology File (CTF Archive)	2-9
MATLAB Builder JA Prerequisites	2-13
What You Need to Know	2-13
Required Products	2-13
Dependency and Non-Compilable Code Considerations ...	2-13
Integrating a Generated Java Package into a Java	
Application	2-15
Gathering Files Needed for Deployment	2-16
Testing the Java Package in a Java Application	2-16
Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)	2-21
Integrating Java Classes Generated by MATLAB into a Java Application	2-23
Calling Class Methods from Java	2-24
Handle Data Conversion as Needed	2-24
Build and Test	2-25
Next Steps	2-26

MATLAB Code Guidelines

3

Write Deployable MATLAB Code	3-2
Compiled Applications Do Not Process MATLAB Files at Runtime	3-2
Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files	3-3
Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths	3-4

Gradually Refactor Applications That Depend on Noncompilable Functions	3-4
Do Not Create or Use Nonconstant Static State Variables	3-5
Get Proper Licenses for Toolbox Functionality You Want to Deploy	3-5
Load MATLAB Libraries using loadlibrary	3-7
Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler	3-8
Use MATLAB Data Files (MAT Files) in Compiled Applications	3-9
Explicitly Including MAT files Using the %#function Pragma	3-9
Load and Save Functions	3-9
MATLAB Objects	3-12

Deploying Java Packages

4

Compile a Java Package with the Library Compiler App	4-2
Compile a Java Package from the Command Line	4-8
Execute Compiler Projects with deploytool	4-8
Compile a Java Package with mcc	4-8
Map Functions to Java Class Methods	4-10
Use the Library Compiler App to Map Functions to Java Classes	4-10
Use mcc to Map Functions to Java Classes	4-12

Customizing a Compiler Project

5

Customizing the Installer	5-2
Changing the Application Icon	5-2
Adding Application Information	5-3
Changing the Splash Screen	5-4
Changing the Installation Path	5-4
Changing the Application Logo	5-5
Editing the Installation Notes	5-5
Manage the Required Files Compiled into a Project ..	5-6
Dependency Analysis	5-6
Using the Compiler Apps	5-6
Using mcc	5-7
Specify Additional Files to Be Installed with the Application	5-8

Programming

6

About the MATLAB Builder JA API	6-3
What Are MATLAB Generated Java Packages and When Should You Create Them?	6-3
Understanding the MATLAB Builder JA API Data Conversion Classes	6-4
Automatic Conversion to MATLAB Types	6-5
Understanding Function Signatures Generated by the MATLAB Builder JA Product	6-6
Adding Fields to Data Structures and Data Structure Arrays	6-7
Returning Data from MATLAB to Java	6-7
Importing Classes	6-8
Creating an Instance of the Class	6-9
What Is an Instance?	6-9

Instantiate a Java Class	6-9
Passing Arguments to and from Java	6-13
Format	6-13
Manual Conversion of Data Types	6-13
Automatic Conversion to a MATLAB Type	6-14
Specifying Optional Arguments	6-16
Handling Return Values	6-21
Passing Java Objects by Reference	6-27
MATLAB Array	6-27
Wrappering and Passing Java Objects to MATLAB Functions with MWJavaObjectRef	6-27
Handling Errors	6-33
Error Overview	6-33
Handling Checked Exceptions	6-33
Handling Unchecked Exceptions	6-36
Alternatives to Using of System.exit	6-39
Managing Native Resources	6-40
What Are Native Resources?	6-40
Using Garbage Collection Provided by the JVM	6-40
Using the dispose Method	6-41
Overriding the Object.Finalize Method	6-43
Improving Data Access Using the MCR User Data	
Interface and MATLAB Builder JA	6-44
Supply Run-Time Profile Information for Parallel Computing Toolbox Applications	6-45
Dynamically Specifying Run-Time Options to the	
MCR	6-50
What Run-Time Options Can You Specify?	6-50
Setting and Retrieving MCR Option Values Using MWApplication	6-50
Sharing an MCR Instance in COM or Java	
Applications	6-53
What Is a Singleton MCR?	6-53
Advantages and Disadvantages of Using a Singleton	6-53

Which Products Support Singleton MCR and How Do I Create a Singleton?	6-54
Handling Data Conversion Between Java and MATLAB	6-55
Overview	6-55
Calling MWArray Methods	6-55
Creating Buffered Images from a MATLAB Array	6-56
Setting Java Properties	6-57
How to Set Java System Properties	6-57
Ensure a Consistent GUI Appearance	6-57
Blocking Execution of a Console Application that Creates Figures	6-59
waitForFigures Method	6-59
Block Figure Window Display in a Console Application ...	6-60
Ensuring Multi-Platform Portability	6-62
CTF Archive Embedding and Extraction	6-64
Overview	6-64
Using MWComponentOptions Class to Indicate Extraction Options	6-64
Using Environment Variables to Indicate Extraction Options	6-66
For More Information	6-68
Learning About Java Classes and Methods by Exploring the Javadoc	6-69

Sample Java Applications

7

Plot	7-2
Purpose	7-2
Procedure	7-2

Spectral Analysis	7-9
Purpose	7-9
Procedure	7-10
Matrix Math	7-16
Purpose	7-16
MATLAB Functions to Be Encapsulated	7-17
Understanding the getfactor Program	7-18
Procedure	7-18
Phone Book	7-28
Purpose	7-28
Procedure	7-28
Optimization	7-36
Purpose	7-36
OptimDemo Package	7-36
Procedure	7-37
Web Application	7-47
Overview	7-47
Prerequisites	7-47
Downloading the Example Files	7-48
Build Your Java Package	7-49
Compiling Your Java Code	7-50
Generating the Web Archive (WAR) File	7-50
Running the Web Deployment Example	7-51
Using the Web Application	7-51

Deploying a Java Package Over the Web

8

About the WebFigures Feature	8-2
Supported Renderers for WebFigures	8-2
Preparing to Implement WebFigures for MATLAB	
Builder JA	8-3
Your Role in the WebFigure Deployment Process	8-3

What You Need to Know to Implement WebFigures	8-5
Required Products	8-5
Assumptions About the Examples	8-7
Set DISPLAY on UNIX Systems	8-8
Implement a Custom WebFigure	8-9
Overview	8-9
Setting Up the Web Server	8-9
Create the Default WebFigure	8-13
Interact with the Default WebFigure	8-14
Create a Custom WebFigure	8-15
Advanced Configuration of a WebFigure	8-19
Overview	8-19
How Do WebFigures Work?	8-21
Installing WebFigureService	8-22
Getting the WebFigure Object from Your Method	8-23
Attach a WebFigure	8-24
Using the WebFigure JSP Tag to Reference a WebFigure ..	8-26
Getting an Embeddable String That References a WebFigure Attached to a Cache	8-29

Working with MATLAB Figures and Images

9

Your Role in Working with Figures and Images	9-2
Create and Modify a MATLAB Figure	9-3
Preparing a MATLAB Figure for Export	9-3
Changing the Figure (Optional)	9-3
Exporting the Figure	9-4
Cleaning Up the Figure Window	9-4
Modify and Export Figure Data	9-5
Working with MATLAB Figure and Image Data	9-6
For More Comprehensive Examples	9-6
Working with Figures	9-6
Working with Images	9-7

Creating Scalable Web Applications Using RMI

10

Using Remote Method Invocation (RMI)	10-2
RMI Prerequisites	10-3
Run the Client and Server on a Single Machine	10-4
Run the Client and Server on Separate Machines	10-8
Use Native Java with Cell Arrays and Struct Arrays ..	10-9
Why Use Native Type Cell Arrays and Struct Arrays?	10-9
Native Type Data Marshaling Prerequisites	10-10
Native Java Cell and Struct	10-10
Additional RMI Examples	10-16

Troubleshooting

11

Common MATLAB Builder JA Error Messages	11-2
-----------------------------------------------	------

Reference Information for Java

12

Requirements for the MATLAB Builder JA Product ...	12-2
System Requirements	12-2
Path Modifications Required for Accessibility	12-2
MATLAB Builder JA Limitations	12-3
Data Conversion Rules	12-4
Java to MATLAB Conversion	12-4

MATLAB to Java Conversion	12-6
Unsupported MATLAB Array Types	12-7
Programming Interfaces Generated by the MATLAB	
Builder JA Product	12-8
APIs Based on MATLAB Function Signatures	12-8
Standard API	12-9
mlx API	12-11
Code Fragment: Signatures Generated for the myprimes	
Example	12-11
MWArray Class Specification	12-13
Application Deployment Terms	12-14

Function Reference

13

Using MATLAB Compiler on Mac or Linux

A

Overview	A-2
Installing MATLAB Compiler on Mac or Linux	A-3
Installing MATLAB Compiler	A-3
Selecting Your gcc Compiler	A-3
Custom Configuring Your Options File	A-3
Install Apple Xcode from DVD on Maci64	A-3
Writing Applications for Mac or Linux	A-4
Objective-C/C++ Applications for Apple's Cocoa API	A-4
Where's the Example Code?	A-4
Preparing Your Apple Xcode Development Environment ..	A-4
Build and Run the Sierpinski Application	A-5
Running the Sierpinski Application	A-7

Building Your Application on Mac or Linux	A-10
Compiling Your Application with the Deployment Tool ...	A-10
Compiling Your Application with the Command Line	A-10
Testing Your Application on Mac or Linux	A-11
Set MCR Paths on Mac or Linux with Scripts	A-12
Solving Problems Related to Setting MCR Paths on Mac or Linux	A-12

Index

Getting Started

- “MATLAB® Builder™ JA Product Description” on page 1-2
- “Determine Appropriate Tasks for MATLAB® Compiler™ and Builder Products” on page 1-3
- “Roles in the Java Application Deployment Process” on page 1-5
- “Configure Your Environment” on page 1-7
- “Create a Java Package from MATLAB Code” on page 1-10
- “Integrate a Java Package into an Application” on page 1-16

MATLAB Builder JA Product Description

Deploy MATLAB® code as Java classes

MATLAB Builder™ JA enables you to create Java® classes from your MATLAB programs. These Java classes can be integrated into Java programs and deployed royalty-free to desktop computers or Web servers that do not have MATLAB installed using the MATLAB Compiler Runtime (MCR) that is provided with MATLAB Compiler™.

When used with MATLAB Compiler, the builder creates deployable components that make MATLAB based computations, visualizations, and user interfaces accessible to end users of the Java programs. When the Java program is deployed to the Web, multiple users can access it through a Web browser.

The builder encrypts your MATLAB functions and generates a Java wrapper around them so that they behave just like any other Java class. Java classes created with MATLAB Builder JA are portable and run on all platforms supported by MATLAB.

Key Features

- Royalty-free desktop and Web deployment of Java classes
- MATLAB figure zooming, rotating, and panning via the Web Figures interface
- Ability to port classes not containing MEX-files to all MATLAB supported platforms
- API for automatic conversion between Java and MATLAB data types

Determine Appropriate Tasks for MATLAB Compiler and Builder Products

MATLAB Compiler compiles MATLAB code into standalone applications, libraries that can be integrated into other applications, or into deployable archives for use with MATLAB Production Server™. By default, MATLAB Compiler can generate standalone applications, C/C++ shared libraries, and deployable archives for use with MATLAB Production Server. Additional builders are available for Java, .NET, and Microsoft® Excel®.

While MATLAB Compiler lets you run your MATLAB application outside the MATLAB environment, it is not appropriate for all external tasks you may want to perform. Some tasks require either the MATLAB Coder™ product or MATLAB external interfaces. Use the following table to determine if MATLAB Compiler and builder products are appropriate to your needs.

MATLAB Compiler Task Matrix

Task	MATLAB Compiler and Builders	MATLAB Coder	MATLAB External Interfaces
Package MATLAB applications for deployment to users who do not have MATLAB	■		
Package MATLAB applications for deployment to MATLAB Production Server	■		
Build non-MATLAB applications that include MATLAB functions	■		
Generate readable, efficient, and embeddable C code from MATLAB code		■	

MATLAB Compiler Task Matrix (Continued)

Task	MATLAB Compiler and Builders	MATLAB Coder	MATLAB External Interfaces
Generate MEX functions from MATLAB code for rapid prototyping and verification of generated C code within MATLAB		■	
Integrate MATLAB code into Simulink®		■	
Speed up fixed-point MATLAB code		■	
Generate hardware description language (HDL) from MATLAB code		■	
Integrate custom C code into MATLAB with MEX files			■
Call MATLAB from C and Fortran programs			■

For information on MATLAB Coder see “MATLAB Coder”.

For information on MATLAB external interfaces see “External Programming Language Interfaces”.

Roles in the Java Application Deployment Process

Deploying MATLAB functionality through Java applications is a multistep process that may involve one or more team members. Each step requires that you perform a specific role, as shown in Java Application Deployment Roles on page 1-5.

Java Application Deployment Roles

Role	Knowledge Base	Responsibilities
MATLAB programmer	<ul style="list-style-type: none"> • MATLAB expert • Little to no Java knowledge • No IT experience 	<ul style="list-style-type: none"> • Understand end-user business requirements and the mathematical models needed to support them. • Write MATLAB code. • Build a Java package with MATLAB tools. • Pass the package to the Java developer.
Java developer	<ul style="list-style-type: none"> • Little to no MATLAB experience • Some knowledge of IT systems • Java expert 	<ul style="list-style-type: none"> • Write Java code to execute the Java classes built by the MATLAB programmer. • Address data conversion issues that may be encountered. • Ensure the final Java application executes

Java Application Deployment Roles (Continued)

Role	Knowledge Base	Responsibilities
		reliably in the end user's environment.
IT professional	<ul style="list-style-type: none">• Little to no MATLAB experience• Moderate IT experience• Familiarity with IT systems	<ul style="list-style-type: none">• Ensure that systems using the application have the required specifications.• Install any required software on target machines.• Install the application on target machines.

Configure Your Environment

In this section...

“Install the Required JDK” on page 1-7

“Set JAVA_HOME” on page 1-8

“Set the CLASSPATH” on page 1-8

“Configure the Native Library Path Variables” on page 1-9

Before you can compile MATLAB functions into Java packages or use the generated Java packages in a Java development environment, you need to ensure that your Java environment is properly configured. You must verify that:

- Your system uses the same version of the Java Developer’s Kit (JDK) as MATLAB.
- JAVA_HOME is set to the folder containing the system’s JDK installation.
- CLASSPATH contains all of the MATLAB library JARs and the JARs for the packages containing your compiled MATLAB code.
- The MATLAB native library paths are properly configured.

Note For updated Java system requirements, including versions of Java Developer’s Kit (JDK) and Java Runtime Environment (JRE), see the supported compiler page at http://www.mathworks.com/support/compilers/current_release/.

Install the Required JDK

To install the proper version of the JDK:

- 1 Verify the version of Java your MATLAB installation is using by running the following MATLAB command:

```
version -java
```

- 2 Download the matching version Java Developer's Kit (JDK) from <http://www.oracle.com/us/technologies/java/overview/index.html>.
- 3 Install the JDK, following the instructions provided by Oracle®.

Note If you are not developing applications or compiling MATLAB code, you can use the Java Runtime Environment (JRE) instead of the JDK.

Set JAVA_HOME

- 1 Set the system environment variable, `JAVA_HOME`, to point to your JDK installation.
- 2 At the MATLAB command prompt, type `getenv JAVA_HOME` to verify that MATLAB is reading the correct version of `JAVA_HOME`.
- 3 Verify that the folder containing your Java installation has been added to your system `PATH` environment variable.

Set the CLASSPATH

To build and run a Java application that uses a MATLAB Builder JA generated component, the system must locate:

- JAR files containing the MATLAB libraries
- Packages that you have developed and built with the builder

Java classes compiled by the builder use classes contained in the `com.mathworks.toolbox.javabuilder` package. To use the compiled classes, you need to include a file called `javabuilder.jar` on the Java class path. You can find this file in one of the following folders:

MATLAB installed on your system	<code>matlabroot/toolbox/javabuilder/jar</code>
MATLAB Compiler Runtime installed on your system	<code>mcrroot/toolbox/javabuilder/jar</code>

Note *matlabroot* refers to the root folder into which the MATLAB installer has placed the MATLAB files. *mcrroot* refers to the root folder under which MCR is installed.

In addition, you need to add to the JAR files created by the builder to the class path.

Configure the 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
Macintosh	DYLD_LIBRARY_PATH

The native MATLAB or MCR files needed to execute the compiled MATLAB functions called from the Java code must be included on the paths listed by your system's native library path variable.

Create a Java Package from MATLAB Code

This example shows how to create a Java package using a MATLAB function. You can then pass the generated package to the developer, who is responsible for integrating it into an application.

To compile a Java package from MATLAB code:

- 1** In MATLAB, examine the MATLAB code that want to deploy as a shared library.

- a** Open `makesqr.m`.

```
function y = makesqr(x)
```

```
y = magic(x);
```

- b** At the MATLAB command prompt, enter `makesqr(5)`.

The output appears as follows:

```
ans =
```

```
    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
```

- 2** Open the **Library Compiler**.

- a** On the toolstrip, select the **Apps** tab.
- b** Click the arrow at the far right of the tab to open the apps gallery.
- c** Click **Library Compiler** to open the **MATLAB Compiler** project window.

MATLAB Compiler - Untitled1.prj

DEPLOYMENT

New Save


C Shared Library
C++ Shared Library
Generic CTF
Excel Add-in

Add Exported Functions

Runtime downloaded from web MyA
 Runtime included in package MyA

FILE APPLICATION TYPE EXPORTED FUNCTIONS PACKAGING OF

Application Information







[Save contact info](#)

▶ Additional Installer Options

Files required for your application to run

Files installed with your application

 .dll  .h  .lib  readme.txt

- 3** In the **Application Type** section of the toolstrip, select **Java Package** from the list.

Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

- 4** Specify the MATLAB functions you want to deploy.
 - a** In the **Exported Functions** section of the toolstrip, click the plus button.

Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

- b** In the file explorer that opens, locate and select the `makesqr.m` file.

`makesqr.m` is located in
`matlabroot\toolbox\javabuilder\Examples\MagicSquareExample\MagicDemoComp.`

- c** Click **Open** to select the file, and close the file explorer.

makesqr.m is added to the list of exported files and a minus button appears under the plus button. In addition, `makesqr` is set as:

- the application name
- the package name

- 5** Verify that the function defined in `makesqr.m` is mapped into `Class1`.

makesqr	
Class Name	Method Name
Ⓢ Class1	Ⓜ makesqr.m
Add Class	

- 6** In the **Packaging Options** section of the toolstrip, verify that the **Runtime downloaded from web** check box is selected.

Note If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

This option creates an application installer that automatically downloads the MATLAB Compiler Runtime (MCR) and installs it along with the deployed package.

- 7** Explore the main body of the **MATLAB Compiler** project window.

The project window is divided into the following areas:

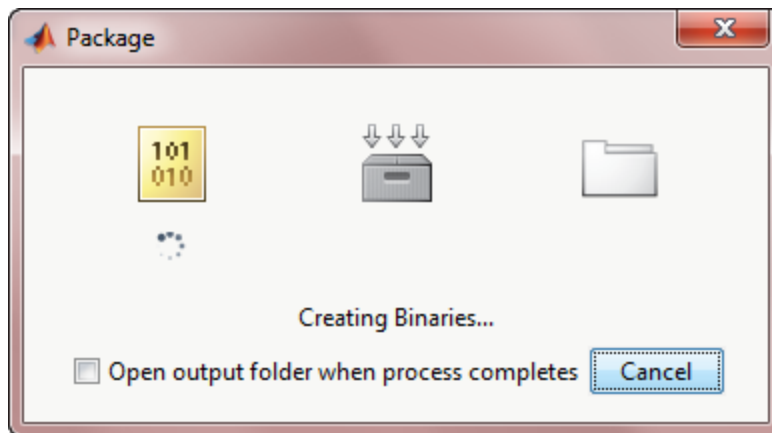
- **Application Information** — Editable information about the deployed application. This information is used by the generated installer to populate the installed application’s metadata. See “Customizing the Installer” on page 5-2.
- **Class mapper** — Description of how the MATLAB functions are mapped to Java classes.
- **Additional Installer Options** — Default installation path for the generated installer. See “Customizing the Installer” on page 5-2.
- **Files required for your application** — Additional files required by the generated application. These files will be included in the generated application installer. See “Manage the Required Files Compiled into a Project” on page 5-6.

- **Files installed with your application** — Files that are installed with your application. These files include:
 - `readme.txt`
 - `.jar` file
 - `doc` folder

See “Specify Additional Files to Be Installed with the Application” on page 5-8.

8 Click **Package**.

The Package window opens while the package is being generated.



9 Select the **Open output folder when process completes** check box.

10 Verify that the generated output contains:

- `for_redistribution` — A folder containing the installer to distribute the package
- `for_testing` — A folder containing the raw generated files to create the installer
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the package

11 Click **Close** on the Package window.

To follow up on this example:

- Try running the function from the command line as follows:
 - 1** Open a system command prompt.
 - 2** Navigate to the `for_testing/application` folder of the generated deployment project.
 - 3** Enter the following command:

```
java -classpath  
matlabroot\toolbox\javabuilder\jar\javabuilder.jar";makesqr.jar  
makesqr.Class1 5
```

matlabroot is the full path to your MATLAB installation.

- Try creating a package that consists of more than one function.
- Try “Integrate a Java Package into an Application” on page 1-16

Integrate a Java Package into an Application

This example shows how to invoke a MATLAB generated method in a Java application.

To create a Java application that calls a MATLAB generated method:

- 1** Install the MATLAB Compiler Runtime (MCR) and generated JARs in one of the following ways:

- Run the installer generated by MATLAB. It is located in the `for_redistribution` folder of the deployment project.

Doing so automatically installs the MCR from the Web and places the generated JARs onto your computer.

- Manually install the MCR and the generated JARs onto your development system.

You can download the MCR installer from <http://www.mathworks.com/products/compiler/mcr>. The generated JARs are located in the MATLAB deployment project's `for_testing` folder.

- 2** In the folder containing the generated JARs, create a new file called `getmagic.java`.

- 3** Using a text editor, open `getmagic.java`.

- 4** Place the following as the first line in the file.

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

This statement imports the MATLAB support classes.

- 5** Place the following line after the first import statement.

```
import makesqr.*;
```

This statement imports the classes generated by the compiler.

- 6** Add the following class definition.

```
class getmagic
```



```
{
}
```

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

- 7** Add the main() method to the application.

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

- 8** Add the following code to the top of the main() method.

```
MWNumericArray n = null;
Object[] result = null;
Class1 theMagic = null;
```

This initializes the variables used by the application.

- *n* is an instance of the MATLAB MWNumericArray class that MATLAB uses for its internal data format.
- *result* is a generic Java object that holds the results of the call to MATLAB.
- *theMagic* is an instance class generated from the MATLAB function.

- 9** Add the following code after the variable initialization.

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

This is a simple check to ensure that the required command-line argument was passed to the application.

- 10** Add a try/catch/finally block after the argument check.

- 11** In the try section of the try/catch/finally block, add the following code.

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

The code instantiates an instance of `MWNumericArray` and populates it with a 1-by-1 array containing the integer passed to the application on the command line. The value is converted to a `Double` because that is the most direct mapping between the Java and MATLAB internal data representation.

- 12** After the code instantiating the input parameter, add the following to instantiate the class generated from MATLAB.

```
theMagic = new Class1();
```

The constructor for the generated class handles all of the setup required to start the MCR and populate it with the required MATLAB code.

- 13** Using the newly instantiated object, call the MATLAB function.

```
result = theMagic.makesqr(1, n);  
System.out.println(result[0]);
```

- 14** Add the following catch section to the try/catch/finally block to handle any exceptions that might be thrown.

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

- 15** Add the following finally section to the try/catch/finally block to clean up any resources.

```
finally  
{  
    MWArray.disposeArray(n);  
    MWArray.disposeArray(result);  
    theMagic.dispose();  
}
```

The `disposeArray()` and `dispose()` methods clean up the resources used by the generated MATLAB code.

- 16** Save the Java file.

The completed Java file should resemble the following.

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

class getmagic
{
    public static void main(String[] args)
    {
        MWNumericArray n = null;
        Object[] result = null;
        Class1 theMagic = null;

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

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

            theMagic = new Class1();

            result = theMagic.makesqr(1, n);
            System.out.println(result[0]);
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
        finally
        {
            MWArray.disposeArray(n);
            MWArray.disposeArray(result);
            theMagic.dispose();
        }
    }
}
```

```
}
```

- 17** Use the system's command line to navigate to the folder where you installed the generated Java package and saved the new Java file.
- 18** Compile the Java application using `javac`.

```
javac -classpath  
mcrroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar  
.\getmagic.java
```

Note On UNIX® platforms, use colon (:) as the class path delimiter instead of semicolon (;).

mcrroot is the path to where the MCR is installed on your system. If you have MATLAB installed on your system instead, you can use the path to your MATLAB installation.

- 19** From the system's command prompt, run the application.

```
java -classpath .;"c:\Program Files\MATLAB\MATLAB Compiler Runtime\v82\tool  
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
```

You must be sure to place a dot (.) in the first position of the class path. If it not, you get a message stating that Java cannot load the class.

Note On UNIX platforms, use colon (:) as the class path delimiter instead of semicolon (;).

mcrroot is the path to where the MCR is installed on your system. If you have MATLAB installed on your system instead, you can use the path to your MATLAB installation.

To follow up on this example:

- Try installing the new application on a different computer.
- Try building an installer for the application.
- Try integrating a package that consists of multiple functions.

Overview

- “Product Overview” on page 2-2
- “Application Deployment Products and the Compiler Apps” on page 2-5
- “MATLAB® Builder™ JA Prerequisites” on page 2-13
- “Integrating a Generated Java Package into a Java Application” on page 2-15
- “Next Steps” on page 2-26

Product Overview

In this section...
“MATLAB® Compiler™ Extension” on page 2-2
“How MATLAB® Builder™ JA Works” on page 2-2
“How the MATLAB® Compiler™ and MATLAB® Builder™ JA Products Work Together” on page 2-4
“How Does Java Package Deployment Work?” on page 2-4
“Limitations of Support” on page 2-4

MATLAB Compiler Extension

MATLAB Builder JA enables you to create Java™ classes from your MATLAB® programs. These Java classes can be integrated into Java programs and deployed royalty-free to desktop computers or Web servers that do not have MATLAB installed.

When used with MATLAB Compiler™, the builder creates deployable classes that make MATLAB based computations, visualizations, and graphical user interfaces accessible to end users of the Java programs.

When the Java program is deployed to the Web, multiple users can access it through a Web browser.

The builder encrypts your MATLAB functions and generates a Java wrapper around them so that they behave just like any other Java class. Java classes created with MATLAB Builder JA are portable and run on all platforms supported by MATLAB. See the Platform Roadmap for MATLAB for more information.

For information about how MATLAB Compiler works, see “How Does MATLAB Compiler Software Build My Application?”

How MATLAB Builder JA Works

MATLAB Builder JA produces packages that depend on `javabuilder.jar`, which ship with the MATLAB Builder JA toolbox. `javabuilder.jar` requires

a matching version of the MCR be installed on the same machine running the Java application.

When the class contained within `javabuilder.jar` is instantiated for the first time, a series of events occur:

- 1** Dependent classes in `javabuilder.jar` are loaded.
- 2** The static initialization of dependent classes triggers the loading of a series of shared libraries (contained within the MATLAB Compiler Runtime (MCR)).

The shared libraries implement a number of native methods which form the bridge from the generated package to the MCR's implementation of the MATLAB language runtime.

- 3** Once the shared libraries are loaded, the MATLAB language runtime is initialized by creating an instance of a C++ class called `mcrInstance`.
- 4** The construction of `mcrInstance` triggers the initialization of many of the subsystems that comprise the MATLAB language runtime environment. One such subsystem is the MATLAB-Java language interface, which allows MATLAB programs to call Java code directly.

When the MCR's native code is loaded into a running JVM, as is the case with a MATLAB Builder JA generated package, the MATLAB-Java interface subsystem establishes a connection to the already running JVM by calling the JNI method `AttachCurrentThread`.

- 5** `AttachCurrentThread` creates a class loader that loads all classes needed by MATLAB code utilizing the MATLAB-Java interface. These include infrastructure classes required by the interface itself, as well as user-defined classes explicitly imported from MATLAB code.
- 6** The static constructors of the classes within `javabuilder.jar` install a shutdown hook to explicitly terminate the MCR's threads and release its resources. This process is predicated on the JVM entering the shutdown state and on all generated class instances being released via their `dispose` method. If you fail to call `dispose`, native threads running during shutdown will cause undefined and unexpected behavior in the JVM.

How the MATLAB Compiler and MATLAB Builder JA Products Work Together

The MATLAB Compiler product can compile MATLAB 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

How Does Java Package Deployment Work?

There are two kinds of deployment:

- Installing the generated packages 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 generated packages 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 packages 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 generated packages.

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

Limitations of Support

MATLAB Builder JA provides a wide variety of support for various Java types and objects. However, MATLAB objects are not supported as inputs or outputs for compiled or deployed functions.

Application Deployment Products and the Compiler Apps

In this section...

“What Is the Difference Between the Compiler Apps and the mcc Command Line?” on page 2-5

“How Does MATLAB® Compiler™ Software Build My Application?” on page 2-5

“Dependency Analysis Function” on page 2-8

“MEX-Files, DLLs, or Shared Libraries” on page 2-9

“Component Technology File (CTF Archive)” on page 2-9

What Is the Difference Between the Compiler Apps and the mcc Command Line?

When you use one of the compiler apps, you perform any function you would invoke using the MATLAB Compiler `mcc` command-line interface. The compiler apps’ interactive menus and dialogs build `mcc` commands that are customized to your specification. As such, your MATLAB code is processed the same way as if you were compiling it using `mcc`.

Compiler app advantages include:

- You perform related deployment tasks with a single intuitive interface.
- You maintain related information in a convenient project file.
- Your project state persists between sessions.
- You load previously stored compiler projects from a prepopulated menu.
- Package applications for distribution.

How Does MATLAB Compiler Software Build My Application?

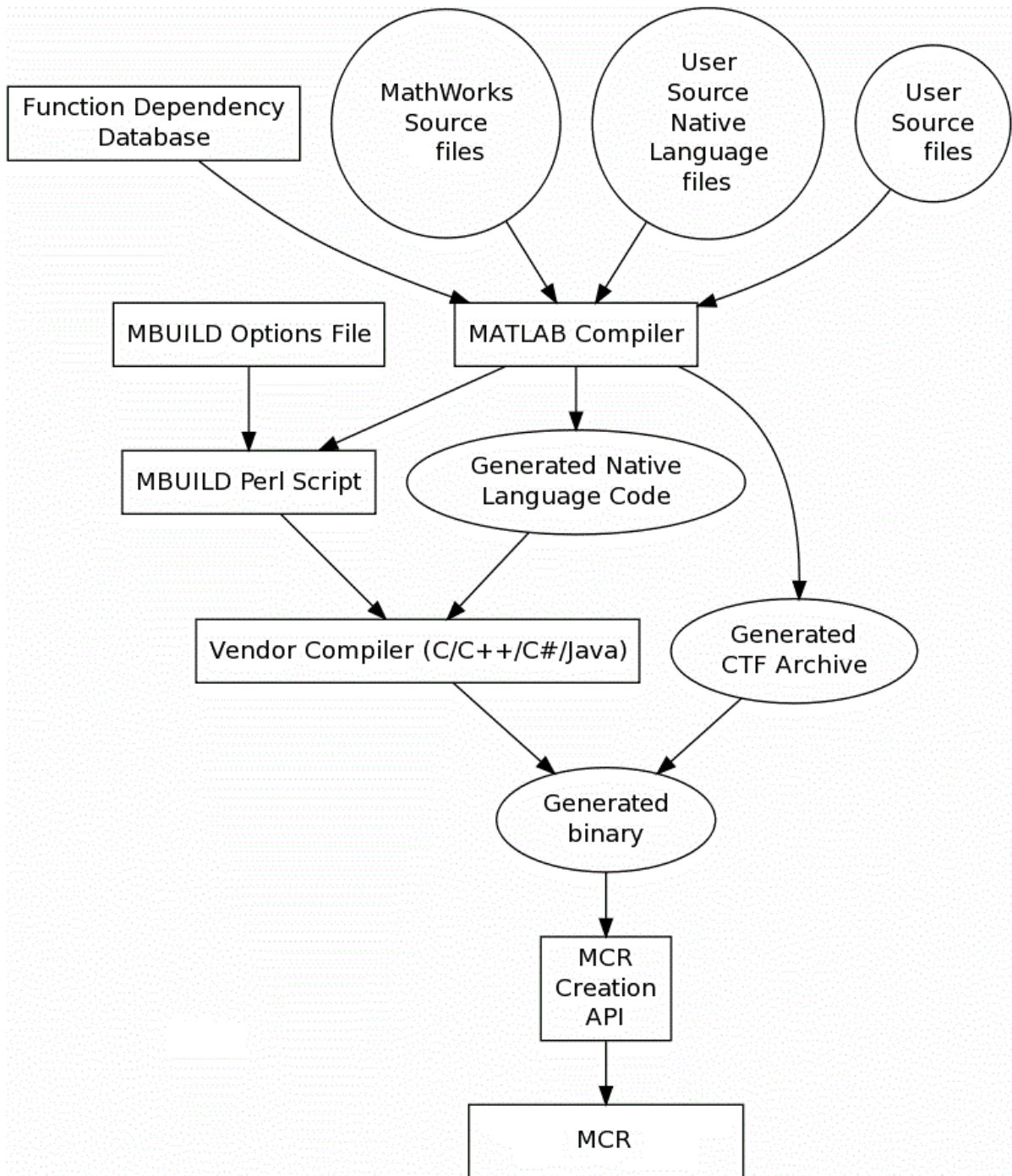
To build an application, MATLAB Compiler software performs these tasks:

- 1 Parses command-line arguments and classifies by type the files you provide.

2 Analyzes files for dependencies using a dependency analysis function. Dependencies affect deployability and originate from functions called by the file. Deployability is affected by:

- File type — MATLAB, Java, MEX, and so on.
- File location — MATLAB, MATLAB toolbox, user code, and so on.

For more information about how the compiler does dependency analysis, see “Dependency Analysis Function” on page 2-8.



- Circles: Input Files
- Boxes: MathWorks-supplied components

- 4** Creates a CTF archive from the input files and their dependencies. For more details about CTF archives see “Component Technology File (CTF Archive)” on page 2-9.
- 5** Generates target-specific wrapper code. For example, a C main function requires a very different wrapper than the wrapper for a Java interface class.
- 6** Generates target-specific binary package. For library targets such as C++ shared libraries, Java packages, or .NET assemblies, the compiler will invoke the required third-party compiler.

Dependency Analysis Function

MATLAB Compiler uses a dependency analysis function to determine the list of necessary files to include in the CTF package. Sometimes, this process generates a large list of files, particularly when MATLAB object classes exist in the compilation and the dependency analyzer cannot resolve overloaded methods at compile time. Dependency analysis also processes `include/exclude` files on each pass (see the `mcc` flag “-a Add to Archive”).

Tip To improve compile time performance and lessen application size, prune the path with “-N Clear Path”, “-p Add Folder to Path”. You can also specify **Files required for your application** in the compiler app.

For more information about dependency analysis, `addpath`, and `rmpath`, see “Dependency Analysis Function (depfun) and User Interaction with the Compilation Path”.

The dependency analyzer searches for executable content such as:

- MATLAB files
- P-files
- Java classes and `.jar` files
- `.fig` files
- MEX-files

The dependency analyzer does not search for data files of any kind. You must manually include data files in the search.

MEX-Files, DLLs, or Shared Libraries

When you compile MATLAB functions containing MEX-files, ensure that `depfun` can find them. Doing so allows you to avoid many common compilation problems. In particular, note that:

- Since the dependency analyzer cannot examine MEX-files, DLLs, or shared libraries to determine their dependencies, explicitly include all executable files these files require. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.
- If you have any doubts that `depfun` can find a MATLAB function called by a MEX-file, DLL, or shared library, then manually include that function. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.
- Not all functions are compatible with MATLAB Compiler. Check the file `mccExcludedFiles.log` after your build completes. This file lists all functions called from your application that you cannot deploy.

Component Technology File (CTF Archive)

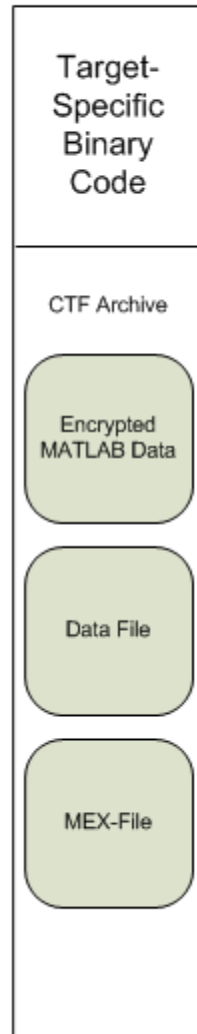
Each application or shared library you produce using MATLAB Compiler has an embedded Component Technology File (CTF) archive. The archive contains all the MATLAB based content (MATLAB files, MEX-files, and so on) associated with the component. All MATLAB files in the CTF archive are encrypted using the Advanced Encryption Standard (AES) cryptosystem.

If you choose to extract the CTF archive as a separate file, the files remain encrypted. For more information on how to extract the CTF archive refer to the references in the following table.

Information on CTF Archive Embedding/Extraction and Component Cache

Product	Refer to
MATLAB Compiler	“MCR Component Cache and CTF Archive Embedding”
MATLAB Builder NE	“MCR Component Cache and CTF Archive Embedding”
MATLAB Builder JA	“CTF Archive Embedding and Extraction” on page 6-64
MATLAB Builder EX	Using MCR Component Cache and CTF Archive Embedding

Generated Component (EXE, DLL, SO, etc)



Additional Details

Multiple CTF archives, such as those generated with COM, .NET, or Excel[®] components, can coexist in the same user application. You cannot, however, mix and match the MATLAB files they contain. You cannot combine encrypted and compressed MATLAB files from multiple CTF archives into another CTF archive and distribute them.

All the MATLAB files from a given CTF archive associate with a unique cryptographic key. MATLAB files with different keys, placed in the same CTF archive, do not execute. If you want to generate another application with a different mix of MATLAB files, recompile these MATLAB files into a new CTF archive.

MATLAB Compiler deletes the CTF archive and generated binary following a failed compilation, but only if these files did not exist before compilation initiates. Run `help mcc -K` for more information.

Note CTF archives are extracted by default to `temp\user_name\mcrCache.nn`.

Caution Release Engineers and Software Configuration Managers: Do not use build procedures or processes that strip shared libraries on CTF archives. If you do, you can possibly strip the CTF archive from the binary, resulting in run-time errors for the driver application.

MATLAB Builder JA Prerequisites

In this section...
“What You Need to Know” on page 2-13
“Required Products” on page 2-13
“Dependency and Non-Compilable Code Considerations” on page 2-13

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 developer, the following is required:
 - Exposure to the Java programming language
 - Object-oriented programming concepts

Required Products

You must install the following products to run the example described in this chapter:

- MATLAB
- MATLAB Compiler
- MATLAB Builder JA

Dependency and Non-Compilable Code Considerations

Before you deploy your code, examine the code for dependencies on functions that may not be compatible with MATLAB Compiler.

For more detailed information about dependency analysis (`depfun`) and how MATLAB Compiler evaluates MATLAB code prior to compilation, see “Write Deployable MATLAB Code” in the MATLAB Compiler documentation.

Integrating a Generated Java Package into a Java Application

In this section...
“Gathering Files Needed for Deployment” on page 2-16
“Testing the Java Package in a Java Application” on page 2-16
“Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)” on page 2-21
“Integrating Java Classes Generated by MATLAB into a Java Application” on page 2-23
“Calling Class Methods from Java” on page 2-24
“Handle Data Conversion as Needed” on page 2-24
“Build and Test” on page 2-25

Key Tasks

Task	Reference
Ensure you have the needed files from the MATLAB Programmer before proceeding.	“Gathering Files Needed for Deployment” on page 2-16
Test the Java code by using it in a Java application. Compile and run the package to ensure it produces the same results as your MATLAB code.	“Testing the Java Package in a Java Application” on page 2-16
Install the MATLAB Component Runtime (MCR) and update system paths.	“Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)” on page 2-21
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 2-23
Use built-in Java class methods to enhance your Java application.	“Calling Class Methods from Java” on page 2-24

Key Tasks (Continued)

Task	Reference
Address potential data conversion issues with differing data types.	“Handle Data Conversion as Needed” on page 2-24
Verify your Java application works as expected in your end user’s deployment environment.	“Build and Test” on page 2-25

Gathering Files Needed for Deployment

Before beginning, verify you have access to the installer created by the MATLAB Programmer in “Create a Java Package from MATLAB Code” on page 1-10. In addition to the generated installer, the following files are required to use the package:

- Javadoc documentation at `matlabroot/help/javabuilder/MWArrayAPI`.
- `readme.txt` file
- `com.mathworks.toolbox.javabuilder` at `matlabroot/help/javabuilder/MWArrayAPI`.
 - For 32-bit installations:
`matlabroot/toolbox/javabuilder/jar/javabuilder.jar`
 - For 64-bit installations:
`matlabroot/toolbox/javabuilder/jar/win64/javabuilder.jar`

Testing the Java Package in a Java Application

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

First, create a small Java program that uses the package created for you by the MATLAB Programmer (see “Integrate a Java Package into an Application” on page 1-16). The example provides a sample Java program that accomplishes this.

The program imports the `magicsquare` package you created with the library compiler app 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 MATLAB file becomes a method of the Java class that encapsulates it. The source code of `getmagic.java` follows, for your reference:

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

class getmagic
{
    public static void main(String[] args)
    {
        MWNumericArray n = null;
        Object[] result = null;
        Class1 theMagic = null;

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

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

            theMagic = new Class1();

            result = theMagic.makesqr(1, n);
            System.out.println(result[0]);
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
    }
}
```

```
    }  
    finally  
    {  
        MWArray.disposeArray(n);  
        MWArray.disposeArray(result);  
        theMagic.dispose();  
    }  
}  
}
```

Ensure your current working folder contains the application code. Then, do the following:

- 1 Compile the application with the Java compiler, `javac`. At the command prompt, enter one of the following commands. When entering these commands, ensure they are entered as one continuous command. On Windows systems, the semicolon (;) is a concatenation character. On UNIX systems, the colon (:) is a concatenation character.

- On Windows platforms:

```
javac -classpath  
mcrroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar  
.\getmagic.java
```

- On UNIX platforms:

```
javac -classpath  
mcrroot\toolbox\javabuilder\jar\javabuilder.jar":.\makesqr.jar  
.\getmagic.java
```

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

```
javac -classpath mcrroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr
```

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).

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 application.
- `matlabroot\toolbox\javabuilder\jar\javabuilder.jar`
— The location of the MATLAB Builder JA package file (`com.mathworks.toolbox.javabuilder`).
 - For 32-bit installations:
`matlabroot\toolbox\javabuilder\jar\javabuilder.jar`
 - For 64-bit installations:
`matlabroot\toolbox\javabuilder\jar\win64\javabuilder.jar`
- `.\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 command prompt. When entering these commands, ensure they are entered as one continuous command. On Windows systems, the semicolon (;) is a concatenation character. On UNIX systems, the colon (:) is a concatenation character.

- On Windows platforms:

```
java -classpath .;"c:\Program Files\MATLAB\MATLAB Compiler Runtime\v82\toolbox\javabuilder\jar\javabuilder.jar";.
```

- On UNIX platforms:

```
java -classpath .;"c:\Program Files\MATLAB\MATLAB Compiler Runtime\v82\toolbox\javabuilder\jar\javabuilder.jar";.
```

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

```
java -classpath .;"c:\Program Files\MATLAB\MATLAB Compiler
Runtime\v82\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar getmagic 5
```

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “MATLAB® Builder™ JA Limitations” on page 12-3 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.class`. 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.
 - For 32-bit installations:
`matlabroot\toolbox\javabuilder\jar\javabuilder.jar`
 - For 64-bit installations:
`matlabroot\toolbox\javabuilder\jar\win64\javabuilder.jar`
- `.\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 output. If the program ran successfully, a magic square of order 5 will print, matching the output of the MATLAB function, 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
```

Using `mcrroot` to Test Against the MCR

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_Compiler_Runtime\MCR_version_number`. Remember to double-quote all parts of the `java` command path arguments that contain spaces.

Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)

On target computers without MATLAB, install the MCR, if it is not already present on the deployment machine.

Install MATLAB Compiler Runtime (MCR)

The *MATLAB Compiler Runtime (MCR)* is an execution engine made up of the same shared libraries MATLAB uses to enable execution of MATLAB files on systems without an installed version of MATLAB.

The MATLAB Compiler Runtime (MCR) is now available for downloading from the Web to simplify the distribution of your applications or components created with the MATLAB Compiler. Download the MCR from the MATLAB Compiler Runtime product page.

The MCR installer does the following:

- 1** Installs the MCR (if not already installed on the target machine)
- 2** Installs the component assembly in the folder from which the installer is run
- 3** Copies the `MWArray` assembly to the Global Assembly Cache (GAC), as part of installing the MCR

MCR Prerequisites

- 1 Since installing the MCR requires write access to the system registry, ensure you have administrator privileges to run the MCR Installer.
- 2 The version of the MCR that runs your application on the target computer must be compatible with the version of MATLAB Compiler that built the component.
- 3 Do not install the MCR in MATLAB installation directories.
- 4 The MCR installer requires approximately 2 GB of disk space.

Add the MCR Installer to the Installer

This example shows how to include the MCR in the generated installer, using one of the compiler apps. The generated installer contains all files needed to run the standalone application or shared library built with MATLAB Compiler and properly lays them out on a target system.

- 1 On the **Packaging Options** section of the compiler interface, select one or both of the following options:
 - **Runtime downloaded from web** — This option builds an installer that invokes the MCR installer from the MathWorks Web site.
 - **Runtime included in package** — The option includes the MCR installer into the generated installer.
- 2 Click **Package**.
- 3 Distribute the installer as needed.

Install the MCR

This example shows how to install the MATLAB Compiler Runtime (MCR) on a system.

If you are given an installer containing the compiled artifacts, then the MCR is installed along with the application or shared library. If you are given just the raw binary files, download the MCR installer from the Web and run the installer.

Note If you are running on a platform other than Windows, set the system paths on the target machine. Setting the paths enables your application to find the MCR.

Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths. See “Using MATLAB Compiler on Mac or Linux” for detailed information on performing all deployment tasks specifically with UNIX variants such as Linux and Mac.

Integrating Java Classes Generated by MATLAB into a Java Application

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

- 1 Install the MATLAB Compiler Runtime on the target system. See “Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)” in the MATLAB Compiler documentation.
- 2 Consult the Javadoc for information on classes generated by MATLAB classes. Reference the Javadoc from *matlabroot/help/javabuilder/MWArrayAPI*.
- 3 To integrate the Java class generated by MATLAB Builder JA both the generated classes and the supporting MATLAB classes need to be imported. Import the MATLAB libraries and the generated classes into your code with the Java `import` function. For example:

```
import com.mathworks.toolbox.javabuilder.*;  
import packagename.classname; or import packagename.*;
```

For more information, see “Importing Classes ” on page 6-8.

- 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 “Creating an Instance of the Class” on page 6-9.

- 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 “Managing Native Resources” on page 6-40, in particular, “Using the dispose Method” on page 6-41.

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 packages. Detailed examples and complete listings of input parameters and possible thrown exceptions can be found in the Javadoc.

Handle Data Conversion as Needed

When you invoke a method on a generated class, 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.

How MATLAB Builder JA Handles Data

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.

For more detailed information on data handling within the product and programming with the `MWArray` package, see the `com.mathworks.toolbox.javabuilder.MWArray` Javadoc and “About the MATLAB® Builder™ JA API” on page 6-3.

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 in the chapters described in “Next Steps” on page 2-26.

Running a 64-Bit Mac Application

Before you run a 64-bit Macintosh application, you need to use the Macintosh Application Launcher. See in the *MATLAB Compiler User’s Guide* for more information.

See “Using MATLAB Compiler on Mac or Linux” in the *MATLAB Compiler User’s Guide* for complete information about building, deploying, and testing UNIX applications with MATLAB Compiler.

Next Steps

Writing Java applications that can access Java methods that encapsulate MATLAB code

“About the MATLAB® Builder™ JA API” on page 6-3

“Importing Classes ” on page 6-8

“Creating an Instance of the Class” on page 6-9

“Passing Arguments to and from Java” on page 6-13

“Passing Java Objects by Reference” on page 6-27

“Handling Errors” on page 6-33

“Managing Native Resources” on page 6-40

Sample applications that access methods developed in MATLAB

“Plot” on page 7-2

“Spectral Analysis” on page 7-9

“Matrix Math” on page 7-16

“Phone Book” on page 7-28

“Optimization” on page 7-36

“Web Application” on page 7-47

Deploying figures over the Web

“Implement a Custom WebFigure” on page 8-9

Reference information about automatic data conversion rules

“Data Conversion Rules” on page 12-4

MATLAB Code Guidelines

- “Write Deployable MATLAB Code” on page 3-2
- “Load MATLAB Libraries using loadlibrary” on page 3-7
- “Use MATLAB Data Files (MAT Files) in Compiled Applications” on page 3-9

Write Deployable MATLAB Code

In this section...

“Compiled Applications Do Not Process MATLAB Files at Runtime” on page 3-2

“Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files” on page 3-3

“Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths” on page 3-4

“Gradually Refactor Applications That Depend on Noncompilable Functions” on page 3-4

“Do Not Create or Use Nonconstant Static State Variables” on page 3-5

“Get Proper Licenses for Toolbox Functionality You Want to Deploy” on page 3-5

Compiled Applications Do Not Process MATLAB Files at Runtime

MATLAB Compiler secures your code against unauthorized changes. Deployable MATLAB files are suspended or frozen at the time MATLAB Compiler encrypts them—they do not change from that point onward. This does not mean that you cannot deploy a flexible application—it means that *you must design your application with flexibility in mind*. If you want the end user to be able to choose between two different methods, for example, both methods must be available in the built component.

The MCR only works on MATLAB code that was encrypted when the component was built. Any function or process that dynamically generates new MATLAB code will not work against the MCR.

Some MATLAB toolboxes, such as the Neural Network Toolbox™ product, generate MATLAB code dynamically. Because the MCR only executes encrypted MATLAB files, and the Neural Network Toolbox generates unencrypted MATLAB files, some functions in the Neural Network Toolbox cannot be deployed.

Similarly, functions that need to examine the contents of a MATLAB function file cannot be deployed. `HELP`, for example, is dynamic and not available in deployed mode. You can use `LOADLIBRARY` in deployed mode if you provide it with a MATLAB function prototype.

Instead of compiling the function that generates the MATLAB code and attempting to deploy it, perform the following tasks:

- 1 Run the code once in MATLAB to obtain your generated function.
- 2 Compile the MATLAB code with MATLAB Compiler, including the generated function.

Tip Another alternative to using `EVAL` or `FEVAL` is using anonymous function handles.

If you require the ability to create MATLAB code for dynamic run time processing, your end users must have an installed copy of MATLAB.

Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files

In general, good programming practices advise against redirecting a program search path dynamically within the code. Many developers are prone to this behavior since it mimics the actions they usually perform on the command line. However, this can lead to problems when deploying code.

For example, in a deployed application, the MATLAB and Java paths are fixed and cannot change. Therefore, any attempts to change these paths (using the `cd` command or the `addpath` command) fails

If you find you cannot avoid placing `addpath` calls in your MATLAB code, use `ismcc` and `isdeployed`. See the next section for details.

Use `ismcc` and `isdeployed` Functions To Execute Deployment-Specific Code Paths

The `isdeployed` function allows you to specify which portion of your MATLAB code is deployable, and which is not. Such specification minimizes your compilation errors and helps create more efficient, maintainable code.

For example, you find it unavoidable to use `addpath` when writing your `startup.m`. Using `ismcc` and `isdeployed`, you specify when and what is compiled and executed.

For an example of using `isdeployed`, see “Passing Arguments to and from a Standalone Application”.

Gradually Refactor Applications That Depend on Noncompilable Functions

Over time, refactor, streamline, and modularize MATLAB code containing non-compilable or non-deployable functions that use `ismcc` and `isdeployed`. Your eventual goal is “graceful degradation” of non-deployable code. In other words, the code must present the end user with as few obstacles to deployment as possible until it is practically eliminated.

Partition your code into design-time and run time code sections:

- *Design-time code* is code that is currently evolving. Almost all code goes through a phase of perpetual rewriting, debugging, and optimization. In some toolboxes, such as the Neural Network Toolbox product, the code goes through a period of self-training as it reacts to various data permutations and patterns. Such code is almost never designed to be deployed.
- *Run-time code*, on the other hand, has solidified or become stable—it is in a finished state and is ready to be deployed by the end user.

Consider creating a separate directory for code that is not meant to be deployed or for code that calls undeployable code.

Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

- Global variables in MATLAB code
- Static variables in MEX-files
- Static variables in Java code

The state of these variables is persistent and shared with everything in the process.

When deploying applications, using persistent variables can cause problems because the MCR process runs in a single thread. You cannot load more than one of these non-constant, static variables into the same process. In addition, these static variables do not work well in multithreaded applications.

When programming with builder components, you should be aware that an instance of the MCR is created for each instance of a new class. If the same class is instantiated again using a different variable name, it is attached to the MCR created by the previous instance of the same class. In short, if an assembly contains n unique classes, there will be maximum of n instances of MCRs created, each corresponding to one or more instances of one of the classes.

If you must use static variables, bind them to instances. For example, defining instance variables in a Java class is preferable to defining the variable as `static`.

Note This guideline does not apply to MATLAB Builder EX. When programming with Microsoft Excel, you can assign global variables to large matrices that persist between calls.

Get Proper Licenses for Toolbox Functionality You Want to Deploy

You must have a valid MathWorks® license for toolboxes you use to create deployable components.

If you do not have a valid license for your toolbox, you cannot create a deployable component with it.

Load MATLAB Libraries using loadlibrary

Note It is important to understand the difference between the following:

- MATLAB `loadlibrary` function — Loads shared library into MATLAB.
 - Operating system `loadlibrary` function — Loads specified Windows or UNIX operating system module into the address space of the calling process.
-

With MATLAB Compiler version 4.0 (R14) and later, you can use MATLAB file prototypes as described below to load your library in a compiled application. Loading libraries using H-file headers is not supported in compiled applications. This behavior occurs when `loadlibrary` is compiled with the header argument as in the statement:

```
loadlibrary(library, header)
```

In order to work around this issue, execute the following at the MATLAB command prompt:

```
loadlibrary(library, header, 'mfilename', 'mylibrarymfile');
```

where *mylibrarymfile* is the name of a MATLAB file you would like to use when loading this library. This step only needs to be performed once to generate a MATLAB file for the library.

In the code that is to be compiled, you can now call `loadlibrary` with the following syntax:

```
loadlibrary(library, @mylibrarymfile, 'alias', alias)
```

It is only required to add the prototype `.m` file and `.dll` file to the CTF archive of the deployed application. There is no need for `.h` files and C/C++ compilers to be installed on the deployment machine if the prototype file is used.

Once the prototype file is generated, add the file to the CTF archive of the application being compiled. You can do this with the `-a` option (if using the

`mcc` command) or by dragging it under **Other/Additional Files** (as a helper file) if using the Deployment Tool.

With MATLAB Compiler versions 4.0.1 (R14+) and later, generated MATLAB files will automatically be included in the CTF file as part of the compilation process. For MATLAB Compiler versions 4.0 (R14) and later, include your library MATLAB file in the compilation with the `-a` option with `mcc`.

Restrictions on Using MATLAB Function `loadlibrary` with MATLAB Compiler

Note the following limitations in regards to using `loadlibrary` with MATLAB Compiler. For complete documentation and up to date restrictions on `loadlibrary`, please reference the MATLAB documentation.

- You can not use `loadlibrary` inside of MATLAB to load a shared library built with MATLAB Compiler.
- With MATLAB Compiler Version 3.0 (R13SP1) and earlier, you cannot compile calls to `loadlibrary` because of general restrictions and limitations of the product.

Use MATLAB Data Files (MAT Files) in Compiled Applications

In this section...

“Explicitly Including MAT files Using the `%#function` Pragma” on page 3-9

“Load and Save Functions” on page 3-9

“MATLAB Objects” on page 3-12

Explicitly Including MAT files Using the `%#function` Pragma

MATLAB Compiler excludes MAT files from “Dependency Analysis Function” on page 2-8 by default.

If you want MATLAB Compiler to explicitly inspect data within a MAT file, you need to specify the `%#function` pragma when writing your MATLAB code.

For example, if you are creating a solution with Neural Network Toolbox, you need to use the `%#function` pragma within your GUI code to include a dependency on the `gmdistribution` class, for instance.

Load and Save Functions

If your deployed application uses MATLAB data files (MAT-files), it is helpful to code `LOAD` and `SAVE` functions to manipulate the data and store it for later processing.

- Use `isdeployed` to determine if your code is running in or out of the MATLAB workspace.
- Specify the data file by either using `WHICH` (to locate its full path name) define it relative to the location of `ctfroot`.
- All MAT-files are unchanged after `mcc` runs. These files are not encrypted when written to the CTF archive.

For more information about CTF archives, see “Component Technology File (CTF Archive)” on page 2-9.

See the `ctfroot` reference page for more information about `ctfroot`.

Use the following example as a template for manipulating your MATLAB data inside, and outside, of MATLAB.

Using Load/Save Functions to Process MATLAB Data for Deployed Applications

The following example specifies three MATLAB data files:

- `user_data.mat`
- `userdata\extra_data.mat`
- `..\externdata\extern_data.mat`

1 Navigate to `matlab_root\extern\examples\compiler\Data_Handling`.

2 Compile `ex_loadsave.m` with the following `mcc` command:

```
mcc -mv ex_loadsave.m -a 'user_data.mat' -a
    '\userdata\extra_data.mat' -a
    '..\externdata\extern_data.mat'
```

ex_loadsave.m.

```
function ex_loadsave
% This example shows how to work with the
% "load/save" functions on data files in
% deployed mode. There are three source data files
% in this example.
%   user_data.mat
%   userdata\extra_data.mat
%   ..\externdata\extern_data.mat
%
% Compile this example with the mcc command:
%   mcc -m ex_loadsave.m -a 'user_data.mat' -a
%     '\userdata\extra_data.mat'
%     -a '..\externdata\extern_data.mat'
% All the folders under the current main MATLAB file directory will
% be included as
% relative path to ctroot; All other folders will have the
% folder
% structure included in the ctf archive file from root of the
```

```

%      disk drive.
%
% If a data file is outside of the main MATLAB file path,
%      the absolute path will be
% included in ctf and extracted under ctffoot. For example:
%      Data file
%      "c:\$matlabroot\examples\externdata\extern_data.mat"
%      will be added into ctf and extracted to
%      "$ctffoot\$matlabroot\examples\externdata\extern_data.mat".
%
% All mat/data files are unchanged after mcc runs. There is
% no exryption on these user included data files. They are
% included in the ctf archive.
%
% The target data file is:
%      .\output\saved_data.mat
%      When writing the file to local disk, do not save any files
%      under ctffoot since it may be refreshed and deleted
%      when the application isnext started.

%==== load data file =====
if isdeployed
    % In deployed mode, all file under CTFRoot in the path are loaded
    % by full path name or relative to $ctffoot.
    % LOADFILENAME1=which(fullfile(ctffoot,mfilename,'user_data.mat'));
    % LOADFILENAME2=which(fullfile(ctffoot,'userdata','extra_data.mat'));
    LOADFILENAME1=which(fullfile('user_data.mat'));
    LOADFILENAME2=which(fullfile('extra_data.mat'));
    % For external data file, full path will be added into ctf;
    % you don't need specify the full path to find the file.
    LOADFILENAME3=which(fullfile('extern_data.mat'));
else
    %running the code in MATLAB
    LOADFILENAME1=fullfile(matlabroot,'extern','examples','compiler',
        'Data_Handling','user_data.mat');
    LOADFILENAME2=fullfile(matlabroot,'extern','examples','compiler',
        'Data_Handling','userdata','extra_data.mat');
    LOADFILENAME3=fullfile(matlabroot,'extern','examples','compiler',
        'externdata','extern_data.mat');
end

```

```
% Load the data file from current working directory
disp(['Load A from : ',LOADFILENAME1]);
load(LOADFILENAME1,'data1');
disp('A= ');
disp(data1);

% Load the data file from sub directory
disp(['Load B from : ',LOADFILENAME2]);
load(LOADFILENAME2,'data2');
disp('B= ');
disp(data2);

% Load extern data outside of current working directory
disp(['Load extern data from : ',LOADFILENAME3]);
load(LOADFILENAME3);
disp('ext_data= ');
disp(ext_data);

%==== multiple the data matrix by 2 =====
result = data1*data2;
disp('A * B = ');
disp(result);

%==== save the new data to a new file =====
SAVEPATH=strcat(pwd,filesep,'output');
if ( ~isdir(SAVEPATH))
    mkdir(SAVEPATH);
end
SAVEFILENAME=strcat(SAVEPATH,filesep,'saved_data.mat');
disp(['Save the A * B result to : ',SAVEFILENAME]);
save(SAVEFILENAME, 'result');
```

MATLAB Objects

When working with MATLAB objects, remember to include the following statement in your MAT file:

```
%#function class_constructor
```

Using the `%#function` pragma in this manner forces `depfun` to load needed class definitions, enabling the MCR to successfully load the object.

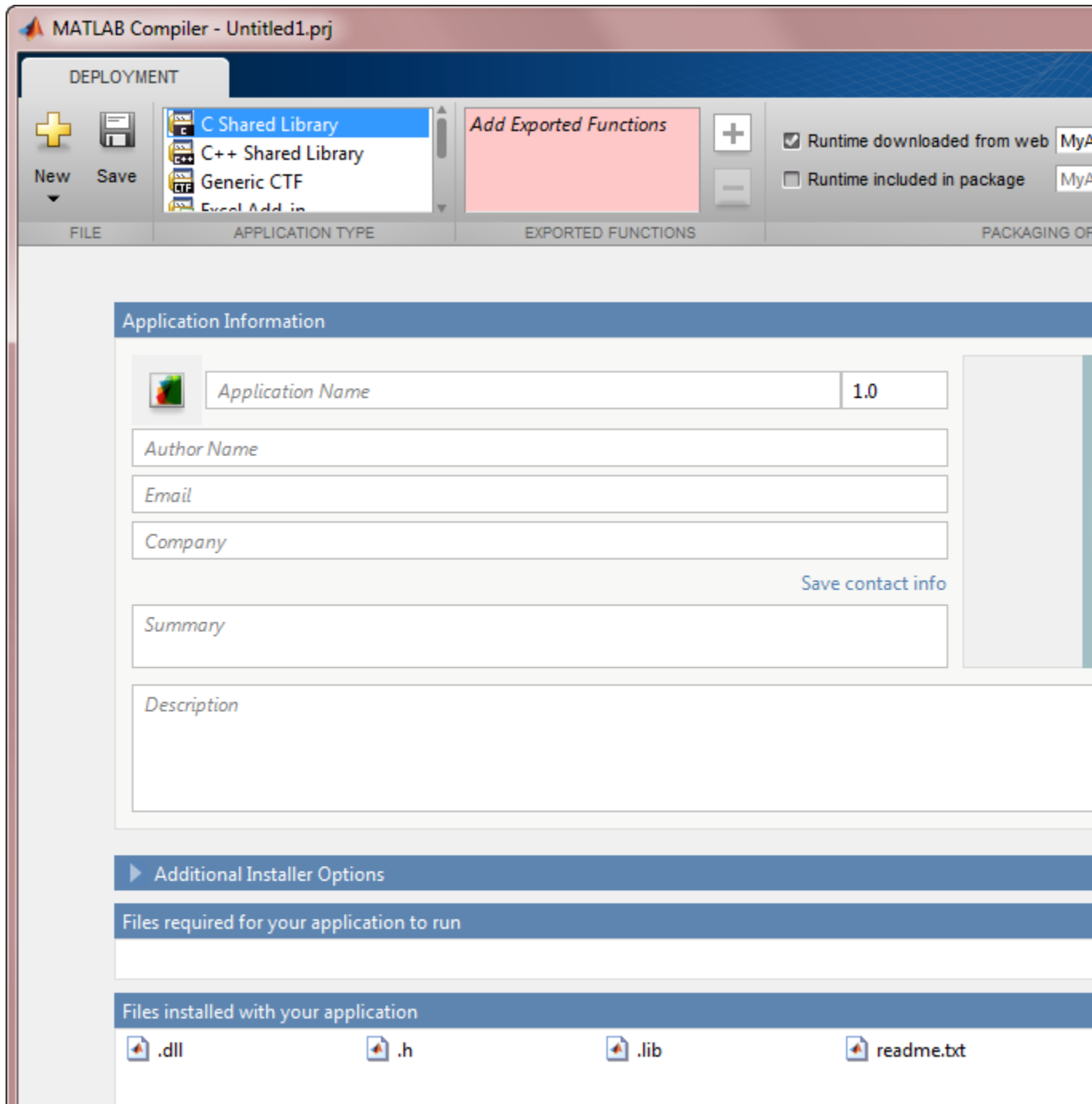
Deploying Java Packages

- “Compile a Java Package with the Library Compiler App” on page 4-2
- “Compile a Java Package from the Command Line” on page 4-8
- “Map Functions to Java Class Methods” on page 4-10

Compile a Java Package with the Library Compiler App

To compile MATLAB code into a Java package:

- 1** Open the **Library Compiler**.
 - a** On the toolbar select the **Apps** tab.
 - b** Click the arrow at the far right of the tab to open the apps gallery.
 - c** Click **Library Compiler** to open the **MATLAB Compiler** project window.



Note You can also start the shared library compiler using the `libraryCompiler` function.

- 2 In the **Application Type** section of the toolbar, select **Java Package**.

Note If the **Application Type** section of the toolbar is collapsed, expand it by clicking the down arrow.

- 3 Specify the MATLAB files you want deployed in the package.
 - a In the **Exported Functions** section of the toolbar, click the plus button.

Note If the **Exported Functions** section of the toolbar is collapsed, expand it by clicking the down arrow.

- b In the file explorer that opens, locate and select one or more the MATLAB files.
 - c Click **Open** to select the file and close the file explorer.

The names of the selected files are added to the list and a minus button appears below the plus button. The name of the first file listed is used as the default application name and the default package name.

- 4 Verify that the function defined in the selected files are properly mapped into classes.

makesqr	
Class Name	Method Name
Ⓢ Class1	Ⓢ makesqr.m
Add Class	

For more information, see “Map Functions to Java Class Methods” on page 4-10.

- In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Compiler Runtime (MCR) with the package.

Note If the **Packaging Options** section of the toolstrip is collapsed, expand it by clicking the down arrow.

You can select one or both of the following options:

- **Runtime downloaded from web** — Generates an installer that downloads the MCR installer from the Web.
- **Runtime included in package** — Generates an installer that includes the MCR installer.

Note Selecting both options creates two installers.

Regardless of the options selected the generated installer scans the target system to determine if there is an existing installation of the appropriate MCR. If there is not, the installer installs the MCR.

- Specify the name of any generated installers.

- 7** In the **Application Information** and **Additional Installer Options** sections of the compiler, customize the look and feel of the generated installer.

You can change the information used to identify the application data used by the installer:

- Splash screen
- Application icon
- Application version
- Name and contact information of the package’s author
- Brief summary of the package’s purpose
- Detailed description of the package

You can also change the default location into which the package is installed and provide some notes to the installer.

All of the provided information is displayed as the installer runs.

For more information, see “Customizing the Installer” on page 5-2.

- 8** In the **Files required for your application to run** section of the compiler, verify that the files required by the deployed MATLAB functions are listed.

Note These files are compiled into the generated binaries along with the exported files.

In general, the built-in dependency checker will automatically populate this section with the appropriate files. However, if needed you can manually add any files it missed.

For more information, see “Manage the Required Files Compiled into a Project” on page 5-6.

- 9** In the **Files installed with your application** section of the compiler, verify that any additional non-MATLAB files you want installed with the application are listed.

Note These files are placed in the `applications` folder of the installation.

This section automatically lists:

- Generated package
- `doc` folder containing the Javadoc for the generated classes
- Readme file

You can manually add files to the list. Additional files can include documentation, sample data files, and examples to accompany the application.

For more information, see “Specify Additional Files to Be Installed with the Application” on page 5-8.

- 10** Click the **Settings** button to customize the flags passed to the compiler and the folders to which the generated files are written.

Note To create a package that uses a singleton MCR, pass the `-S` flag to the compiler. For more information, see “Sharing an MCR Instance in COM or Java Applications” on page 6-53.

- 11** Click the **Package** button to compile the MATLAB code and generate any installers.

- 12** Verify that the generated output contains:

- `for_redistribution` — A folder containing the installer to distribute the package
- `for_testing` — A folder containing the raw generated files to create the installer
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the package

Compile a Java Package from the Command Line

In this section...
“Execute Compiler Projects with deploytool” on page 4-8
“Compile a Java Package with mcc” on page 4-8

You can compile Java packages from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes the compiler to execute a presaved compiler project
- `mcc` invokes the raw compiler

Execute Compiler Projects with `deploytool`

The `deploytool` command has two flags to invoke the compiler without opening a window:

- `-build project_name` — Invoke the compiler to build the project and do not generate an installer.
- `-package project_name` — Invoke the compiler to build the project and generate an installer.

For example, `deploytool -package magicsqaure` generates of the binary files defined by the `magicaquire` project and packages them into an installer that you can distribute to others.

Compile a Java Package with `mcc`

The `mcc` command invokes the raw compiler and provides fine-level control over the compilation of the Java package. It, however, cannot package the results in an installer.

To invoke the compiler to generate a Java package use the `-W java:packageName,className` flag with `mcc`. This flag creates a Java package named `packageName`. The package contains a class `className` with methods for each of the provided MATLAB functions.

For compiling Java packages, you can also use the following options.

Compiler Java Options

Option	Description
<code>-a filePath</code>	Add any files on the path to the generated binary.
<code>-d outFolder</code>	Specify the folder into which the results of compilation are written.
<code>-S</code>	Specify that the generated classes instantiate a singleton MCR.
<code>class{className:mfilename...}</code>	Specify that an additional class is generated that includes methods for the listed MATLAB files.

Map Functions to Java Class Methods

In this section...

“Use the Library Compiler App to Map Functions to Java Classes” on page 4-10

“Use mcc to Map Functions to Java Classes” on page 4-12

Use the Library Compiler App to Map Functions to Java Classes

The library compiler presents a visual class mapper for mapping MATLAB functions to Java classes. The class mapper is located between the **Application Information** and the **Additional Installer Options** sections of the interface.

Class Name	Method Name
 Class1	 makesqr.m

[Add Class](#)

The top field specifies the name of the package into which the generated classes are placed. By default, the name of the first listed MATLAB file is used as the package name. You can change the package name to fit the naming conventions used by your organization.

The table used to match functions to classes is below the package name. The **Class Name** column specifies the name of the generated Java class. The **Method Name** column specifies the list of MATLAB functions that are mapped into methods of the generated class.

Add a New Class to a Java Package

To add a class to a Java package:

- 1 Click **Add Class**.
- 2 Rename the class as described in “Rename a Java Class” on page 4-11.
- 3 Add one or more methods to the class as described in “Add a Method to a Java Class” on page 4-11.

Rename a Java Class

To rename a Java class:

- 1 Select the name of the class to be renamed.
- 2 Open the context menu.
- 3 Select **Rename**.
- 4 Enter the new class name.

The class name must follow the Java naming guidelines. It cannot contain any special characters, dots, or spaces.

Delete a Class from a Java Package

To delete a class from a Java package:

- 1 Select the name of the class to be deleted.
- 2 Open the context menu.
- 3 Select **Delete**.

Add a Method to a Java Class

To add a Method to a Java class:

- 1 In the **Method Name** column of the row for the class to which the method is being added, click the plus button.
- 2 Select the name of the function to add.

Delete a Method from a Java Class

To delete a method from a Java class:

- 1 Select the name of the function to be deleted.
- 2 Open the context menu.
- 3 Select **Delete**.

Tip You can also delete the method using the **Delete** key.

Use `mcc` to Map Functions to Java Classes

When using `mcc` to generate Java packages, you map your MATLAB functions into Java classes based on the list into which they are placed on the command line. Class groupings are specified by adding one or more `class{className:mfilename...}` entries to the command line. All of the files not specifically included in a class grouping are added to the class specified by the `-W java:packageName,className` flag.

For example, `mcc -W java:myPackage,MyClass fun1.m fun2.m fun3.m` generates a Java package `myPackage` that contains a single class `MyClass`. `MyClass` has three methods: `fun1`, `fun2`, and `fun3`.

However, `mcc -W java:myPackage,MyClass fun1.m fun2.m class{MyOtherClass:fun3.m}` generates a Java package `myPackage` that contains two classes: `MyClass` and `MyOtherClass`. `MyClass` has two methods: `fun1` and `fun2`. `MyOtherClass` has one method `fun3`.

Customizing a Compiler Project

- “Customizing the Installer” on page 5-2
- “Manage the Required Files Compiled into a Project” on page 5-6
- “Specify Additional Files to Be Installed with the Application” on page 5-8

Customizing the Installer

In this section...

- “Changing the Application Icon” on page 5-2
- “Adding Application Information” on page 5-3
- “Changing the Splash Screen” on page 5-4
- “Changing the Installation Path” on page 5-4
- “Changing the Application Logo” on page 5-5
- “Editing the Installation Notes” on page 5-5

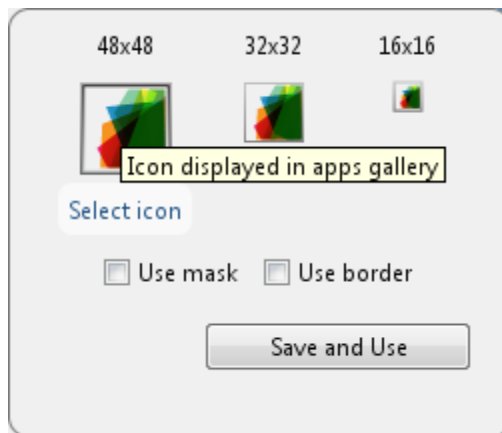
Changing the Application Icon

The application icon is used for the generated installer. For standalone applications, it is also the application’s icon.

You can change the default icon in **Application Information**. To set a custom icon:

- 1 Click the graphic to the left of the **Application name** field.

A window previewing the icon opens.



- 2 Click **Select icon**.
- 3 Using the file explorer, locate the graphic file to use as the application icon.
- 4 Select the graphic file.
- 5 Click **OK** to return to the icon preview.
- 6 Select **Use mask** to fill any blank spaces around the icon with white.
- 7 Select **Use border** to add a border around the icon.
- 8 Click **Save and Use** to return to the main compiler window.

Adding Application Information

The **Application Information** section of the compiler app allows you to provide these values:

- Application name

Determines the name of the installed executable file or shared library. For example, if the application name is `foo`, the installed executable would be `foo.exe`, the Windows start menu entry would be **foo**. The folder created for the application would be `InstallRoot/foo`.

The default value is the name of the first function listed in the **Main File(s)** field of the compiler.

- Application version

The default value is 1.0.

- Author name
- Support e-mail address
- Company name

Determines the full installation path for the installed executable or shared library. For example, if the company name is `bar`, the full installation path would be `InstallRoot/bar/ApplicationName`.

- Summary
- Description

This information is all optional and, unless otherwise stated, is only used for display purposes. It appears on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.

Changing the Splash Screen

The installer's splash screen displays after the installer is started. It is displayed, along with a status bar, while the installer initializes.

You can change the default image by clicking the **Select custom splash screen** link in **Application Information**. When the file explorer opens, locate and select a new image.

Note You can drag and drop a custom image onto the default splash screen.

Changing the Installation Path

Default Installation Paths on page 5-4 lists the default path the installer will use when installing the compiled binaries onto a target system.

Default Installation Paths

Windows	C:\Program Files\ <i>companyName</i> \ <i>appName</i>
Mac OS X	/Applications/ <i>companyName</i> / <i>appName</i>
Linux	/usr/ <i>companyName</i> / <i>appName</i>

You can change the default installation path by editing the **Default installation folder** field under **Additional Installer Options**.

The **Default installation folder** field has two parts:

- root folder — A drop down list that offers options for where the install folder is installed. Custom Installation Roots on page 5-5 lists the optional root folders for each platform.

Custom Installation Roots

Windows	C:\Users\userName\AppData
Linux	/usr/local

- **install folder** — A text field specifying the path appended to the root folder.

Changing the Application Logo

The application logo displays after the installer is started. It is displayed on the right side of the installer.

You change the default image by clicking the **Select custom logo** link in **Additional Installer Options**. When the file explorer opens, locate and select a new image.

Note You can drag and drop a custom image onto the default logo.

Editing the Installation Notes

Installation notes are displayed once the installer has successfully installed the packaged files on the target system. They can provide useful information concerning any additional set up that is required to use the installed binaries or simply provide instructions for how to run the application.

The field for editing the installation notes is in **Additional Installer Options**.

Manage the Required Files Compiled into a Project

In this section...
“Using the Compiler Apps” on page 5-6 “Using mcc” on page 5-7

Dependency Analysis

The compiler uses a dependency analysis function to automatically determine what additional MATLAB files are required for the application to compile and run. These files are automatically compiled into the generated binary. The compiler does not generate any wrapper code allowing direct access to the functions defined by the required files.

Using the Compiler Apps

If you are using one of the compiler apps, the required files discovered by the dependency analysis function are listed in the **Files required by your application to run** field.

To add files:

- 1 Click the plus button in the field.
- 2 Select the desired file from the file explorer.
- 3 Click **OK**.

To remove files:

- 1 Select the desired file.
- 2 Press the **Delete** key.

Caution Removing files from the list of required files may cause your application to not compile or to not run properly when deployed.

Using `mcc`

If you are using `mcc` to compile your MATLAB code, the compiler does not display a list of required files before running. Instead, it compiles all of the required files that are discovered by the dependency analysis function and adds them to the generated binary file.


You can add files to the list by passing one, or more, `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all of the files in `foo`, and its subfolders, to the list of required files.

Specify Additional Files to Be Installed with the Application

The compiler apps packages additional files to be installed along with the ones it generates. By default the installer includes a readme file with instructions on installing the MATLAB Compiler Runtime (MCR) and configuring it.

These files are listed in the **Files installed with your application** section of the app.

to add files to the list:

- 1 Click the plus () button in the field.
- 2 Select the desired file from the file explorer.
- 3 Click **OK** to close the file explorer.

To remove files from the list:

- 1 Select the desired file.
- 2 Press the **Delete** key.

Caution Removing the binary targets from the list results in an installer that does not install the intended functionality.

When installed on a target computer, the files listed in the **Files installed with your application** are placed in the application folder.

Programming

To access a Java package built and packaged by the MATLAB Builder JA product, you must first install them and the required MATLAB support files so you can use them on a particular machine. See example for more information.

- “About the MATLAB® Builder™ JA API” on page 6-3
- “Importing Classes ” on page 6-8
- “Creating an Instance of the Class” on page 6-9
- “Passing Arguments to and from Java” on page 6-13
- “Passing Java Objects by Reference” on page 6-27
- “Handling Errors” on page 6-33
- “Managing Native Resources” on page 6-40
- “Improving Data Access Using the MCR User Data Interface and MATLAB® Builder™ JA” on page 6-44
- “Dynamically Specifying Run-Time Options to the MCR” on page 6-50
- “Sharing an MCR Instance in COM or Java Applications” on page 6-53
- “Handling Data Conversion Between Java and MATLAB” on page 6-55
- “Setting Java Properties” on page 6-57
- “Blocking Execution of a Console Application that Creates Figures” on page 6-59
- “Ensuring Multi-Platform Portability” on page 6-62
- “CTF Archive Embedding and Extraction” on page 6-64
- “Learning About Java Classes and Methods by Exploring the Javadoc” on page 6-69

Note For examples of these tasks, see the sample Java applications in this documentation. See “Next Steps” on page 2-26 for a list of links to the examples.

For information about deploying your application after you complete these tasks, see “How Does Java Package Deployment Work?” on page 2-4.

About the MATLAB Builder JA API

In this section...

“What Are MATLAB Generated Java Packages and When Should You Create Them?” on page 6-3

“Understanding the MATLAB® Builder™ JA API Data Conversion Classes” on page 6-4

“Automatic Conversion to MATLAB Types” on page 6-5

“Understanding Function Signatures Generated by the MATLAB® Builder™ JA Product” on page 6-6

“Adding Fields to Data Structures and Data Structure Arrays” on page 6-7

“Returning Data from MATLAB to Java” on page 6-7

What Are MATLAB Generated Java Packages and When Should You Create Them?

MATLAB generated Java packages include one or more Java classes that wrap your MATLAB functions. The classes provide methods that allow you to call the functions as you would any other Java method. In addition, the generated classes provide all of the functionality required to manage the MCR required to run the MATLAB functions.

The builder encrypts your MATLAB functions and generates a Java wrapper around them so that they behave just like any other Java class. You can deploy generated packages to enterprise or Web environments, sharing them with anyone running a Web browser and having the MATLAB Component Runtime (MCR) installed.

For development on Linux platforms, Java packages and applications provide portable and scalable solutions for applications in large-scale enterprise or Web environments.

Understanding the MATLAB Builder JA 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 Developer's 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 generated methods.

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 12-4 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 To work directly with cell arrays and data structures in native Java, see “Use Native Java with Cell Arrays and Struct Arrays” on page 10-9 for information and comprehensive examples.

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 6-5 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 MATLAB Code

When the MATLAB Builder JA product encapsulates your MATLAB 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 12-8 for details.

Adding Fields to Data Structures and Data Structure Arrays

When adding fields to data structures and data structure arrays, do so using standard programming techniques. Do not use the `set` command as a shortcut.

For examples of how to correctly add fields to data structures and data structure arrays, see the programming examples in this documentation. For a table of links, see “Next Steps” on page 2-26.

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.

Note To work directly with cell arrays and data structures in native Java, see “Use Native Java with Cell Arrays and Struct Arrays” on page 10-9 for information and comprehensive examples.

Importing Classes

To use a package 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 classes created by the builder, for example:

```
import packagename.classname;
```

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 embedded `.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 6-62 for more details.

Creating an Instance of the Class

In this section...

“What Is an Instance?” on page 6-9

“Instantiate a Java Class” on page 6-9

What Is an Instance?

With a MATLAB Java class, it is necessary first to create an instance of the class, since the methods are non-static.

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

```
MyClass instance = new MyClass();
```

Instantiate 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
 */
public class useMyClass
{
    /** Constructs a new useMyClass */
    public useMyClass()
```

```
{
    super();
}

/* Returns an array containing the primes between 0 and n */
public double[] getPrimes(int n) throws MWException
{
    myClass Class = null;
    Object[] y = null;

    try
    {
        Class = new myClass ();
        y = Class.myPrimes(1, new Double((double)n));
        /* The above signature returns outputs in an
           object array. You must know the output
           type to know what type to cast. */
        return (double[])((MWArray)y[0]).getData();
    }

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

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

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

The following statement instantiates the class `myclass`:

```
Class = new myClass();
```

The following statement calls the class method `myPrimes`:

```
y = Class.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 2001-2010 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” on page 6-13

“Manual Conversion of Data Types” on page 6-13

“Automatic Conversion to a MATLAB Type” on page 6-14

“Specifying Optional Arguments” on page 6-16

“Handling Return Values” on page 6-21

Format

When you invoke a method on a generated class, 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.

Using `MWNumericArray`

The Magic Square example (“Integrating a Generated Java Package into a Java Application” on page 2-15) 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 MATLAB function without further conversion:

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

    theMagic = new Class1();

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

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 MATLAB 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 12-4.

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.

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 MATLAB function. This is the default conversion rule for `java.lang.Double`.

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 MATLAB function.

Calling MArray Methods

The conversion rules apply not only when calling your own methods, but also when calling constructors and factory methods belonging to the MArray 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 MArray 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 MATLAB functions that have varargin or varargout arguments. Consider the following MATLAB 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 2001-2010 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.

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 nargsout) throws MWException
{
    (implementation omitted)
}

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

```

```
                throws MWEException
{
    (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 MATLAB 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 MWEException
{
    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.

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 = 0; i < vals.length; i++)
            x[i] = new MWNumericArray(vals[i], MWClassID.DOUBLE);

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

Passing a Variable Number of Outputs

When present, `varargout` arguments are handled in the same way that `varargin` arguments are handled. Consider the following MATLAB 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 2001-2010 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 *i*th 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)
}
```

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 *i*th 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 MATLAB 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 `toTypeArray` methods.

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.

```
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.

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)
```

```

    {
        long[][] x = (long[][])((MArray)y[0]).toArray();

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

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

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

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

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

```

Using toType Array Methods

The next example demonstrates how you can coerce or force data to a specified numeric type by invoking any of the *toType*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 MATLAB 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 6-27

“Wrapping and Passing Java Objects to MATLAB Functions with MWJavaObjectRef” on page 6-27

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.

You can find the Javadoc at *matlabroot/help/javabuilder/MWArrayAPI* in your product installation.

Wrapping and Passing Java Objects to MATLAB Functions with MWJavaObjectRef

You can create a MATLAB 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 generated package, as well as perform other object marshaling specific to the MATLAB Compiler product. The examples in this section present some common use cases.

Passing a Java Object into a MATLAB Builder JA Method

To pass an object into a MATLAB Builder JA method:

- 1 Use MWJavaObjectRef to wrap your object.
- 2 Pass your object to a MATLAB 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 MATLAB 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 MATLAB function')
h
className = class(h)
whos('h')
methods(h)

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

Cloning an Object

You can also use `MWJavaObjectRef` to clone an object. Continuing with the example in “Passing a Java Object into a MATLAB® Builder™ JA Method” on page 6-27, 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 2001-2010 The MathWorks, Inc.
% $Revision: 1.1.6.60.2.2 $ $Date: 2013/07/18 20:03:31 $

% 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);
```

Passing a Date into a Method and Getting a Date from a Method

In addition to passing in created objects, as in “Passing a Java Object into a MATLAB® Builder™ JA Method” on page 6-27, 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 MATLAB 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 a MATLAB 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 2001-2010 The MathWorks, Inc.
% $Revision: 1.1.6.60.2.2 $ $Date: 2013/07/18 20:03:31 $

% 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 method (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`:

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}

```

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 method, see “Optimization” on page 7-36.

Handling Errors

In this section...

“Error Overview” on page 6-33

“Handling Checked Exceptions” on page 6-33

“Handling Unchecked Exceptions” on page 6-36

“Alternatives to Using of System.exit” on page 6-39

Error Overview

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

The Java interface to each MATLAB function declares itself as throwing an `MWException` using the `throws` clause. For example, the `myprimes` MATLAB 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.

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 6-40.

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[])((MArray)y[0]).getData();
}

finally
{
    MArray.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 MArray API classes all throw unchecked exceptions.

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

- `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.

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[])((MArray)y[0]).getData();
    }

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

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

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[])((MArray)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
{
    MArray.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.

Alternatives to Using of System.exit

Any Java application that uses a class generated using MATLAB Builder JA should avoid any direct or indirect calls to `System.exit`.

Any direct or indirect call to `System.exit` will result in the JVM shutting down in an abnormal fashion. This may result in system deadlocks.

Using `System.exit` also causes the java process to exit unpredictably.

Java programs using Swing components are most likely to invoke `System.exit`. Here are a few ways to avoid it:

- Use public interface `WindowConstants.DISPOSE_ON_CLOSE` method as an alternative to `WindowConstants.EXIT_ON_CLOSE` as input to the `JFrame` class `setDefaultCloseOperation` method.
- If you want to provide an **Exit** button in your GUI that terminates your application, instead of calling `System.exit` in the `ActionListener` for the button, call the `dispose` method on `JFrame`.

Managing Native Resources

In this section...

“What Are Native Resources?” on page 6-40

“Using Garbage Collection Provided by the JVM” on page 6-40

“Using the dispose Method” on page 6-41

“Overriding the Object.Finalize Method” on page 6-43

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.

Currently, the only way to increase heap size for a compiled application is write a `java.opts` file.

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.

Calling `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`.

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.

Calling `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 MATLAB 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 information to a client running an application created with the Parallel Computing Toolbox. Profile information may be supplied (and change) on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles.
- You want to initialize the MCR with constant values that can be accessed by all your MATLAB 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 software supports per-MCR instance state access through an object-oriented API. Unlike MATLAB Compiler software, access to a 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. Alternatively, you use a helper function to call these methods as demonstrated in “Supply Run-Time Profile Information for Parallel Computing Toolbox Applications” on page 6-45.

For more information, see “Using the MCR User Data Interface” in the MATLAB Compiler User’s Guide.

Supply Run-Time Profile Information for Parallel Computing Toolbox Applications

Following is a complete example of how you can use the MCR User Data Interface as a mechanism to specify a profile for Parallel Computing Toolbox applications.

Note Standalone executables and shared libraries generated from MATLAB Compiler for parallel applications can now launch up to twelve local workers without MATLAB Distributed Computing Server™.

Step 1: Write Your Parallel Computing Toolbox Code

1 Compile `sample_pct.m` in MATLAB.

This example code uses the cluster defined in the default profile.

The output assumes that the default profile is local.

```
function speedup = sample_pct (n)
warning off all;
tic
if(ischar(n))
    n=str2double(n);
end
for ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
end
time1 =toc;
matlabpool('open');
tic
parfor ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
end
time2 =toc;
disp(['Normal loop times: ' num2str(time1) ...
    ',parallel loop time: ' num2str(time2) ]);
disp(['parallel speedup: ' num2str(1/(time2/time1)) ...
```

```
        ' times faster than normal']);  
matlabpool('close');  
disp('done');  
speedup = (time1/time2);
```

- 2 Run the code as follows after changing the default profile to `local`, if needed.

```
a = sample_pct(200)
```

- 3 Verify that you get the following results;

```
Starting matlabpool using the 'local'  
profile ... connected to 4 labs.  
Normal loop times: 1.4625, parallel loop time: 0.82891  
parallel speedup: 1.7643 times faster than normal  
Sending a stop signal to all the labs ... stopped.  
done  
a =  
    1.7643
```

Step 2: Set the Parallel Computing Toolbox Profile

In order to compile MATLAB code to a Java package and utilize the Parallel Computing Toolbox, the `mcruserdata` must be set directly from MATLAB. There is no Java API available to access the `MCRUserdata` as there is for C and C++ applications built with MATLAB Compiler.

To set the `mcruserdata` from MATLAB, create an `init` function in your Java class. This is a separate MATLAB function that uses `setmcruserdata` to set the Parallel Computing Toolbox profile once. You then call your other functions to utilize the Parallel Computing Toolbox functions.

Create the following `init` function:

```
function init_sample_pct  
% Set the Parallel Profile:  
if(isdeployed)  
    [profile, profpath] = uigetfile('*.settings');  
    % let the USER select file
```



```
        setmcuserdata('ParallelProfile', fullfile(profpath, profile));  
end
```

Tip If you need to change your profile in the application, use the `parallel.importProfile` and `parallel.defaultClusterProfile` methods. See the Parallel Computing Toolbox™ documentation for more information.

Step 3: Compile Your Function with the Deployment Tool or the Command Line

You can compile your function from the command line by entering the following:

```
mcc -S -W 'java:parallelComponent,PctClass' init_sample_pct.m sample_pct.m
```

Alternately, you can use the Deployment Tool as follows:

- 1 Follow the steps in to compile your application. When the compilation finishes, a new folder (with the same name as the project) is created. This folder contains two subfolders: `distrib` and `src`.

Project Name	parallelComponent
Class Name	PctClass
File to Compile	pct_sample.m and init_pct_sample.m

Note If you are using the GPU feature of Parallel Computing Toolbox, you need to manually add the PTX and CU files.

If you are using a Deployment Tool project, click **Add files/directories** on the **Build** tab.

If you are using the `mcc` command, use the `-a` option.

2 To deploy the compiled application, copy the `distrib` folder, which contains the following, to your end users. The packaging function of `deploytool` offers a convenient way to do this.

- `parallelComponent.jar`
- `javabuilder.jar`
- MCR installer
- Cluster profile

Note The end-user's target machine must have access to the cluster.

Step 4: Write the Java Driver Application

Write the following Java driver application to use the generated package, as follows, using a Java-compatible IDE such as Eclipse™:

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

public class JavaParallelClass
{
    public static void main(String[] args)
    {
        MWArray A = null;
        PctClass C = null;
        Object[] B = null;
        try
        {
            C = new PctClass();
            /* Set up the MCR with Parallel Data */
            C.init_sample_pct();
            A = new MWNumericArray(200);
            B = C.sample_pct(1, A);
            System.out.println(" The Speed Up was:" + B[0]);
        }
        catch (Exception e)
        {
```

```
        System.out.println("The error is " + e.toString());
    }
    finally
    {
        MWArray.disposeArray(A);
        C.dispose();
    }
}
}
```

The output is as follows:

```
(UIGETFILE brings up the window to select the MAT file)
Starting matlabpool using the 'profile' cluster profile
        ... connected to 4 labs.
Normal loop times: 2.6641, parallel loop time: 1.2568
parallel speedup:  2.1198 times faster than normal
Sending a stop signal to all the labs ... stopped.
Did not find any pre-existing parallel jobs created
        by matlabpool.
done
The Speed Up was:2.1198
```

Compiling and Running the Application Without Using an IDE. If you are not using an IDE, compile the application using command-line Java, as follows:

Note Enter these commands on a single line, using the semi-colon as a delimiter.

```
javac -classpath .;C:\pct_compile\javaApp\parallelComponent.jar;
        matlabroot\toolbox\javabuilder\jar\javabuilder.jar
JavaParallelClass.java
```

Run the application from the command-line, as follows:

```
java -classpath .;C:\pct_compile\javaApp\parallelComponent.jar;
        matlabroot\toolbox\javabuilder\jar\javabuilder.jar
JavaParallelClass
```

Dynamically Specifying Run-Time Options to the MCR

In this section...

“What Run-Time Options Can You Specify?” on page 6-50

“Setting and Retrieving MCR Option Values Using MWApplication” on page 6-50

What Run-Time Options Can You Specify?

As of R2009a, you can pass MCR run-time options `-nojvm`, `-nodisplay`, and `-logfile` to MATLAB Builder JA from the client application using two classes in `javabuilder.jar`:

- `MWApplication`
- `MWMCROption`

Setting and Retrieving MCR Option Values Using MWApplication

The `MWApplication` class provides several static methods to set MCR option values and also to retrieve them. The following table lists static methods supported by this class.

MWApplication Static Methods	Purpose
<code>MWApplication.initialize(MWMCROption... options);</code>	Passes MCR options (see “Specifying Run-Time Options Using MWMCROption” on page 6-51)
<code>MWApplication.isMCRInitialized();</code>	Returns <code>true</code> if MCR is initialized; otherwise returns <code>false</code>
<code>MWApplication.isMCRJVMEEnabled();</code>	Returns <code>true</code> if MCR is launched with JVM; otherwise returns <code>false</code>

MWApplication Static Methods	Purpose
<code>MWApplication.isMCRNoDisplaySet();</code>	<p>Returns true if <code>MWMCROption.NODISPLAY</code> is used in <code>MWApplication.initialize</code></p> <hr/> <p>Note false is always returned on Windows systems since the <code>-nodisplay</code> option is not supported on Windows systems.</p>
<code>MWApplication.getMCRLogfileName();</code>	Retrieves the name of the log file

Specifying Run-Time Options Using `MWMCROption`

`MWApplication.initialize` takes zero or more `MWMCROptions`.

Calling `MWApplication.initialize()` without any inputs launches the MCR with the following default values.

You must call `MWApplication.initialize()` before performing any other processing.

These options are all write-once, read-only properties.

MCR Run-Time Option	Default Values
<code>-nojvm</code>	false
<code>-logfile</code>	null
<code>-nodisplay</code>	false

Note If there are no MCR options being passed, you do not need to use `MWApplication.initialize` since initializing a generated class initializes the MCR with default options.

Use the following static members of `MWMCROption` to represent the MCR options you want to modify.

MWMCROption Static Members	Purpose
<code>MWMCROption.NOJVM</code>	Launches the MCR without a Java Virtual Machine (JVM). When this option is used, the JVM launched by the client application is unaffected. The value of this option determines whether or not the MCR should attach itself to the JVM launched by the client application.
<code>MWMCROption.NODISPLAY</code>	Launches the MCR without display functionality.
<code>MWMCROption.logFile</code> (<code>"logfile.dat"</code>)	Allows you to specify a log file name (must be passed with a log file name).

Passing and Retrieving MCR Option Values from a Java Application.

Following is an example of how MCR option values are passed and retrieved from a client-side Java application:

```
MWApplication.initialize(MWMCROption.NOJVM,
    MWMCROption.logFile("logfile.dat"),MWMCROption.NODISPLAY);
System.out.println(MWApplication.getMCRLogfileName());
System.out.println(MWApplication.isMCRInitialized());
System.out.println(MWApplication.isMCRJVMEnabled());
System.out.println(MWApplication.isMCRNoDisplaySet()); //UNIX

//Following is the initialization
// of MATLAB Builder JA
// class
myclass cls = new myclass();
cls.hello();
```

Sharing an MCR Instance in COM or Java Applications

In this section...

“What Is a Singleton MCR?” on page 6-53

“Advantages and Disadvantages of Using a Singleton” on page 6-53

“Which Products Support Singleton MCR and How Do I Create a Singleton?” on page 6-54

What Is a Singleton MCR?

You create an instance of the MCR that can be shared (and reused) among all subsequent class instances within a component. This is commonly called a shared MCR instance or a *Singleton MCR*.

Advantages and Disadvantages of Using a Singleton

In most cases, a singleton MCR will provide many more advantages than disadvantages. Following are examples of when you might and might not create a shared MCR instance.

When You Should Use a Singleton

If you have multiple users running from a specific instance of MATLAB, using a singleton will most likely:

- Utilize system memory more efficiently
- Decrease MCR start-up or initialization time
- Promote reuse of your application code base

When You Might Avoid Using a Singleton

Situations where using a singleton may not benefit you include:

- Running applications with a large number of global variables. This can promote crosstalk which can eventually impact performance.
- Your installation runs many different versions of MATLAB, for testing purposes.

- Your installation has a relative
 - On the library compiler app, select **Exclusive MCR** under **Additional Runtime Settings**.
- y small number of users and is not overly concerned with performance.

Which Products Support Singleton MCR and How Do I Create a Singleton?

Singleton MCR is only supported by the following products on these specific targets:

Product	Target supported by Singleton MCR	Create a Singleton MCR by...
MATLAB Builder EX	COM component	Default behavior for target is Singleton MCR. You do not need to perform other steps.
MATLAB Builder NE	.NET assembly	Default behavior for target is Singleton MCR. You do not need to perform other steps.
MATLAB Builder NE	COM component	<ul style="list-style-type: none"> • Using the shared library compiler app, click Settings and add -S to the Additional flags to pass to mcc field. • Using mcc pass the -S flag.
MATLAB Builder JA	Java packages	

Handling Data Conversion Between Java and MATLAB

In this section...

“Overview” on page 6-55

“Calling MArray Methods” on page 6-55

“Creating Buffered Images from a MATLAB Array” on page 6-56

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 MArray classes are converted by the builder to the correct MArray 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 MArray Methods

The conversion rules apply not only when calling your own methods, but also when calling constructors and factory methods belonging to the MArray 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. The Javadoc is located at `matlabroot/help/javabuilder/MWArrayAPI`.

Setting Java Properties

In this section...

“How to Set Java System Properties” on page 6-57

“Ensure a Consistent GUI Appearance” on page 6-57

How to Set Java System Properties

Set Java system properties in one of two ways:

- *In the Java statement.* Use the syntax: `java -Dpropertyname=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 classes:

```
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.

Ensure 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`.

Setting DisableSetLookAndFeel

The following are examples of how to set `mathworks.DisableSetLookAndFeel` using the techniques in “How to Set Java System Properties” on page 6-57:

- 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

In this section...

“waitForFigures Method” on page 6-59

“Block Figure Window Display in a Console Application” on page 6-60

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 MATLAB code are displayed. Typically you use `waitForFigures` when:

- There are one or more figures open that were created by a Java class 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. Call this method *only* from console-based programs.

Block Figure Window Display in a Console Application

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

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

- 2 In this folder, create the following MATLAB file:

```
drawplot.m

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

- 3 Use the MATLAB Builder JA product to create a Java package 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:

```
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 Package in a Java Application” on page 2-16.

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 MATLAB 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 package 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_matlab_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 folder.

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 package 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 MATLAB function as follows:

```
function callyprime
```



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

Ensure the `yprime.mexa64` file is in the same folder 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.

Caution Some toolbox functionality will not be deployable when compiled into a Java package and run on a platform other than the one compiled on. This is because some toolbox code includes data that may be platform specific. If this is the case, you can only deploy the application to like platforms. For example, the Image Processing Toolbox function `IMHIST` will fail if deployed cross-platform with an `undefined function` error.

JAR files produced by MATLAB Builder JA are tested and qualified to run on platforms supported by MATLAB. See the Platform Roadmap for MATLAB for more information.

CTF Archive Embedding and Extraction

In this section...

“Overview” on page 6-64

“Using MWComponentOptions Class to Indicate Extraction Options” on page 6-64

“Using Environment Variables to Indicate Extraction Options” on page 6-66

“For More Information” on page 6-68

Overview

CTF data is extracted from the JAR file with no separate CTF or *componentname* folder needed on the target machine. This behavior is helpful when storage space on a file system is limited.

If you don't want CTF data extracted by default, use either the `MWComponentOptions` class, or use environment variables, to specify how MATLAB Builder JA handles CTF data extraction and utilization.

Using MWComponentOptions Class to Indicate Extraction Options

You can find the Javadoc for `MWComponentOptions` at <http://www.mathworks.com/help/javabuilder/MWArrayAPI/com/mathworks/toolbox/javab>

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 folder to be scanned when instantiating the component: if a file with a `.ctf` suffix is found in the folder 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 R2007a and forward (see).
 - `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 folder. This option most closely matches the behavior of R2007a and previous releases.

Note CTF archives are extracted by default to `temp\user_name\mcrCache.nn`.

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()`

Enabling MCR Component Cache, Utilizing CTF Content Already on Your System. If you want to enable the MCR Component Cache for a generated Java class utilizing CTF content already resident in your file system, instantiate `MWComponentOptions` using the following statements:

```
MWComponentOptions options = new MWComponentOptions();

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

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

Using Environment Variables to Indicate Extraction Options

Use the following environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the CTF archive to be extracted, this variable overrides the default per-user component cache location.	Does not apply
MCR_CACHE_VERBOSE	When set to any value, this variable prints logging details about the component cache for diagnostic reasons. This can be very helpful if problems are encountered during CTF archive extraction.	Logging details are turned off by default (for example, when this variable has no value).
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file <code>.max_size</code> , which resides in the file designated by running the <code>mrcachedir</code> command, with the desired cache size limit.

You can override this automatic embedding and extraction behavior by compiling with the `-C` option described in .

Overriding Default Behavior

To extract the CTF archive, compile using the option `mcc -C`.

You can also implement this override by checking the appropriate **Option** in the Deployment Tool.

You might want to use this option to troubleshoot problems with the CTF archive, for example, as the log and diagnostic messages are much more visible.

For More Information

For more information about the CTF Archive, see “Component Technology File (CTF Archive)”.

Learning About Java Classes and Methods by Exploring the Javadoc

The documentation generated by Oracles Javadoc can be a powerful resource when using the MATLAB Builder JA product. The Javadoc can be browsed from *matlabroot/help/javabuilder/MWArrayAPI* in your product installation and by entering the name of the class or method you want to learn more about in the search field on the Index page.

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

Sample Java Applications

- “Plot” on page 7-2
- “Spectral Analysis” on page 7-9
- “Matrix Math” on page 7-16
- “Phone Book” on page 7-28
- “Optimization” on page 7-36
- “Web Application” on page 7-47

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

In this section...
“Purpose” on page 7-2
“Procedure” on page 7-2

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.m`) to a method of a Java class (`plotter`) and wrap the class in a Java package (`plotdemo`).
- Access the MATLAB function 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.m` 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 folder that ships with MATLAB to your work folder:

`matlabroot\toolbox\javabuilder\Examples\PlotExample`

- b At the MATLAB command prompt, `cd` to the new `PlotExample` subfolder in your work folder.

- 2 If you have not already done so, set the environment variables that are required on a development machine. See .
- 3 Write the `drawplot.m` function as you would any MATLAB function.

The following code defines the `drawplot.m` function:

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

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

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

```
deploytool
```

- 5 You create a Java application by using the Deployment Tool GUI to build a Java class that wraps around your MATLAB code.

To compile or build the Java application using the Deployment Tool, use the following information as you work through this example in :

Project Name	plotdemo
Class Name	plotter
File to compile	drawplot.m

- 6 Write source code for an application that accesses the MATLAB function.

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 2001-2011 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,
                MWCClassID.DOUBLE, MWComplexity.REAL);
            y = MWNumericArray.newInstance(dims,
                MWCClassID.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.

7 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 folder. Ensure `createplot.java` is in your work folder

- 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
```

8 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
```

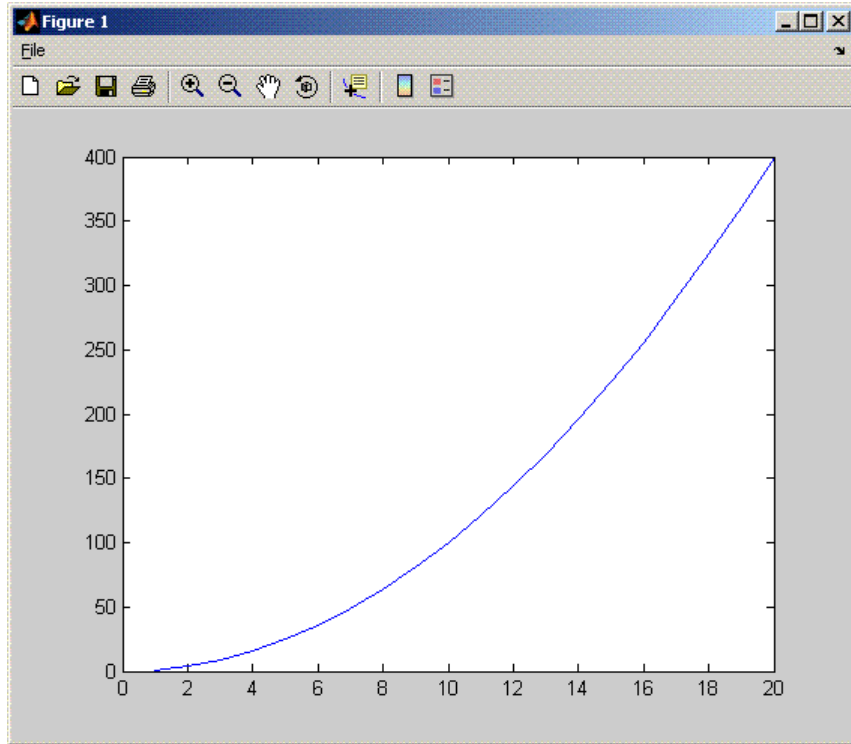
Note You should be using the same version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Caution MathWorks only supports the Oracle JDK and JRE. A certain measure of cross-version compatibility resides in the Oracle software and it *may* be possible to run MCR-based components with non-Oracle JDKs under some circumstances—however, compatibility is not guaranteed.

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “MATLAB® Builder™ JA Limitations” on page 12-3 for more specific information.

The `createplot` program should display the output.



Spectral Analysis

In this section...
“Purpose” on page 7-9
“Procedure” on page 7-10

Purpose

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

- How to use the MATLAB Builder JA product to create a package (spectralanalysis) containing a class that has a private method that is automatically encapsulated
- How to access the MATLAB functions in a Java application (powerspect.java), including use of the MArray class hierarchy to represent data

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

- How to build and run the application

The `spectralanalysis` package 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 is found in `matlabroot\toolbox\javabuilder\Examples\SpectraExample\SpectraDemoComp.`

Procedure

- 1 If you have not already done so, copy the files for this example as follows:
 - a Copy the following folder that ships with MATLAB to your work folder:

```
matlabroot\toolbox\javabuilder\Examples\SpectraExample
```

- b At the MATLAB command prompt, cd to the new SpectraExample subfolder in your work folder.

- 2 If you have not already done so, set the environment variables that are required on a development machine. See .
- 3 Write the MATLAB code that you want to access.

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

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

```
deploytool
```

- 5 You create a Java application by using the Deployment Tool GUI to build a Java class that wraps around your MATLAB code.

To compile or build the Java application using the Deployment Tool, use the following information as you work through this example in :

Project Name	spectralanalysis
Class Name	fourier
File to compile	plotfft.m

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 package, you do not need to add the `computefft` function, although doing so does no harm.

6 Write source code for an application that accesses the MATLAB functions.

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 2001-2011 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  $\sin(2\pi \cdot 15 \cdot t)$  +
             *  $\sin(2\pi \cdot 40 \cdot t)$  plus a random signal.
             * Duration = 10
            */
        }
    }
}
```

```
        *   Sampling interval = 0.01
        */
int[] dims = {1, nSamples};
data = MWNumericArray.newInstance(dims, MWClassID.DOUBLE,
                                MWCComplexity.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 `computefft` and plots the data
- Uses a try-catch block to handle exceptions
- Frees native resources using `MWArray` methods

- 7 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` folder. `cd` to your work folder. Ensure `powerspect.java` is in your work folder

- 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 folder on your system. Type `matlabroot` to see this folder name.

8 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
```

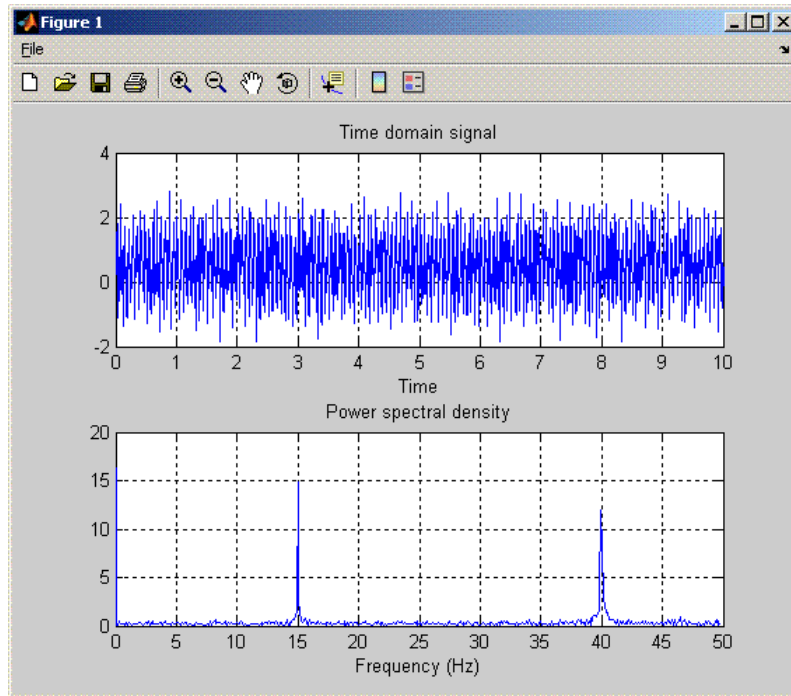
Note You should be using the same version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Caution MathWorks only supports the Oracle JDK and JRE. A certain measure of cross-version compatibility resides in the Oracle software and it *may* be possible to run MCR-based components with non-Oracle JDKs under some circumstances—however, compatibility is not guaranteed.

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “MATLAB® Builder™ JA Limitations” on page 12-3 for more specific information.

The powerspect program should display the output:



Matrix Math

In this section...

“Purpose” on page 7-16

“MATLAB Functions to Be Encapsulated” on page 7-17

“Understanding the getfactor Program” on page 7-18

“Procedure” on page 7-18

Purpose

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

- How to assign more than one MATLAB function to a generated class.
- How to manually handle native memory management.
- How to access the MATLAB functions 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 package 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:

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

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 2001-2011 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 2001-2011 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 2001-2011 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 folder that ships with MATLAB to your work folder:

```
matlabroot\toolbox\javabuilder\Examples\MatrixMathExample
```
 - b At the MATLAB command prompt, `cd` to the new `MatrixMathExample` subfolder in your work folder.
- 2 If you have not already done so, set the environment variables that are required on a development machine. See .
- 3 Write the MATLAB functions as you would any MATLAB function.

The code for `cholesky.m`, `ludcomp.m`, and `qrdecomp.m` functions is already in your work folder in `MatrixMathExample\MatrixMathDemoComp\`.

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

```
deploytool
```

- 5 You create a Java application by using the Deployment Tool GUI to build a Java class that wraps around your MATLAB code.

To compile or build the Java application using the Deployment Tool, use the following information as you work through this example in :

Project Name	factormatrix
Class Name	factor
Files to compile	cholesky.m ludcomp.m qrdecomp.m

- 6 Write source code for an application that accesses the MATLAB functions.

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 2001-2011 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[0], 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[0]);
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);
...
```

- 7** Copy `getfactor.java` into the `factormatrix` folder.
- 8** 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 `factormatrix` folder in your work folder.

- 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
```

- 9** 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
```

Note You should be using the same version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Caution MathWorks only supports the Oracle JDK and JRE. A certain measure of cross-version compatibility resides in the Oracle software and it *may* be possible to run MCR-based components with non-Oracle JDKs under some circumstances—however, compatibility is not guaranteed.

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “MATLAB® Builder™ JA Limitations” on page 12-3 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

Phone Book

In this section...
“Purpose” on page 7-28
“Procedure” on page 7-28

Purpose

An example of how to process an `MWStructArray` as output from a generated class might be:

```
Object[] tmp = myComponent.myFunction(1, myArray);  
MWStructArray myStruct = (MWStructArray) tmp[0];
```

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 folder that ships with MATLAB to your work folder:

```
matlabroot\toolbox\javabuilder\Examples\PhoneExample
```
 - b At the MATLAB command prompt, `cd` to the new `PhoneExample` subfolder in your work folder.
- 2 If you have not already done so, set the environment variables that are required on a development machine. See .
- 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 2006-2011 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 folder in `makephone.m`.

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

```
deploytool
```

- 5** You create a Java application by using the Deployment Tool GUI to build a Java class that wraps around your MATLAB code.

To compile or build the Java application using the Deployment Tool, use the following information as you work through this example in :

Project Name	phonebookdemo
Class Name	phonebook
File to compile	makephone.m

- 6** Write source code for an application that accesses the MATLAB functions.

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 2001-2011 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 a MATLAB 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.

- 7** 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 folder. Ensure `getphone.java` is in your work folder

- 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
```

- 8** 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 You should be using the same version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Caution MathWorks only supports the Oracle JDK and JRE. A certain measure of cross-version compatibility resides in the Oracle software and it *may* be possible to run MCR-based components with non-Oracle JDKs under some circumstances—however, compatibility is not guaranteed.

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “MATLAB® Builder™ JA Limitations” on page 12-3 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
```

Optimization

In this section...
“Purpose” on page 7-36
“OptimDemo Package” on page 7-36
“Procedure” on page 7-37

Purpose

This example shows how to:

- Use the MATLAB Builder JA product to create a package (OptimDemo) that applies MATLAB optimization routines to objective functions implemented as Java objects.
- Access the MATLAB functions 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 generated Java methods.

Note For complete reference information about the MWArray class hierarchy, see the `com.mathworks.toolbox.javabuilder` Javadoc package in `matlabroot/help/toolbox/javabuilder/MWArrayAPI`.

- Build and run the application.

OptimDemo Package

The OptimDemo package finds a local minimum of an objective function and returns the minimal location and value. The package 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 folder that ships with MATLAB to your work folder:
`matlabroot\toolbox\javabuilder\Examples\ObjectRefExample`
 - b** At the MATLAB command prompt, `cd` to the new `ObjectRefExample` subfolder in your work folder.
- 2** If you have not already done so, set the environment variables that are required on a development machine. See .
- 3** Write the MATLAB code that you want to access. This example uses `doOptim.m` and `displayObj.m`, which are already in your work folder in `ObjectRefExample\ObjectRefDemoComp`.

For reference, the code of `doOptim.m` is displayed here:

```
function [x,fval] = doOptim(h, x0)
%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 You create a Java application by using the Deployment Tool GUI to build a Java class that wraps around your MATLAB code.

To compile or build the Java application using the Deployment Tool, use the following information as you work through this example in :

Project Name	OptimDemo
Class Name	Optimizer
File to compile	doOptim.m displayObj.m

- 6 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 2001-2011 The MathWorks, Inc.
 * $Revision: 1.1.6.10.2.1 $ $Date: 2013/07/08 15:59:30 $
 */

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.

- 7** Write source code for an application that accesses the MATLAB functions. 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 2001-2011 The MathWorks, Inc.
 * $Revision: 1.1.6.10.2.1 $ $Date: 2013/07/08 15:59:30 $
 */

/* 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 a MATLAB 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 optim**");
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(x0);
    }
}
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

8 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` folder.

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
```

9 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 You should be using the same version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Caution MathWorks only supports the Oracle JDK and JRE. A certain measure of cross-version compatibility resides in the Oracle software and it *may* be possible to run MCR-based components with non-Oracle JDKs under some circumstances—however, compatibility is not guaranteed.

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “MATLAB® Builder™ JA Limitations” on page 12-3 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

Web Application

In this section...

“Overview” on page 7-47

“Prerequisites” on page 7-47

“Downloading the Example Files” on page 7-48

“Build Your Java Package” on page 7-49

“Compiling Your Java Code” on page 7-50

“Generating the Web Archive (WAR) File ” on page 7-50

“Running the Web Deployment Example” on page 7-51

“Using the Web Application” on page 7-51

Overview

This example demonstrates how to display a plot created by a Java servlet calling a class 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 “Specifying Optional Arguments” on page 6-16.

Prerequisites

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 7-47
- “Ensure Your Web Server Is Java Compliant” on page 7-48
- “Install the `javabuilder.jar` Library” on page 7-48

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 a JDK installed on your system. You can download it from Oracle, Inc.

Note You should be using the same version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Caution MathWorks only supports the Oracle JDK and JRE. A certain measure of cross-version compatibility resides in the Oracle software and it *may* be possible to run MCR-based components with non-Oracle JDKs under some circumstances—however, compatibility is not guaranteed.

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 example (“Implement a Custom WebFigure” on page 8-9) 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 folder.

Downloading the Example Files

Download the example 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 Example Files

The example files contain the following three folders:

- `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 example.

Note As an alternative to compiling the example code manually and creating the application WAR (Web Archive) manually, you can run `compileVarArgServletDemo.m` in the `compile` folder. 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 Example” on page 7-51.

Build Your Java Package

Build your Java package by compiling your code into a deployable `.jar` file.

Note For a more detailed explanation of building a Java package, including further details on setting up your Java environment, the `src` and `distrib` folders, and other information, see .

- 1 Start `deploytool` at the MATLAB command line.
- 2 You create a Java application by using the Deployment Tool GUI to build a Java class that wraps around your MATLAB code.

To compile or build the Java application using the Deployment Tool, use the following information as you work through this example in :

Project Name	vararg_java
Class Name	vararg_javaclass
File to compile	varargexample.m

Compiling Your Java Code

Use `javac` to compile the Java source file `VarArgServletClass.java` from example folder `JavaCode\src\VarArg`.

`javac.exe` should be located in the `bin` folder of your JDK installation.

Ensure your `classpath` is set to include:

- `javabuilder.jar` (shipped with the MATLAB Builder JA product)
- `vararg_java.jar` (shipped with the MATLAB Builder JA product)
- `servlet-api.jar` (in the example folder `JavaCode\lib`)

For more details about using `javac`, see the Oracle 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** Add `javabuilder.jar` to the `WEB-INF\lib` directory. For more information, see “Helper Library Locations” in the *MATLAB Application Deployment Web Example Guide*.
- 2** Copy the JAR file created using the MATLAB Builder JA product into the `JavaCode\build\WEB-INF\classes\VarArg` example folder.
- 3** Copy the compiled Java class to the `JavaCode\build\WEB-INF\classes\VarArg` example folder.
- 4** From the folder `JavaCode`, 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 folder.

Caution Placing `javabuilder.jar` in the `WEB-INF/Lib` folder for a single Web application generally works. However, if another application also places `javabuilder.jar` in its `WEB-INF/Lib` locations, problems may occur. The native resources associated with `javabuilder.jar` can be loaded only once in an application. Therefore, `javabuilder.jar` must only be visible to a single class loader.

For more information about the `jar` command, refer to the Oracle Web site.

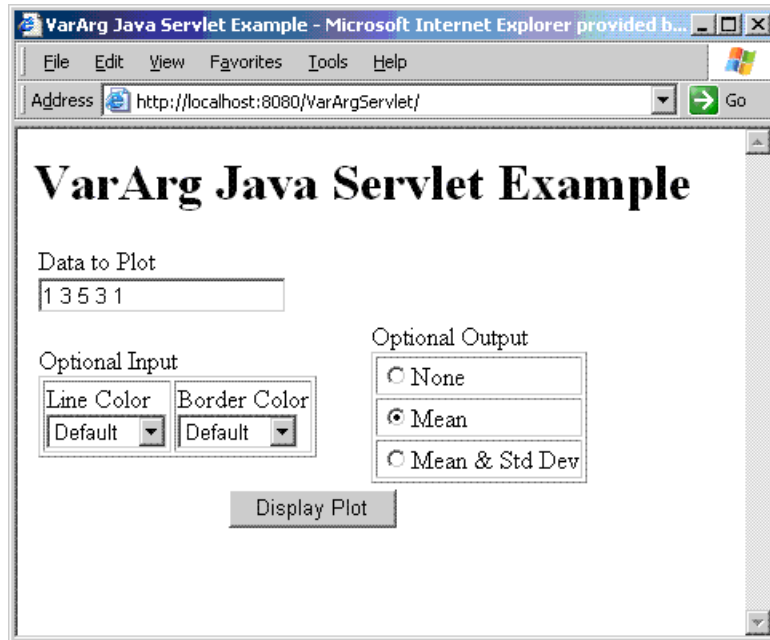
Running the Web Deployment Example

When you're ready to run the application, do the following:

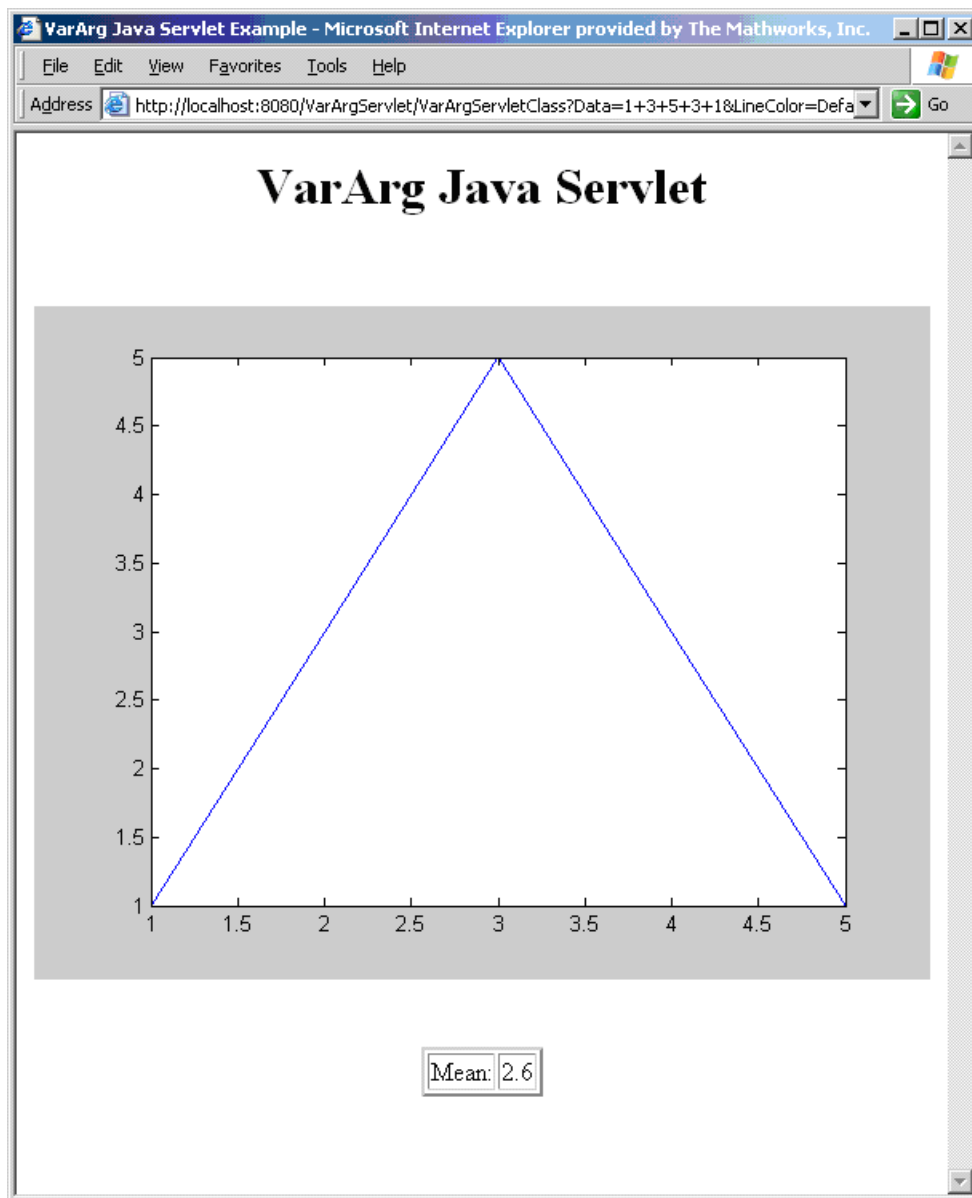
- 1 Install the `VarArgServlet.war` file into your Web server's `webapps` folder.
- 2 Run the application by entering `http://localhost:port_number/VarArgServlet` in the address field of your Web browser, where `port_number` is the port that your Web server is configured to use (usually 8080).

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 MATLAB 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 MATLAB code.
- 4 Click **Display Plot**. Example output is shown below using the **Mean** optional output.



Deploying a Java Package Over the Web

- “About the WebFigures Feature” on page 8-2
- “Preparing to Implement WebFigures for MATLAB® Builder™ JA” on page 8-3
- “Implement a Custom WebFigure” on page 8-9
- “Advanced Configuration of a WebFigure” on page 8-19

About the WebFigures Feature

Using the WebFigures feature in MATLAB Builder JA, you display MATLAB figures on a Web site for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the Web without the need to download MATLAB or other tools that can consume costly resources. End users do not need to have the MATLAB Compiler Runtime (MCR) installed on their systems to use WebFigures.

“Implement a Custom WebFigure” on page 8-9 guides you through implementing the basic features of WebFigures, and lets you customize your configuration depending on your server architecture.

Supported Renderers for WebFigures

The MATLAB Builder JA WebFigures feature uses the same renderer used when the figure was originally created by the MATLAB renderer.

For more information about MATLAB renderers, see the MATLAB documentation.

Note The WebFigures feature does not support the `Painter` renderer due to technical limitations. If this renderer is requested, the `renderer Zbuffer` will be invoked before the data is displayed on the Web page.

Preparing to Implement WebFigures for MATLAB Builder JA

In this section...

- “Your Role in the WebFigure Deployment Process” on page 8-3
- “What You Need to Know to Implement WebFigures” on page 8-5
- “Required Products” on page 8-5
- “Assumptions About the Examples” on page 8-7
- “Set DISPLAY on UNIX Systems” on page 8-8

Your Role in the WebFigure Deployment Process

Depending on your role in your organization, as well as a number of other criteria, you may need to implement either the beginning or the advanced configuration of WebFigures.

The table WebFigures for MATLAB® Builder™ JA Deployment Roles, Responsibilities, and Tasks on page 8-3 describes some of the different roles, or jobs, that MATLAB Builder JA users typically perform and which method of configuration they would most likely use when implementing WebFigures for MATLAB Builder JA.

WebFigures for MATLAB Builder JA Deployment Roles, Responsibilities, and Tasks

Role	Typical Responsibilities	Tasks
MATLAB programmer	<ul style="list-style-type: none"> • Understand end-user business requirements and the mathematical models needed to support them. • Write MATLAB code. • Build an executable component with MATLAB tools (usually with support from a Java developer). 	<ul style="list-style-type: none"> • Write and deploy MATLAB code, such as that in “Assumptions About the Examples” on page 8-7. • Use “Implement a Custom WebFigure” on page 8-9 to easily create a graphic, such as a MATLAB figure,

WebFigures for MATLAB Builder JA Deployment Roles, Responsibilities, and Tasks (Continued)

Role	Typical Responsibilities	Tasks
	<ul style="list-style-type: none"> • Package the component for distribution to end users. 	<p>that the end user can manipulate over the Web.</p>
Application developer	<ul style="list-style-type: none"> • Design and configure the IT environment, architecture, or infrastructure. • Install deployable applications along with the proper version of the MCR. • Create mechanisms for exposing application functionality to the end user. 	<ul style="list-style-type: none"> • Use “Implement a Custom WebFigure” on page 8-9 to easily create a graphic, such as a MATLAB figure, that the end user can manipulate over the Web. • Use “Advanced Configuration of a WebFigure” on page 8-19 to create a flexible, scalable implementation that can meet a number of varied architectural requirements.

What You Need to Know to Implement WebFigures

The following knowledge is assumed when you implement WebFigures for MATLAB Builder JA:

- If you are a MATLAB programmer:
 - Advanced to expert knowledge of MATLAB
- If you are a Java developer:
 - Knowledge of how to create a J2EE Web site on a J2EE-compliant Web server
 - Experience deploying MATLAB applications is helpful

Required Products

Install the following products to implement WebFigures for MATLAB Builder JA, depending on your role.

MATLAB Programmer

- MATLAB
- MATLAB Builder JA
- MATLAB Compiler
- MATLAB Compiler Runtime (see the system requirements at http://www.mathworks.com/support/sysreq/current_release/)

Java Developer

- Java Developer's Kit (JDK) (see the list of supported compilers).
- J2EE compliant Web server, such as Apache Tomcat
- Java Runtime Environment (JRE) (see the system requirements).

Note At this time, only J2EE Web servers that use the Oracle JVM™ support WebFigures.

Assumptions About the Examples

To work with the examples in this chapter:

- Assume the following MATLAB function has been created:

```
function df = getKnot()
    f = figure('Visible','off'); %Create a figure.
                                   %Make sure it isn't visible.
    knot;                            %Put something into figure.
    df = webfigure(f);                %Give figure to function
                                   % and return the result.
    close(f);                          %Close the figure.
end
```

- Assume that the function `getKnot` has been deployed in a Java package (using `example`, for instance) with a namespace of `MyComponent.MyComponentclass`.
- Assume the MATLAB Compiler Runtime (MCR) has been installed. If not, refer to “Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)” on page 2-21 in the MATLAB Compiler documentation.

Set DISPLAY on UNIX Systems

If you are running a UNIX variant, such as Linux, WebFigures requires a display to be available in the Web server's environment in order for text labels to be rendered properly.

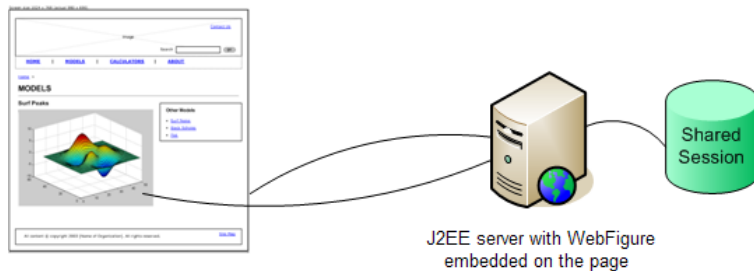
Set the `DISPLAY` environment variable to a valid X Server before running the WebFigure application.

Implement a Custom WebFigure

Overview

By following the Quick Start procedure, both the WebFigure service and the page that has the WebFigure embedded in it will be set up to reside on a single server. This configuration allows you to quickly reference your WebFigure from a JSP page with minimal configuration.

Using the WebFigure Control On the Frontend Servers



Setting Up the Web Server

Ensure that your Web server is properly configured with the required components by performing these steps:

- “Install and Configure Apache Tomcat” on page 8-9
- “Install `javabuilder.jar`” on page 8-11
- “Install the Web Archive (WAR)” on page 8-12

Install and Configure Apache Tomcat

- 1 Download Apache Tomcat from the Apache Web site.
- 2 Install the product using an available port number. Note the port number you choose for future reference.

- 3 Navigate to C:\Program Files\Apache Software Foundation\Tomcat x.x\conf.
- 4 Using a text editor, edit tomcat-users.xml.
- 5 Browse to find the section listing sample users and roles, usually at the end of the file:

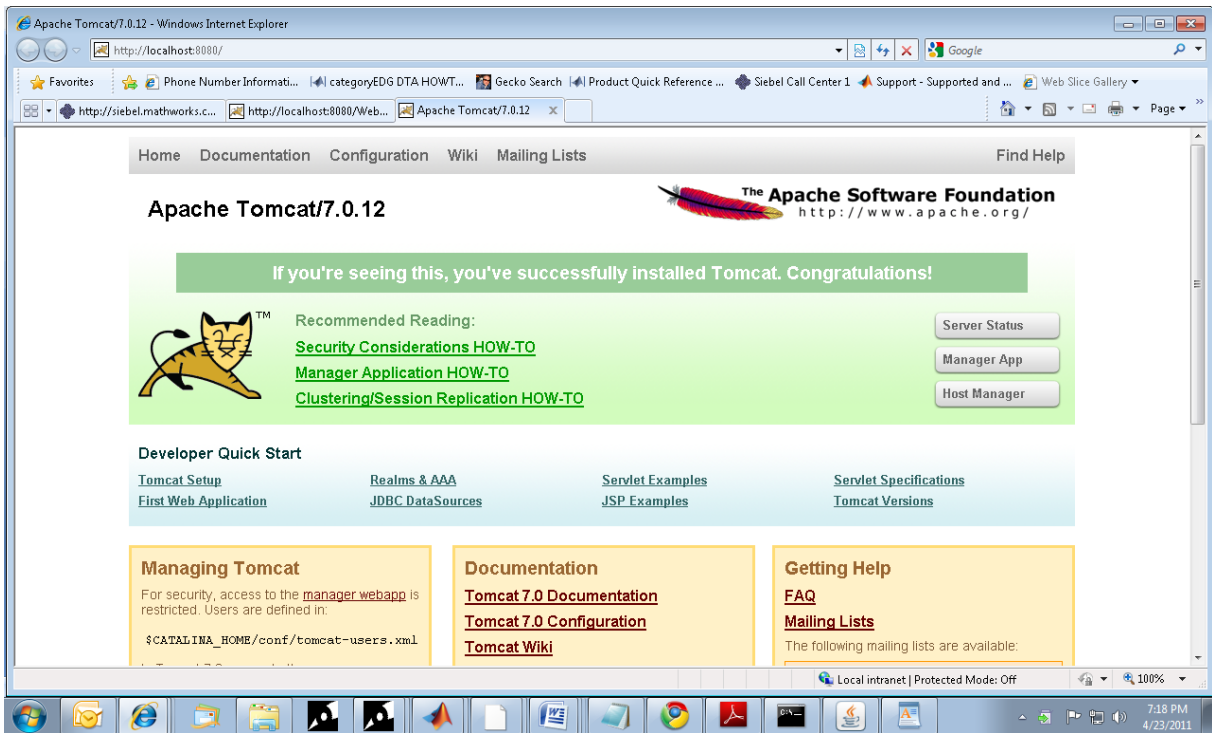
```
<role rolename="manager"/>
<role rolename="admin"/>
<user username="admin" password="borg" roles="admin,manager-
gui,manager-script"/>
```

Tomcat Role and User Listing in tomcat-users.xml

- 6 Remove comments from *all* the role and user statements (`<!-- ... -->`).
- 7 Modify the `<user` statement, customizing it with a username and password of your choice. In the example above, the chosen username is `admin` and the password is `borg`. The `roles=` parameter assigns specific roles and accorded privileges to this user, defined earlier in the `<role` statements. See the Apache Tomcat documentation for further information regarding users and roles.
- 8 Save and close tomcat-users.xml.
- 9 Open a browser session and enter this URL:

```
http://localhost:port_number/
```

For example, if you chose port number 8080 when you installed Tomcat, you would enter: `http://localhost:8080/`. An image similar to the following should appear:



This image indicates that you have successfully set up Apache Tomcat.

Install javabuilder.jar

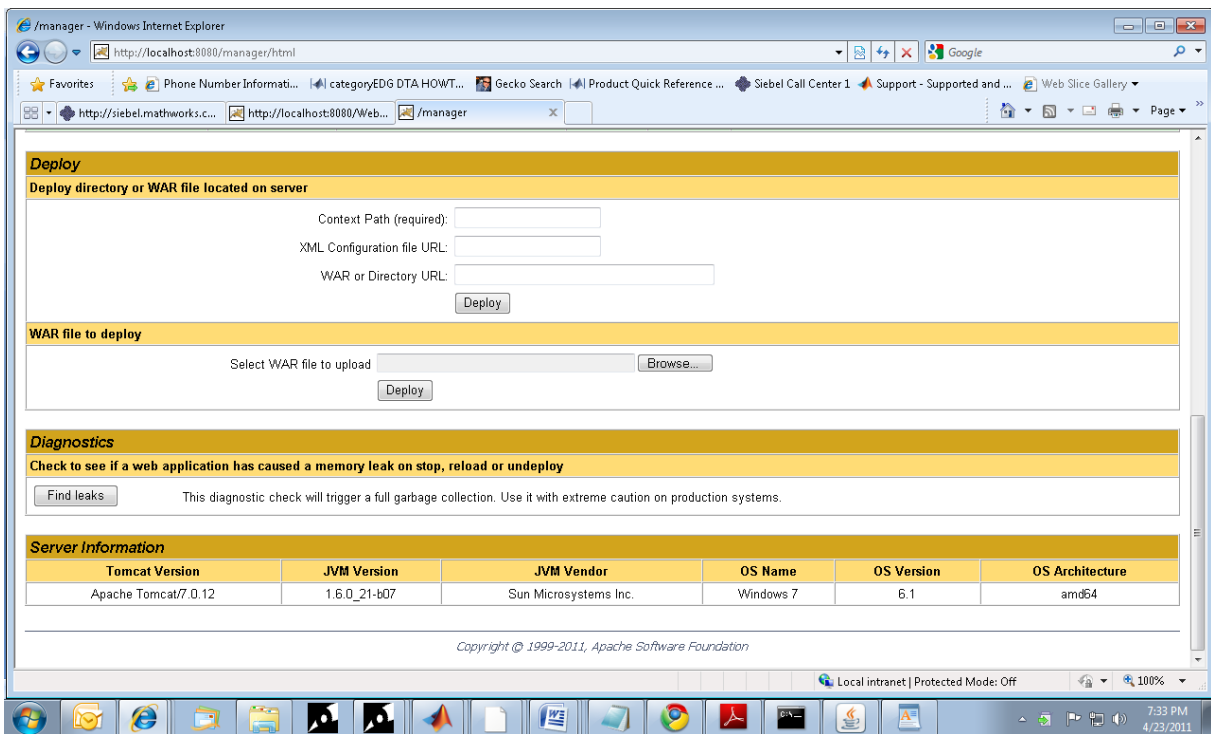
- 1 From MATLAB, navigate to the folder: C:\Program Files\MATLAB\release_name\toolbox\javabuilder\jar\.

Caution This file uses native resources. It is critical that it exist in your Web server's class PATH only once. Embedding this file into Web applications causes errors.

- From this folder, copy `javabuilder.jar` to `C:\Program Files\Apache Software Foundation\Tomcat x.x\lib`. By doing this, you are adding the MATLAB Builder JA JAR file to Apache Tomcat folder of global JARs.

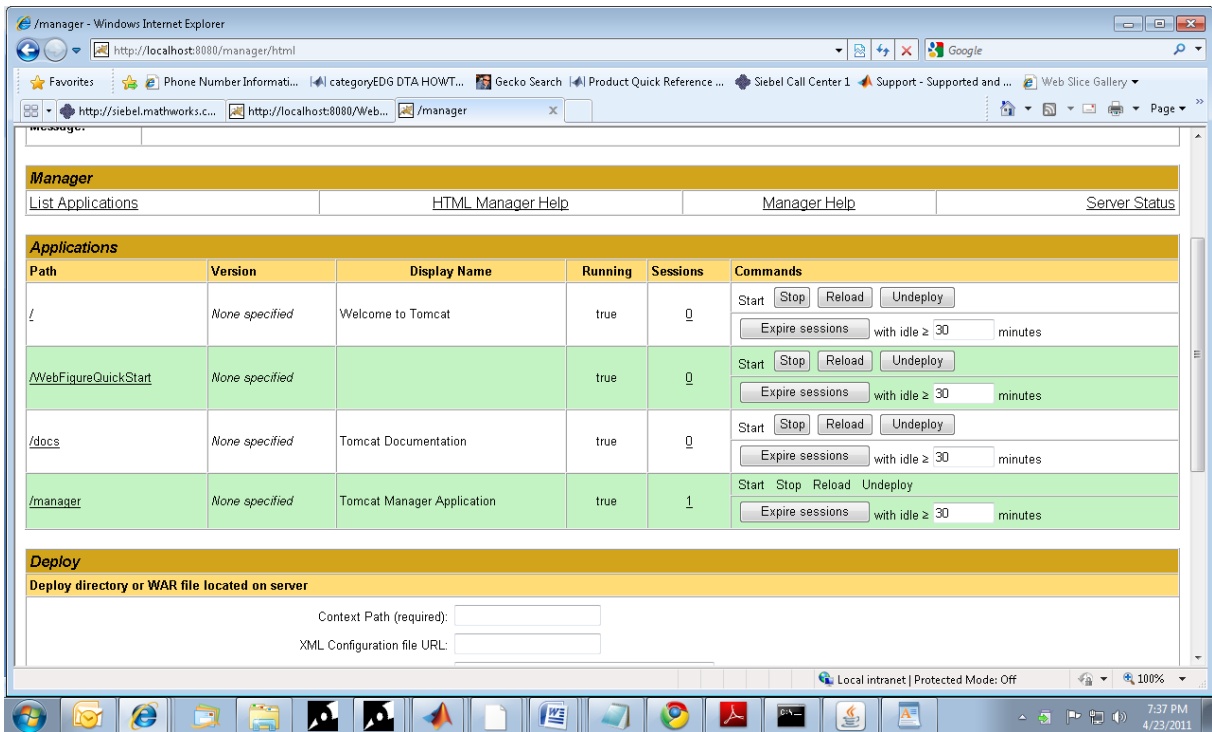
Install the Web Archive (WAR)

- In the browser session you started in “Install and Configure Apache Tomcat” on page 8-9, click the **Manager App** button, displayed in the above screenshot.
- On the **Tomcat Web Application Manager** page, find the section **WAR File to Deploy**:



- To the right of the field **Select WAR file to upload**, click **Browse**.

- 4 Navigate to the folder C:\Program Files\MATLAB\release_name\toolbox\javabuilder\jar\ and select WebFigureQuickStart.war.
- 5 In the **WAR File to Deploy** section, click **Deploy**.
- 6 WebFigureQuickStart should now be listed in the **Applications** section of the **Tomcat Web Application Manager** page:



You are now ready to create your first WebFigure.

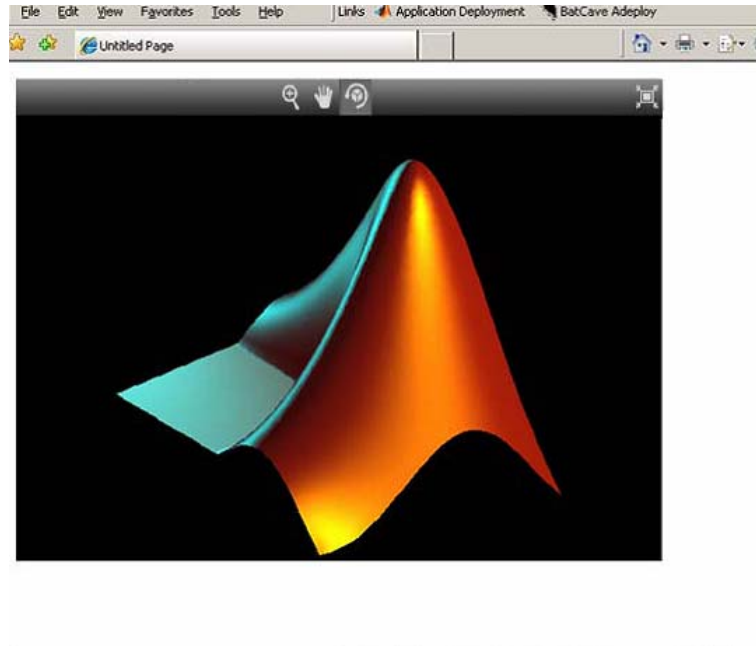
Create the Default WebFigure

- 1 Start up your Web server.
- 2 Open a browser and navigate to the JSP file contained in the WebFigureQuickStart application. If you are running this locally, the URL

is:

```
http://hostName:portNumber/WebFigureQuickStart/WebFigureExample.jsp
```

The following default figure page appears:



Behind the Scenes: How a WebFigure Is Referenced

The Web application that MathWorks ships contains a reference to a servlet in `WebFigureQuickStart.war` (installed in “Setting Up the Web Server” on page 8-9). The JSP file instantiates a deployed class that is also in `WebFigureQuickStart.war` and attaches it to the application scope of the server. It uses the JSP tag to reference the figure on the page.

Interact with the Default WebFigure

Interact with the default figure on the page using your mouse:

- 1 Click one of the three control icons at the top of the figure to select the desired control.

- 2 Select the region of the figure you want to manipulate.
- 3 Click and drag to manipulate the figure. For example, to zoom in the figure, click the magnifying glass icon, then hover over the figure.

Create a Custom WebFigure

After you access the default figure, add one of your own figures:

- 1 Ensure you have done the following with the MATLAB code referenced in “Assumptions About the Examples” on page 8-7 (or your own MATLAB code):
 - Tested the code
 - Compiled the code using MATLAB Builder JA
 - Successfully generated the *yourcomponent.jar* file from the compilation.

For more information on this process, please see example in this guide.

- 2 Test to ensure that your Web server is functioning. You can do this by creating a JSP Web page, deploying it to your server, and then attempting to access it.
- 3 Create a new Web application and an associated JSP file within that application. In the `web.xml` file for your Web application (in the `WEB-INF` folder in a Web application), add the following reference to the built-in `WebFigureServlet`:

```
<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>
```

- 4 Copy `MATLABROOT/toolbox/javabuilder/webfigures/webfigures.tld`, the WebFigures customer tag handler file, to the WEB-INF folder under your Web application directory.
- 5 In the JSP file, add a reference to the WebFigure tag by including the following line of code at the beginning of the file. The URI listed here is for example purposes only.

```
<%@ taglib
    prefix="wf"
    uri="http://www.mathworks.com/builderja/webfigures.tld"
%>
```

- 6 Add an actual WebFigure tag in the body of the page:

```
<wf:web-figure />
```

- 7 At this point, test that the configuration is working properly following the changes you previously made. By having an empty WebFigure tag, the WebFigureService automatically displays the default WebFigure and the resulting page should resemble that achieved in “Create the Default WebFigure” on page 8-13.
- 8 Reference the previously built and deployed package from your JSP page:

- a Add the following import statement to your JSP page:

```
<%@ page import="yourComponentsPackage.YourComponentsClass" %>
```

- b Add the following import statement to invoke the WebFigure:

```
<%@ page import="com.mathworks.toolbox.javabuilder.webfigures.WebFigure" %>
```

- c Add the following statement to enable access to MWJavaObjectRef:

```
<%@ page import="com.mathworks.toolbox.javabuilder.*" %>
```

- 9 Instantiate the deployed class and call the method that will return the WebFigure, as in this sample code:

```
<%
    MyComponentClass myDeployedComponent = null;
```

```

try {
    //Instantiate the Deployed Component
    myDeployedComponent = new MyComponentClass();

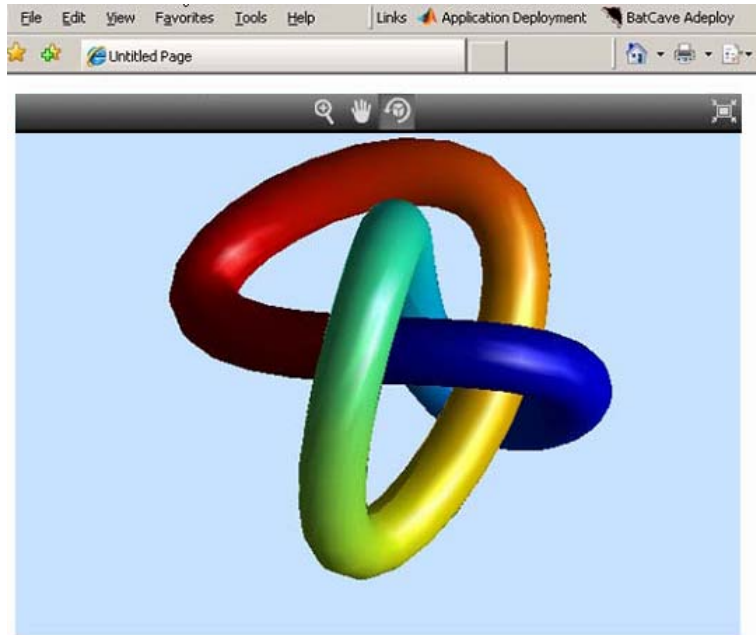
    try {
        WebFigure webFigure = (WebFigure)
            ((MWJavaObjectRef)myDeployedComponent.getKnot(1)[0]).get();

        //Get the WebFigure from your function's output
        // and set it to the tag
        request.getSession().setAttribute("YourFigure", webFigure);
    }
    catch(ClassCastException e)
    {
        throw new Exception
            ("
Issue casting deployed components outputs to WebFigure", e);
    }
}
catch (Exception e) {
    e.printStackTrace();
}
finally {
    //Dispose of the Deployed Component
    // (necessary since this has native resources).
    myDeployedComponent.dispose();
}
}%>
<wf:web-figure name="YourFigure" scope="session"/>

```

10 Run your application. Your custom WebFigure appears:

8 Deploying a Java® Package Over the Web



Advanced Configuration of a WebFigure

In this section...

“Overview” on page 8-19

“How Do WebFigures Work?” on page 8-21

“Installing WebFigureService” on page 8-22

“Getting the WebFigure Object from Your Method” on page 8-23

“Attach a WebFigure” on page 8-24

“Using the WebFigure JSP Tag to Reference a WebFigure” on page 8-26

“Getting an Embeddable String That References a WebFigure Attached to a Cache” on page 8-29

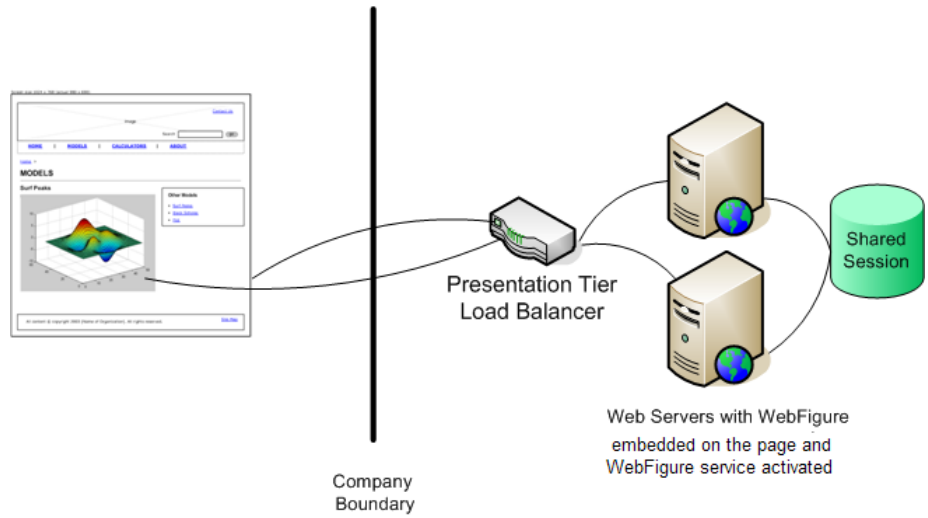
Overview

The advanced configuration gives the experienced application developer flexibility and control in configuring system architecture based on differing needs. For example, with the `WebFigureService` and the `Web` page on different servers, the administrator can optimally position the MCR (for performance reasons) or place customer-sensitive customer data behind a security firewall, if needed.

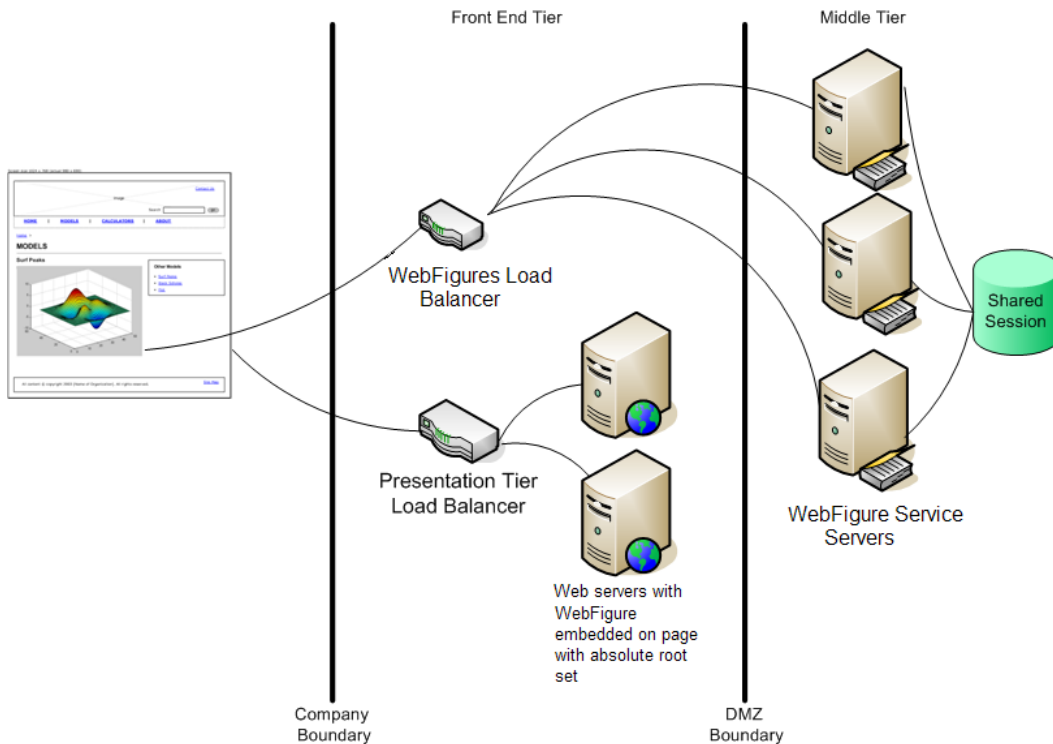
This section describes various ways to customize the basic `WebFigures` implementation described in “Implement a Custom `WebFigure`” on page 8-9.

The advanced configuration offers more choices and adaptability for the user more familiar with `Web` environments and related technology, as illustrated by the following graphics.

Using the Web Figure Control On the Frontend Servers



Using Absolute Root to Keep the WebFigure Service Cluster Behind the Firewall



How Do WebFigures Work?

When choosing the best architecture for your configuration, it is important to understand the fundamental components that enable an application.

WebFigures is made up of several different components that work together:

- Your Web application
- Client-side code
- WebFigureService

- Your server's cache

When you enable a user to rotate a figure, for example, you are using standard AJAX techniques to request different static images depending on the requested orientation. `WebFiguresService` (which is exposed by referencing `WebFigureServlet` in `web.xml`) delivers the HTML and JavaScript® to a browser, getting the defaults for a figure, and rendering a figure in any of its available orientations.

Your Web application calls one of your deployed components to get the specific `WebFigure`, and attaches it to your server's cache for `WebFiguresService` to use later. Your application also puts an HTML reference to your `WebFigure` on a page. This can either be done automatically using the JSP tag or manually by using `WebFigureHtmlGenerator`. This gives the client browser what it needs to request the client-side code.

The client-side JavaScript AJAX code provides a user experience similar to that in MATLAB when using a `FIGURE`. It provides rotation, zooming, and panning in a highly usable medium by using a JavaScript application that monitors for user interaction such as dragging or clicking with a mouse, and calls back into `WebFiguresService` to service those requests.

For example, when a user selects the rotate icon and clicks in the `WebFigureTag` and drags it, that drag translates to coordinates and issues a request for the new rotated image from `WebFiguresService`. A rotating cube is displayed so the user knows what orientation they are looking at. Since there is no efficient way to pass an actual `WebFigure` from your application to the client-side application and then back to `WebFiguresService`, the server's built-in cache is leveraged as a central repository.

Installing WebFigureService

In order for the client-side code to call back to request images, you need a reference to the built-in servlet in the application's `web.xml` file. This reference should look like this:

```
<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>

```

Note You can rename the servlet and its mapping. If you do, you must reference it whenever you interact with either the `WebFigureHtmlGenerator` or the JSP tag (using the `root` attribute) so it can call back to the servlet. It is critical that the URL pattern you map to ends with the literal `/*`. This allows all end points to be redirected to the servlet. To test this end point at any time, navigate to it in a browser and you should see the default Web page. If you have `web.xml` set correctly in your application, the URL will look something like `http://hostName:portNumber/yourWebApplication/WebFigures`.

Getting the WebFigure Object from Your Method

In MATLAB Builder JA, when a `WebFigure` Java Object is created in MATLAB code and returned, you convert that `MWJavaObjectRef` into a regular `WebFigure` in order to access it. For example:

```

try
{
    Object[] results = null;
    try
    {
        //This assumes there is only a single option
        // from your function
        // and it has no inputs.
        results = myDeployedComponent.getSurfPeaks(1);

        //Since we know the only output is the WebFigure
        // we get the MWJavaObjectRef from
        // the Object Array.
        //By calling "get" on the MWJavaObjectRef we
        // retrieve the
        // actual object from it.
    }
}

```

```
        WebFigure myFigure =
            (WebFigure)((MWJavaObjectRef)results[0]).get();
    }
    finally
    {
        //Deployed Components use MWArrays which utilize
        // Native Resources.
        //The Java Garbage collector can not properly
        // clean up this memory so it is important to
        // dispose these resources.
        MWArray.disposeArray(results);
    }
}
catch(ClassCastException e)
{
    throw new Exception
        ("WebFigure object was not
         of Type WebFigure.", e);
}
```

Attach a WebFigure

All components access available WebFigures is by using Web server cache mechanisms. This allows you to leverage built-in J2EE mechanisms to scale your servers into a farm and automatically propagate the session across the servers.

There are a number of ways to attach a WebFigure to a scope, depending on the state:

- Attaching to the session cache session
- Attaching to the application cache application

Attaching to the Session Cache

This cache is visible only to the current user in a system and is usually used to store user session-specific information.

Attaching to the session cache can be an ideal choice if the figure is valid only for a specific user, at a certain time. To do this, add the following line of code to a JSP scriptlet or a servlet:

```
//from a JSP scriptlet or a servlet to the Session cache  
request.getSession().setAttribute("myFigure", myFigure);
```

If you manually attached the figure, but want the JSP tag to reference it, you can add the tag attributes:

```
name="myFigure" scope="session"
```

Note The name given to the JSP tag must match the one used to attach it to a cache, and the name must be unique within that cache.

Attaching to the Application Cache

This cache is visible by all sessions in the current application. Attach to the application cache if you want to attach the figure globally for every page and servlet to use.

To attach to the Application scope, add the following line of code to a JSP scriptlet or a servlet:

```
//from a JSP scriptlet or a servlet to the Application cache  
request.getSession().getServletContext().setAttribute("GlobalFigure",  
    myFigure);
```

If you manually attached the figure, but want the JSP tag to reference it, you can add the tag attributes:

```
name="GlobalFigure" scope="application"
```

Note The name given to the JSP tag must match the one used to attach it to a cache, and the name must be unique within that cache.

Using the WebFigure JSP Tag to Reference a WebFigure

Once the WebFigure has been retrieved from the function output (see “Getting the WebFigure Object from Your Method” on page 8-23), you can attach it to one of your server’s caches and reference it from the JSP tag.

Initializing the JSP Tag

Reference the tag library by adding the following line to a JSP page:

```
<%@ taglib
    prefix="wf"
    uri="http://www.mathworks.com/builderja/webfigures.tld"
%>
```

Note This code references the `.tld` file from the `WEB-INF` folder under your web application folder. This URI must be typed exactly as shown above for the name to properly resolve the reference. Once this tag has been referenced, you can add tags to the page similar to this:

```
<wf:web-figure />
```

Note If you use an empty tag as shown above, the default WebFigure appears. To bind the tag to your WebFigure, see “Attach a WebFigure” on page 8-24.

Attributes of a WebFigure Tag

The key attributes for the WebFigure tag are `name` and `scope`. For each tag, use these parameters to indicate which figure to use in which cache on your server. Assuming you have attached a figure to the session cache using the string `MyFigure` (as shown in the “Attach a WebFigure” on page 8-24), the JSP tag resembles this:

```
<wf:web-figure name="MyFigure" scope="session"/>
```

Use this table to reference the following WebFigure tag attributes.

WebFigure Tag Attributes and Their Default Values

Attribute Name	Description	Optional?	Default Value
<code>name</code>	Name used when attaching your figure to a cache. Case sensitive.	Yes	The name of the default WebFigure built into WebFigureService. If you provide an empty WebFigure tag, this figure is displayed.
<code>scope</code>	Scope that your figure has been saved to (either <code>application</code> or <code>session</code>).	Yes	If this is not specified, an error is thrown unless the name is also not specified. In this case, the default figure is attached to the session scope and is used.
<code>style</code>	Style attribute that you want embedded and attached to the iFrame.	Yes	If this is not passed, a basic iFrame is used.
<code>height</code>	Height of the iFrame that will be embedded.	Yes	If this is not passed, the height of the WebFigure is retrieved from cache.

WebFigure Tag Attributes and Their Default Values (Continued)

Attribute Name	Description	Optional?	Default Value
width	Width of the iFrame that will be embedded.	Yes	If this is not passed, the width of the WebFigure is retrieved from cache.
root	Name used to map the WebFiguresServlet for a figure.	Yes	If this is not specified, it is assumed to be mapped to WebFigures. If it is specified to a relative servlet end point, that is used.

Getting an Embeddable String That References a WebFigure Attached to a Cache

If you do not want to use the WebFigure JSP tag to display the figure, or want a servlet to display it directly, use this method to get a snippet of HTML that will embed an iFrame containing the figure in another page.

- 1 Create an instance of the `WebFigureHtmlGenerator` class using either a scriptlet or a servlet. The constructor for this class has three overloads:

```
//The import statement needed to invoke this class
import com.mathworks.toolbox.javabuilder.webfigures.WebFigureHtmlGenerator;

//WebFigureHtmlGenerator(HttpServletRequest servletRequest)
//This overload just takes the ServletRequest and will map the
// embed string to the same server and assumes that the
// WebFiguresService was mapped to "WebFigures"
WebFigureHtmlGenerator htmlGenerator =
    WebFigureHtmlGenerator(servletRequest);

//OR

//WebFigureHtmlGenerator(String webFigureServletNameMapping, HttpServletRequest
//     servletRequest)
//This overload takes the ServletRequest and the name that
// the WebFigureServlet was mapped to.
//It will reference this servlet on the same server
WebFigureHtmlGenerator htmlGenerator =
    WebFigureHtmlGenerator("SomeServletMappingName", servletRequest);

//OR

//WebFigureHtmlGenerator(String absolutePathName)
//This overload takes the absolute URL path to a server that has
// WebFiguresService running.
//This would be used if you have a cluster of servers that are all running
// WebFigureService
// a load balancer (all sharing cache state). Use
// this parameter to reference that base load balancer URL.
WebFigureHtmlGenerator htmlGenerator =
```

```
WebFigureHtmlGenerator("http://someLoadBalancer/someWebApplication/
WebFigureServletNameMapping");
```

- 2** Call the method to get the embedded string (getFigureEmbedString). Use this table to specify appropriate attributes:

Attribute Name	Attribute Type	Description	Optional	Default Value
figure	WebFigure	WebFigure for which you want to create the embedded string.	Yes	This is used to determine the figure's default height and width if no other is provided .
name	String	Name used when attaching your figure to a cache. Case sensitive.	No	Not optional
scope	String	Scope that figure has been saved to (application or session).	No	Not optional
style	String	Embedded attribute you want attached to the iFrame.	Yes	If this is not passed, a basic iFrame is used.
height	String	Height of the iFrame that will be embedded.	Yes	If this is not passed, the height of the WebFigure is retrieved from cache. If the WebFigure cannot be found, the MATLAB default height for

Attribute Name	Attribute Type	Description	Optional	Default Value
				a figure (420) is used.
width	String	Width of the iFrame that will be embedded.	Yes	If this is not passed, the width of the WebFigure is retrieved from cache. If the WebFigure cannot be found, the MATLAB default width for a figure (560) is used.

Working with MATLAB Figures and Images

- “Your Role in Working with Figures and Images” on page 9-2
- “Create and Modify a MATLAB Figure” on page 9-3
- “Working with MATLAB Figure and Image Data” on page 9-6

Your Role in Working with Figures and Images

When you work with figures and images as a MATLAB programmer, you are responsible for:

- Preparing a MATLAB figure for export
- Making changes to the figure (optional)
- Exporting the figure
- Cleaning up the figure window

When you work with figures and images as a front-end Web developer, some of the tasks you are responsible for include:

- Getting a WebFigure from a deployed component
- Getting raw image data from a deployed component converted into a byte array
- Getting a buffered image from a component
- Getting a buffered image or a byte array from a WebFigure

Create and Modify a MATLAB Figure

In this section...

“Preparing a MATLAB Figure for Export” on page 9-3


“Changing the Figure (Optional)” on page 9-3

“Exporting the Figure” on page 9-4

“Cleaning Up the Figure Window” on page 9-4

“Modify and Export Figure Data” on page 9-5

MATLAB Programmer

Role	Knowledge Base	Responsibilities
 MATLAB programmer	<ul style="list-style-type: none"> • MATLAB expert • No IT experience • No access to IT systems 	<ul style="list-style-type: none"> • Develops models; implements in MATLAB • Uses tools to create a component that is used by the Java developer

Preparing a MATLAB Figure for Export

1 Create a figure window. For example:

```
h = figure;
```

2 Add graphics to the figure. For example:

```
surf(peaks);
```

Changing the Figure (Optional)

Optionally, you can change the figure numerous ways. For example:

Alter Visibility

```
set(h, 'Visible', 'off');
```

Change Background Color

```
set(h, 'Color', [.8,.9,1]);
```

Alter Orientation and Size

```
width=500;  
height=500;  
rotation=30;  
elevation=30;  
set(h, 'Position', [0, 0, width, height]);  
view([rotation, elevation]);
```

Exporting the Figure

Export the contents of the figure in one of two ways:

WebFigure

To export as a WebFigure:

```
returnFigure = webfigure(h);
```

Image Data

To export image data, for example:

```
imgform = 'png';  
returnByteArray = figToImStream(`figHandle', h, ...  
                                `imageFormat', imgform, ...  
                                `outputType', `uint8');
```

Cleaning Up the Figure Window

To close the figure window:

```
close(h);
```

Modify and Export Figure Data

WebFigure

```
function returnFigure = getWebFigure()
h = figure;
set(h, 'Visible', 'off');
surf(peaks);
set(h, 'Color', [.8,.9,1]);
returnFigure = webfigure(h);
close(h);
```


Image Data

```
function returnByteArray = getImageDataOrientation(height,
width, elevation, rotation, imageFormat )
h = figure;
set(h, 'Visible', 'off');
surf(peaks);
set(h, 'Color', [.8,.9,1]);
set(h, 'Position', [0, 0, width, height]);
view([rotation, elevation]);
returnByteArray = figToImStream(`figHandle', h, ...
                                `imageFormat', imageFormat, ...
                                `outputType', `int8');
close(h);
```

Working with MATLAB Figure and Image Data

In this section...
“For More Comprehensive Examples” on page 9-6
“Working with Figures” on page 9-6
“Working with Images” on page 9-7

Front-End Web Developer

Role	Knowledge Base	Responsibilities
 <p>Front-end Web developer</p>	<ul style="list-style-type: none"> • No MATLAB experience • Minimal IT experience • Expert at usability and Web page design • Minimal access to IT systems • Expert at JSP 	<ul style="list-style-type: none"> • As service consumer, manages presentation and usability • Creates front-end applications • Integrates MATLAB code with language-specific frameworks and environments • Integrates WebFigures with the rest of the Web page

For More Comprehensive Examples

This section contains code snippets intended to demonstrate specific functionality related to working with figure and image data.

To see these snippets in the context of more fully-functional multi-step examples, see the *“The MATLAB Application Deployment Web Example Guide”*.

Working with Figures

Getting a Figure From a Deployed Component

For information about how to retrieve a figure from a deployed component, see “Implement a Custom WebFigure” on page 8-9

Working with Images

Getting Encoded Image Bytes from an Image in a Component

Java

```
public byte[] getByteArrayFromDeployedComponent()
{
    Object[] byteImageOutput = null;
    MWNumericArray numericImageByteArray = null;
    try
    {
        byteImageOutput =
            deployment.getImageDataOrientation(
                1,      //Number Of Outputs
                500,   //Height
                500,   //Width
                30,    //Elevation
                30,    //Rotation
                "png"  //Image Format
            );

        numericImageByteArray =
            (MWNumericArray)byteImageOutput[0];
        return numericImageByteArray.getBytes();
    }
    finally
    {
        MWArray.disposeArray(byteImageOutput);
    }
}
```

Getting a Buffered Image in a Component

Java

```
public byte[] getByteArrayFromDeployedComponent()
{
```

```
Object[] byteImageOutput = null;
MWNumericArray numericImageByteArray = null;
try
{
    byteImageOutput =
        deployment.getImageDataOrientation(
            1,      //Number Of Outputs
            500,   //Height
            500,   //Width
            30,    //Elevation
            30,    //Rotation
            "png"  //Image Format
        );

    numericImageByteArray =
        (MWNumericArray)byteImageOutput[0];
    return numericImageByteArray.getBytes();
}
finally
{
    MWArray.disposeArray(byteImageOutput);
}
}

public BufferedImage getBufferedImageFromDeployedComponent()
{
    try
    {
        byte[] imageByteArray =
            getByteArrayFromDeployedComponent()
            return ImageIO.read
                (new ByteArrayInputStream(imageByteArray));
    }
    catch(IOException io_ex)
    {
        io_ex.printStackTrace();
    }
}
```

Creating Scalable Web Applications Using RMI

- “Using Remote Method Invocation (RMI)” on page 10-2
- “RMI Prerequisites” on page 10-3
- “Run the Client and Server on a Single Machine” on page 10-4
- “Run the Client and Server on Separate Machines” on page 10-8
- “Use Native Java with Cell Arrays and Struct Arrays” on page 10-9
- “Additional RMI Examples” on page 10-16

Using Remote Method Invocation (RMI)

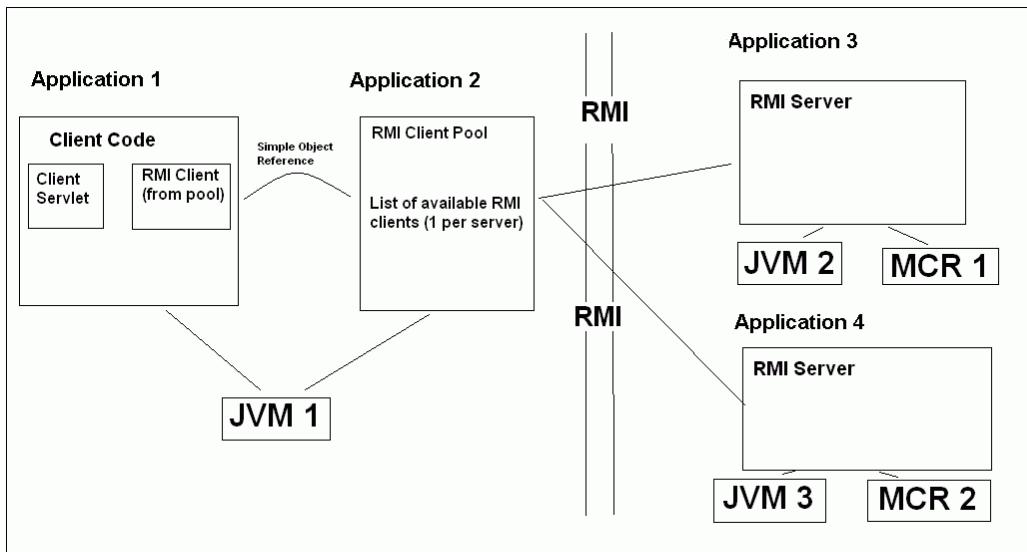
You can expand your application's throughput capacity by taking advantage of the MATLAB Builder JA product's use of RMI, the Java native remote procedure call (RPC) mechanism. The builder's implementation of RMI technology provides for automatic generation of interface code to enable components to start 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 and server on separate machines

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.



RMI Prerequisites

See “Web Application” on page 7-47 for information on properly setting up your Java environment before you run the example in this section.

Run the Client and Server on a Single Machine

The following example shows how to run two separate processes to initialize MATLAB struct arrays.

Note You do not need 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 Javadoc API documentation for details at `matlabroot/help/toolbox/javabuilder/MWArrayAPI`.

- 1 Compile the MATLAB Builder JA package by issuing the following command 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_package*.

```
javac -classpath  
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;  
directory_containing_package\dataTypesComp.jar  
DataTypesServer.java
```

You can find `DataTypesServer.java` in:

```
matlabroot\toolbox\javabuilder\Examples\RMIExamples  
\DataTypes\DataTypesDemoJavaApp
```

- 3 Compile the client Java code by issuing the following `javac` command. Ensure there are no spaces between `javabuilder.jar`; and *directory_containing_package*.

```
javac -classpath  
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
```

```
directory_containing_pacakage\dataTypesComp.jar
DataTypesClient.java
```

4 Run the client and server as follows:

- a** Open two command windows on DOS or UNIX, depending on the platform you are using.
- b** If running Windows, ensure that *matlabroot/runtime/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_pacakage\dataTypesComp.jar;
matlabroot\toolbox\javabuilder\jar\javabuilder.jar
-Djava.rmi.server.codebase=
"file:///matlabroot/toolbox/javabuilder/jar/javabuilder.jar
file:///directory_containing_pacakage/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_pacakage\dataTypesComp.jar;
matlabroot\toolbox\javabuilder\jar\javabuilder.jar
DataTypesClient
```

You can find `DataTypesClient.java` in:

```
matlabroot\toolbox\javabuilder\Examples\RMIExamples\DataTypes
\DataTypesDemoJavaApp.
```

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'

#####

Run the Client and Server on Separate Machines

To implement RMI with a client on one machine and a server on another, you must:

- 1 Change how the server is bound to the system registry.
- 2 Redefine how the client accesses the server.

After this is done, follow the steps in “Run the Client and Server on a Single Machine” on page 10-4.

Use Native Java with Cell Arrays and Struct Arrays

In this section...

“Why Use Native Type Cell Arrays and Struct Arrays?” on page 10-9

“Native Type Data Marshaling Prerequisites” on page 10-10

“Native Java Cell and Struct” on page 10-10

Why Use Native Type Cell Arrays and Struct Arrays?

In Java, there is no direct representation available for MATLAB-specific struct and cell arrays.

As a result, when an instance of `MWStructArray` or `MWCellArray` is converted to a Java native type using the `toArray()` method, the output is a multi-dimensional `Object` array which can be difficult to process.

When you use MATLAB® Builder™ JA components with RMI, however, you have control over how the server sends the results of MATLAB function calls back to the client. The server can be set to marshal the output to the client as an `MWArray` (`com.mathworks.toolbox.javabuilder` package) sub-type or as a Java™ native data type. The Java native data type representation of `MWArray` subtypes is obtained by invoking the `toArray()` method by the server.

Using Java native representations of MATLAB struct and cell arrays is recommended if both of these are true:

- You have MATLAB functions on a server with MATLAB struct or cell data types as inputs or outputs
- You do not want or need to install an MCR on your client machines

Using Native Types Does Not Require a Client-Side MCR

The classes in the `com.mathworks.extern.java` package (in `javabuilder.jar`) do not need an MCR. The names of the classes in this package are the same as those in `com.mathworks.toolbox.javabuilder` — allowing the end-user to easily create instances of `com.mathworks.extern.java.MWStructArray` or `com.mathworks.extern.java.MWCellArray` that work the same as the

like-named classes in `com.mathworks.toolbox.javabuilder` — on a machine that does not have an MCR.

The availability of an MCR on the client machine dictates how the server should be set for marshaling MATLAB functions, since the `MWArray` class hierarchy can be used only with an MCR. If the client machine does not have an MCR available, the server returns the output of `toArray()` for cell or struct arrays as instances of `com.mathworks.extern.java.MWStructArray` or `com.mathworks.extern.java.MWCellArray`.

Native Type Data Marshaling Prerequisites

Even though client machines don't need to have an MCR, they do need to have `javabuilder.jar` since it contains the `com.mathworks.extern.java` package.

Please refer to the Javadoc (<matlabroot/help/toolbox/javabuilder/MWArrayAPI>) for more information about classes in all MATLAB Builder JA packages.

Native Java Cell and Struct

Before You Run the Example

Before you run this example, note the following:

- This example demonstrates how to implement RMI when the client and the server are running on the same machine. See “Run the Client and Server on Separate Machines” on page 10-8 if you would like to do otherwise.
- On UNIX, use `:` as the path separator in calls to `java` and `javac`. Use `;` as a path separator on Windows.
- Only update the server system path with the location of the MCR. The client does not need access to the MCR.
- This example is shipped in the `matlab\toolbox\javabuilder\Examples\RMIExamples\NativeCellStruct` directory.
- Ensure that:

- On Windows systems, *matlabroot/runtime/arch* is on the system path.
- On UNIX systems, LD_LIBRARY_PATH and DYLD_LIBRARY_PATH are set properly. See “Modifying the Path” in the MATLAB Compiler User’s Guide for further information on setting the path.

Running the Example

Note Be sure to enter the following as single, unbroken commands.

- 1 Use the following `mcc` command to build the package:

```
mcc -W
'java:dataTypesComp,dataTypesClass' createEmptyStruct.m
                                updateField.m -v
```

- 2 Compile the server’s Java code:

```
javac -classpath
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
directory_containing_pacakage\dataTypesComp.jar
NativeCellStructServer.java
```

- 3 Compile the client’s Java code:

```
javac -classpath
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
directory_containing_pacakage\dataTypesComp.jar
NativeCellStructClient.java
```

- 4 Prepare to run the server and client applications by opening two DOS or UNIX command windows—one for client and one for server.

- 5 Run the server:

```
java -classpath
.;directory_containing_pacakage\dataTypesComp.jar;
    matlabroot\toolbox\javabuilder\jar\javabuilder.jar
-Djava.rmi.server.codebase="file:///matlabroot/toolbox/javabuilder/
```

```
jar/javabuilder.jar file:///
directory_containing_package/dataTypesComp.jar"
NativeCellStructServer
```

6 Run the client:

```
java -classpath
.;directory_containing_package\dataTypesComp.jar;
matlabroot\toolbox\javabuilder\jar\javabuilder.jar
NativeCellStructClient
```

7 If your application has run successfully, the output will display as follows:

- **Server output:**

```
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]
```

```
#####
```

- **Client output:**

Running the client application!!

EVENT 2: Initialized structure as received in client applications:

1x1 struct array with fields:

Name
Address

Updating the 'Address' field to :

1x1 struct array with fields:

Street
City
State
Zip

#####

EVENT 5: Final structure as received by client:

1x1 struct array with fields:

Name
Address

Address field:

1x1 struct array with fields:

Street
City
State
Zip

#####

Additional RMI Examples

For more examples of RMI implementation, see the examples in *matlabroot/toolbox/javabuilder/Examples/RMIExamples*.

Troubleshooting

Common MATLAB Builder JA Error Messages

Exception in thread "main" java.lang.UnsatisfiedLinkError: Failed to find the library mclmcr712.dll, required by MATLAB Builder JA, on java.library.path

Install the MCR or add it to the MATLAB path.

Failed to find the library <library_name>, required by MATLAB Builder JA, on java.library.path.

This error commonly occurs on Linux or Mac systems if the LD_LIBRARY_PATH variable is not set.

See and in the *MATLAB Compiler User's Guide*.

'javac' is not recognized as an internal or external command, operable program or batch file.

This is a common error when the javac executable (javac.exe), installed with Java, is not on your system PATH.

Edit your system environment variables and add your Java installation folder to the PATH variable.

Reference Information for Java

- “Requirements for the MATLAB® Builder™ JA Product” on page 12-2
- “Data Conversion Rules” on page 12-4
- “Programming Interfaces Generated by the MATLAB® Builder™ JA Product” on page 12-8
- “MWArray Class Specification” on page 12-13
- “Application Deployment Terms” on page 12-14

Requirements for the MATLAB Builder JA Product

In this section...
“System Requirements” on page 12-2
“Path Modifications Required for Accessibility” on page 12-2
“MATLAB® Builder™ JA Limitations” on page 12-3

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 in the MATLAB Compiler documentation.
- Java Development Kit must be installed.
- Java Runtime Environment (JRE) that is used by MATLAB and MCR.

Note You should be using the same version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Caution MathWorks only supports the Oracle JDK and JRE. A certain measure of cross-version compatibility resides in the Oracle software and it *may* be possible to run MCR-based components with non-Oracle JDKs under some circumstances—however, compatibility is not guaranteed.

Path Modifications Required for Accessibility

In order to use some screen-readers or assistive technologies, such as JAWS®, you must add the following DLLs to your Windows path:

JavaAccessBridge.dll
WindowsAccessBridge.dll

You may not be able to use such technologies without doing so.

MATLAB Builder JA Limitations

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 “MATLAB Compiler Limitations” in the MATLAB Compiler documentation for details.

Include -d64 Flag for Macintosh 64-Bit Systems

On 64-bit Mac systems, 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 these machines and the MATLAB Compiler Runtime (MCR) requires a 64-bit JVM. For example when running , you should use the following command:

```
matlabroot/sys/java/jre/architecture/jre_folder/bin/java  
-classpath  
:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:  
MagicDemoJavaApp/magicsquare/distrib/magicsquare.jar -d64  
getmagic 5
```

Since Leopard (Mac OS X 10.5), is capable of running both 32-bit and 64-bit operating systems, including the `-d64` flag is only required if you are running in 64-bit mode.

MATLAB Java External Interface

JAR files created by MATLAB Builder JA cannot be loaded back into MATLAB with the MATLAB Java External Interface.

MATLAB Objects

In addition, the MATLAB Builder JA product does not support MATLAB object data types (for example, Time Series objects). In other words, MATLAB objects can not "pass" the boundary of MATLAB/Java, but you are free to use objects in your MATLAB code.

Data Conversion Rules

In this section...

“Java to MATLAB Conversion” on page 12-4

“MATLAB to Java Conversion” on page 12-6

“Unsupported MATLAB Array Types” on page 12-7

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
	<hr/> <p>Note Subclasses of java.lang.Number not listed above are converted to double.</p> <hr/>
java.lang.Boolean	logical
java.lang.Character	char
java.lang.String	<hr/> <p>Note A Java string is converted to a 1-by-N array of char with N equal to the length of the input string.</p> <p>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.</p> <p>Higher dimensional arrays of String are converted similarly.</p> <p>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.</p> <hr/>

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	Not applicable	Object Note Cell arrays are constructed and accessed as arrays of <code>MWArray</code> .
structure	Not applicable	Object Note Structure arrays are constructed and accessed as arrays of <code>MWArray</code> .
char	char	<code>java.lang.Character</code>
double	double	<code>java.lang.Double</code>
single	float	<code>java.lang.Float</code>
int8	byte	<code>java.lang.Byte</code>
int16	short	<code>java.lang.Short</code>
int32	int	<code>java.lang.Integer</code>
int64	long	<code>java.lang.Long</code>

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 12-8

“Standard API” on page 12-9

“mlx API” on page 12-11

“Code Fragment: Signatures Generated for the myprimes Example” on page 12-11

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 0 elements!
```


And this would be the correct usage:

```
java.util.List outputs = Arrays.asList(new Object[3]);
myclass.myfunction(outputs, inputs); // list has 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 <i>N</i> input arguments	<pre>public Object[] foo(int numArgsOut, Object In1, Object In2, ... Object InN)</pre>
API if there are optional arguments, represented by the <i>varargin</i> argument	<pre>public Object[] foo(int numArgsOut, Object in1, Object in2, ..., Object InN, Object varargin)</pre>

Details about the arguments for these samples of standard signatures are shown in the following table.

Argument	Description	Details About Argument
<i>numArgsOut</i>	Number of outputs	<p>An integer indicating the number of outputs you want the method to return. To return no arguments, omit this argument.</p> <p>The value of <i>numArgsOut</i> must be less than or equal to the MATLAB function <i>nargout</i>.</p> <p>The <i>numArgsOut</i> argument must always be the first argument in the list.</p>
<i>In1, In2, ...InN</i>	Required input arguments	All arguments that follow <i>numArgsOut</i> in the argument list are inputs to the method being called.

Argument	Description	Details About Argument
		Specify all required inputs first. Each required input must be of class <code>MWArray</code> or any class derived from <code>MWArray</code> .
<i>varargin</i>	Optional inputs	You can also specify optional inputs if your MATLAB code uses the <code>varargin</code> input: list the optional inputs, or put them in an <code>Object[]</code> argument, placing the array last in the argument list.
<i>Out1, Out2, ...OutN</i>	Output arguments	With the standard calling interface, all output arguments are returned as an array of <code>MWArrays</code> .

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 the 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 class, 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 12-8 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/` folder.

Note For `matlabroot`, substitute the MATLAB root folder on your system. Type `matlabroot` to see this folder name.

Application Deployment Terms

Glossary of Deployment Product Terms

A

Add-in — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic®.

API — Application program interface. An implementation of the proxy software design pattern. See *MWArray*.

Application — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the Deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

Assembly — An executable bundle of code, especially in .NET. For example, after building a deployable .NET component with MATLAB Builder NE, the .NET developer integrates the resulting .NET assembly into a larger enterprise C# application. See *Executable*.

B

Binary — See *Executable*.

Boxed Types — Data types used to wrap opaque C structures.

Build — See *Compile*.

C

Class — A user-defined type used in C++, C#, and Java, among other object-oriented languages that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes

relate in a class hierarchy. One class is a specialization (a *subclass*) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a *MATLAB class*

Compile — In MATLAB Compiler terminology, to compile a component involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code builds with MATLAB Builder JA, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

COM component — In MATLAB Builder EX, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Builder NE, an executable component, to be integrated with Microsoft COM applications.

Component — In MATLAB, a generic term used to describe the wrapped MATLAB code produced by MATLAB Compiler. You can plug these self-contained bundles of code you plug into various computing environments. The wrapper enables the compatibility between the computing environment and your code.

Console application — Any application that is executed from a system command prompt window.

CTF archive (Component Technology File) — The Component Technology File (CTF) archive is embedded by default in each generated binary by MATLAB Compiler. It houses the deployable package. All MATLAB-based content in the CTF archive uses the Advanced Encryption Standard (AES) cryptosystem. See “Additional Details” in the MATLAB Compiler documentation.

D

Data Marshaling — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks data types such as represented by the `MWArray` API—must be performed manually, often at great cost.

Deploy — The act of integrating a component into a larger-scale computing environment, usually to an enterprise application, and often to end users.

DLL — Dynamic link library. Microsoft's implementation of the shared library concept for Windows. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

E

Empties — Arrays of zero (0) dimensions.

Executable — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

F

Fields — For this definition in the context of MATLAB Data Structures, see *Structs*.

Fields and Properties — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.

H

Helper files — Files that support the main file or the file that calls all supporting functions. Add resources that depend upon the function that calls the supporting function to the **Shared Resources and Helper Files** section of the Deployment Tool GUI. Other examples of supporting files or resources include:

- Functions called using `eval` (or variants of `eval`)
- Functions not on the MATLAB path
- Code you want to remain private
- Code from other programs that you want to compile and link into the main file

I

Integration — Combining a deployed component’s functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment. Integration is usually performed by an IT developer, rather than a MATLAB Programmer, in larger environments.

Instance — For the definition of this term in context of MATLAB Production Server software, see *MATLAB Production Server Server Instance*.

J

JAR — Java archive. In computing software, a JAR file (or Java ARchive) aggregates many files into one. Software developers generally use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

Java-MATLAB Interface — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

JDK — The *Java Development Kit* is a free Oracle product which provides the environment required for programming in Java. The JDK™ is available for various platforms, but most notably Oracle Solaris™ and Microsoft Windows. To build components with MATLAB Builder JA, download the JDK that corresponds to the latest version of Java supported by MATLAB.

JMI Interface — see *Java-MATLAB Interface*.

JRE — *Java Run-Time Environment* is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files. It does not include the compiler, debugger, or other tools present in the JDK. The JRE™ is the smallest set of executables and files that constitute the standard Java platform.

M

Magic Square — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

MATLAB Production Server Client — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

MATLAB Production Server Configuration — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, `main_config`.

MATLAB Production Server Server Instance — A logical server configuration created using the `mps -new` command in MATLAB Production Server software.

MATLAB Production Server Software — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, Web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

Marshaling — See *Data Marshaling*.

mbuild — MATLAB Compiler command that invokes a script which compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

mcc — The MATLAB command that invokes MATLAB Compiler. It is the command-line equivalent of using the Deployment Tool GUI. See the `mcc` reference page for the complete list of options available. Each builder product has customized `mcc` options. See the respective builder documentation for details.

MCR — The *MATLAB Compiler Runtime* is an execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of

MATLAB. To deploy a component, you package the MCR along with it. Before you use the MCR on a system without MATLAB, run the *MCR Installer*.

MCR Installer — An installation program run to install the MATLAB Compiler Runtime on a development machine that does not have an installed version of MATLAB. Find out more about the MCR Installer by reading “Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)”.

MCR Singleton — See *Shared MCR Instance*.

MCR Workers — A MATLAB Compiler Runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MCR session, using the `--num-workers` options in the server configurations file, `main_config`.

Method Attribute — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

mxArray interface — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

MWArray interface — A proxy to `mxArray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxArray`. There are different implementations of the `MWArray` proxy for each application programming language.

P

Package — The act of bundling the deployed component, along with the MCR and other files, for rollout to users of the MATLAB deployment products. After running the packaging function of the Deployment Tool, the package file resides in the `distrib` subfolder. On Windows®, the package is a self-extracting executable. On platforms other than Windows, it is a `.zip` file. Use of this term is unrelated to *Java Package*.

PID File — See *Process Identification File (PID File)*.

Pool — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a *pool*, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

Process Identification File (PID File) — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

Program — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

Properties — For this definition in the context of .NET, see *Fields and Properties*.

Proxy — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxAArray`.

S

Server Instance — See MATLAB Production Server Server Instance.

Shared Library — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept in for Microsoft Windows.

Shared MCR Instance — When using MATLAB Builder NE or MATLAB Builder JA, you can create a shared MCR instance, also known as a *singleton*. For builder NE, this only applies to COM components. When you invoke MATLAB Compiler with the `-S` option through the builders (using either `mcc` or the Deployment Tool), a single MCR instance is created for each COM or Java component in an application. You reuse this instance by sharing it among all subsequent class instances within the component. Such sharing

results in more efficient memory usage and eliminates the MCR startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the MATLAB files used to build the component. MATLAB Builder NE and MATLAB Builder EX are designed to create singletons by default for .NET assemblies and COM components, respectively. For more information, see “Sharing an MCR Instance in COM or Java Applications”.

Standalone application — Programs that can be executed on their own and encapsulate a self contained set of MATLAB functionality. Standalone applications are not dependent on operating system services and can be accessed outside of a shared network environment.

State — A specific data value in a program or program variable. MATLAB functions often carry state, in the form of variable values or even the MATLAB Workspace itself. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions. When running MATLAB Production Server software, there are additional considerations for preserving state. See in the *MATLAB Production Server User’s Guide* for additional information.

Structs — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

System Compiler — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio®. Before using MATLAB Compiler, select a system compiler using the MATLAB command `mbuild -setup`.

T

Threads — In the context of MATLAB Production Server software, this term has essentially the same meaning as in any server/client software product. See *pool* for additional information on managing the number of processing threads available to a server instance.

Type-safe interface — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application. Using “Generate and Implement Type-Safe Interfaces”, for example, .NET Developers work directly

with familiar native data types. You can avoid performing tedious `MWArray` data marshaling by using type-safe interfaces.

W

WAR — Web Application ARchive. In computing, a WAR file is a JAR file used to distribute a collection of `JavaServer` pages, servlets, Java classes, XML files, tag libraries, and static Web pages (HTML and related files) that together constitute a Web application.

WCF — Windows Communication Foundation. The Windows Communication Foundation™ (or WCF) is an application programming interface in the .NET Framework for building connected, service-oriented, Web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed. Clients consume multiple services that can be consumed by multiple clients. Services are loosely coupled to each other.

Webfigure — A MathWorks representation of a MATLAB figure, rendered on the Web. Using the `WebFigures` feature, you display MATLAB figures on a Web site for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the Web, without the need to download MATLAB or other tools that can consume costly resources.

Function Reference

deploytool

Purpose Compile and package functions for external deployment

Syntax `deploytool [-win32] [[[-build] | [-project]]project_name]`

Description `deploytool` opens the MATLAB Compiler app.

`deploytool project_name` opens the MATLAB Compiler app with the project preloaded.

`deploytool -build project_name` runs the MATLAB Compiler to build the specified project. The installer is not generated.

`deploytool -package project_name` runs the MATLAB Compiler to build and package the specified project. The installer is generated.

`deploytool -win32` instructs the compiler to build a 32-bit application on a 64-bit system when the following are true:

- You use the same MATLAB installation root (*matlabroot*) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line (not a MATLAB command line).

Input Arguments

`project_name` - name of the project to be compiled

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

Purpose Stream out figure “snapshot” as byte array encoded in format specified, creating signed byte array in .png format

Syntax `output type = figToImStream ('fighandle', figure_handle, 'imageFormat', image_format, 'outputType', output_type)`

Description The `output type = figToImStream ('fighandle', figure_handle, 'imageFormat', image_format, 'outputType', output_type)` command also accepts user-defined variables for any of the input arguments, passed as a comma-separated list

The size and position of the printed output depends on the figure's `PaperPosition[mode]` properties.

Options `figToImStream('figHandle', Figure_Handle, ...)` allows you to specify the figure output to be used. The Default is the current image
`figToImStream('imageFormat', [png|jpg|bmp|gif])` allows you to specify the converted image format. Default value is png.
`figToImStream('outputType', [int8!uint8])` allows you to specify an output byte data type. int8 (signed byte) is used primarily for Java primitive byte type; Default value is int8.

Examples Convert the current figure to a signed png byte array:

```
surf(peaks)
bytes = figToImStream
```

Convert a specific figure to an unsigned bmp byte array:

```
f = figure;
surf(peaks);
bytes = figToImStream( 'figHandle', f, ...
                       'imageFormat', 'bmp', ...
                       'outputType', 'uint8' );
```

Purpose

Compile MATLAB functions for deployment

Syntax

```
mcc {-e} | {-m} [-a filename]... [-B filename[:arg]]... [-C] [-d outFolder]
[-f filename] [-g] [-I directory]... [-K] [-M string] [-N] [-o filename]
[-p path]... [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename
```

```
mcc -l [-a filename]... [-B filename[:arg]]... [-C] [-d outFolder] [-f
filename] [-g] [-I directory]... [-K] [-M string] [-N] [-o filename]
[-p path]... [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename...
```

```
mcc -c [-a filename]... [-B filename[:arg]]... [-C] [-d outFolder] [-f
filename] [-g] [-I directory]... [-K] [-M string] [-N] [-o filename]
[-p path]... [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename...
```

```
mcc -W cpplib:component_name -T link:lib [-a filename]... [-B
filename[:arg]]... [-C] [-d outFolder] [-f filename] [-g] [-I directory]...
[-K] [-M string] [-N] [-o filename] [-p path]... [-R option] [-S] [-v] [-w
option[:msg]] [-win32] [-Y filename] mfilename...
```

```
mcc -W dotnet:component_name,[className], [framework_version],
security,remote_type -T link:lib [-a filename]... [-B filename[:arg]]...
[-C] [-d outFolder] [-f filename] [-I directory]... [-K] [-M string] [-N]
[-p path]... [-R option] [-S] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename... [class{className:mfilename}]...
```

```
mcc -W excel:component_name,[className], [version] -T link:lib [-a
filename]... [-b] [-B filename[:arg]]... [-C] [-d outFolder] [-f filename]
[-I directory]... [-K] [-M string] [-N] [-p path]... [-R option] [-u] [-v]
[-w option[:msg]] [-win32] [-Y filename] mfilename...
```

```
mcc -W java:packageName,[className] [-a filename]... [-b]
[-B filename[:arg]]... [-C] [-d outFolder] [-f filename] [-I
directory]... [-K] [-M string] [-N] [-p path]... [-R option]
[-S] [-v] [-w option[:msg]] [-win32] [-Y filename] filename...
[class{className:mfilename}]...
```

```
mcc -W CTF:component_name [-a filename]... [-b] [-B filename[:arg]...]
[-d outFolder] [-f filename] [-I directory]... [-K] [-M string] [-N] [-p
path]... [-R option] [-S] [-v] [-w option[:msg]] [-win32] [-Y filename]
filename... [class{className:[mfilename]}]...
```

```
mcc -?
```

Description

`mcc -m mfilename` compiles the function into a standalone application.

This is equivalent to `-W main -T link:exe`.

`mcc -e mfilename` compiles the function into a standalone application that does not open an MS-DOS® command window.

This is equivalent to `-W WinMain -T link:exe`.

`mcc -l mfilename...` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This is equivalent to `-W lib:libname -T link:lib`.

`mcc -c mfilename...` generates C wrapper code for the listed functions.

This is equivalent to `-W lib:libname -T codegen`.

`mcc -W cpplib:component_name -T link:lib mfilename...` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

`mcc -W dotnet:component_name,className,framework_version,security,remote_type -T link:lib mfilename...` creates a .NET component from the specified files.

- *component_name* — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the .NET class to be created.
- *framework_version* — Specifies the version of the Microsoft .NET Framework you want to use to compile the component. Specify either:
 - `0.0` — Use the latest supported version on the target machine.
 - *version_major.version_minor* — Use a specific version of the framework.

Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.

- *security* — Specifies whether the component to be created is a private assembly or a shared assembly.
 - To create a private assembly, specify `Private`.
 - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *remote_type* — Specifies the remoting type of the component. Values are `remote` and `local`.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{className:mfilename...}....`. *className* specifies the name of the class to create using *mfilename*.

`mcc -W excel:component_name,className, version -T link:lib mfilename...` creates a Microsoft Excel component from the specified files.

- *component_name* — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.

- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *component_name* as the default.
- *version* — Specifies the version of the component specified as *major.minor*.
 - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
 - *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.

`mcc -W java:packageName,className mfilename...` creates a Java package from the specified files.

- *packageName* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *packageName*.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{className:mfilename...}....`. *className* specifies the name of the class to create using *mfilename*.

`mcc -W CTF:component_name` instructs the compiler to create a deployable CTF archive that is deployable in a MATLAB Production Server instance.

`mcc -?` displays help.

Tip You can issue the `mcc` command either from the MATLAB command prompt or the DOS or UNIX command line.

Options

-a Add to Archive

Add a file to the CTF archive using

```
-a filename
```

to specify a file to be directly added to the CTF archive. Multiple `-a` options are permitted. MATLAB Compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

If only a folder name is included with the `-a` option, the entire contents of that folder are added recursively to the CTF archive. For example:

```
mcc -m hello.m -a ./testdir
```

In this example, `testdir` is a folder in the current working folder. All files in `testdir`, as well as all files in subfolders of `testdir`, are added to the CTF archive, and the folder subtree in `testdir` is preserved in the CTF archive.

If a wildcard pattern is included in the file name, only the files in the folder that match the pattern are added to the CTF archive and subfolders of the given path are not processed recursively. For example:

```
mcc -m hello.m -a ./testdir/*
```

In this example, all files in `./testdir` are added to the CTF archive and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

In this example, all files with the extension `.m` under `./testdir` are added to the CTF archive and subfolders of `./testdir` are not processed recursively.

All files added to the CTF archive using `-a` (including those that match a wildcard pattern or appear under a folder specified using `-a`) that do not appear on the MATLAB path at the time of compilation causes a path entry to be added to the deployed application's run-time path

so that they appear on the path when the deployed application or component executes.

When files are included, the absolute path for the DLL and header files is changed. The files are placed in the `.\exe_mcr\` folder when the CTF file is expanded. The file is not placed in the local folder. This folder is created from the CTF file the first time the EXE file is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the `exe_mcr\exe` folder.

Caution

If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

Note Currently, `*` is the only supported wildcard.

Note If the `-a` flag is used to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

-b Generate Excel Compatible Formula Function

Generate a Visual Basic file (.bas) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function. This option requires MATLAB Builder EX.

-B Specify Bundle File

Replace the file on the `mcc` command line with the contents of the specified file. Use

```
-B filename[:<a1>,<a2>,...,<an>]
```

The bundle file `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other `-B` options. A bundle file can include replacement parameters for Compiler options that accept names and version numbers. See for a list of the bundle files included with MATLAB Compiler.

-C Do Not Embed CTF Archive by Default

Override automatically embedding the CTF archive in C/C++ and main/Winmain shared libraries and standalone binaries by default. See for more information.

-d Specified Folder for Output

Place output in a specified folder. Use

`-d outFolder`

to direct the generated files to *outFolder*.

-f Specified Options File

Override the default options file with the specified options file. Use

`-f filename`

to specify `filename` as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of MATLAB Compiler. This option is a direct pass-through to the `mbuild` script.

Note MathWorks recommends that you use `mbuild -setup`.

-g Generate Debugging Information

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MCR, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

-G Debug Only

Same as -g.

-I Add Folder to Include Path

Add a new folder path to the list of included folders. Each -I option adds a folder to the beginning of the list of paths to search. For example,

```
-I <directory1> -I <directory2>
```

sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

-K Preserve Partial Output Files

Direct `mcc` to not delete output files if the compilation ends prematurely, due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

-M Direct Pass Through

Define compile-time options. Use

```
-M string
```

to pass `string` directly to the `mbuild` script. This provides a useful mechanism for defining compile-time options, e.g.,

```
-M "-Dmacro=value".
```

Note Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

-N Clear Path

Passing `-N` effectively clears the path of all folders except the following core folders (this list is subject to change over time):

- `matlabroot\toolbox\matlab`
- `matlabroot\toolbox\local`
- `matlabroot\toolbox\compiler\deploy`

It also retains all subfolders of the above list that appear on the MATLAB path at compile time. Including `-N` on the command line lets you replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the `-N` option retains all folders that you included on the path that are not under `matlabroot\toolbox`.

-o Specify Output Name

Specify the name of the final executable (standalone applications only).
Use

`-o outputfile`

to name the final executable output of MATLAB Compiler. A suitable, possibly platform-dependent, extension is added to the specified name (e.g., `.exe` for Windows standalone applications).

-p Add Folder to Path

Use in conjunction with the required option `-N` to add specific folders (and subfolders) under `matlabroot\toolbox` to the compilation MATLAB path in an order sensitive way. Use the syntax

```
-N -p directory
```

where `directory` is the folder to be included. If `directory` is not an absolute path, it is assumed to be under the current working folder. The rules for how these folders are included follow.

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.
- If a folder is included with `-p` that is not on the original MATLAB path, that folder is not included in the compilation. (You can use `-I` to add it.)

If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the folder is added to the head of the path, as it normally would be with `-I`.

-R Run-Time

Provide MCR run-time options. Use the syntax

`-R option`

to provide one of these run-time options.

Option	Description
<code>-logfile <i>filename</i></code>	Specify a log file name.
<code>-nodisplay</code>	Suppress the MATLAB <code>nodisplay</code> run-time warning.
<code>-nojvm</code>	Do not use the Java Virtual Machine (JVM).
<code>-startmsg</code>	Customizable user message displayed at MCR initialization time. See .
<code>-completemsg</code>	Customizable user message displayed when MCR initialization is complete. See .

See for information about using `mcc -R` with initialization messages.

Note The `-R` option is available only for standalone applications. To override MCR options in the other MATLAB Compiler targets, use the `mclInitializeApplication` and `mclTerminateApplication` functions. For more information on these functions, see .

Caution

When running on Mac OS X, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

-S Create Singleton MCR

Create a singleton MCR.

The standard behavior for the MCR is that every instance of a class gets its own base workspace. In a singleton MCR, all instances of a class share the same base workspace.

-T Specify Target Stage

Specify the output target phase and type.

Use the syntax `-T target` to define the output type. Target values are as follow.

Target	Description
<code>codegen</code>	Generate a C/C++ wrapper file. The default is <code>codegen</code> .
<code>compile:exe</code>	Same as <code>codegen</code> plus compiles C/C++ files to object form suitable for linking into a standalone application.
<code>compile:lib</code>	Same as <code>codegen</code> plus compiles C/C++ files to object form suitable for linking into a shared library/DLL.
<code>link:exe</code>	Same as <code>compile:exe</code> plus links object files into a standalone application.
<code>link:lib</code>	Same as <code>compile:lib</code> plus links object files into a shared library/DLL.

-u Register COM Component for the Current User

Register COM component for the current user only on the development machine. The argument applies only for generic COM component and Microsoft Excel add-in targets only.

-v Verbose

Display the compilation steps, including:

- MATLAB Compiler version number
- The source file names as they are processed
- The names of the generated output files as they are created
- The invocation of `mbuild`

The `-v` option passes the `-v` option to `mbuild` and displays information about `mbuild`.

-w Warning Messages

Display warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings. This table lists the syntaxes.

Syntax	Description
-w list	Generate a table that maps <string> to warning message for use with <code>enable</code> , <code>disable</code> , and <code>error</code> . , lists the same information.
-w enable	Enable complete warnings.
-w disable[:<string>]	Disable specific warnings associated with <string>. , lists the <string> values. Omit the optional <string> to apply the <code>disable</code> action to all warnings.
-w enable[:<string>]	Enable specific warnings associated with <string>. , lists the <string> values. Omit the optional <string> to apply the <code>enable</code> action to all warnings.
-w error[:<string>]	Treat specific warnings associated with <string> as an error. Omit the optional <string> to apply the <code>error</code> action to all warnings.

Syntax	Description
<code>-w off[:<string>] [<filename>]</code>	Turn warnings off for specific error messages defined by <i><string></i> . You can also narrow scope by specifying warnings be turned off when generated by specific <i><filename></i> s.
<code>-w on[:<string>] [<filename>]</code>	Turn warnings on for specific error messages defined by <i><string></i> . You can also narrow scope by specifying warnings be turned on when generated by specific <i><filename></i> s.

It is also possible to turn warnings on or off in your MATLAB code.

For example, to turn warnings off for deployed applications (specified using `isdeployed`) in your `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

-win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are true:

- You have a 32-bit installation of MATLAB.
- You use the same MATLAB installation root (*matlabroot*) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line.

-Y License File

Use

-Y license.lic

to override the default license file with the specified argument.

Purpose	Build and package functions for use in external applications
Syntax	<code>libraryCompiler [-win32] [[[-build] [-project]]<i>project_name</i>]</code>
Description	<p><code>libraryCompiler</code> opens the MATLAB shared library compiler for the creation of a new compiler project</p> <p><code>libraryCompiler project_name</code> opens the MATLAB shared library compiler app with the project preloaded.</p> <p><code>libraryCompiler -build project_name</code> runs the MATLAB shared library compiler to build the specified project. The installer is not generated.</p> <p><code>libraryCompiler -package project_name</code> runs the MATLAB shared library compiler to build and package the specified project. The installer is generated.</p> <p><code>libraryCompiler -win32</code> instructs the compiler to build a 32-bit application on a 64-bit system when you use the same MATLAB installation root (<i>matlabroot</i>) for both 32-bit and 64-bit versions of MATLAB.</p>
Input Arguments	<p>project_name - name of the project to be compiled</p> <p>Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.</p>

Using MATLAB Compiler on Mac or Linux

- “Overview” on page A-2
- “Installing MATLAB® Compiler™ on Mac or Linux” on page A-3
- “Writing Applications for Mac or Linux” on page A-4
- “Building Your Application on Mac or Linux ” on page A-10
- “Testing Your Application on Mac or Linux” on page A-11
- “Set MCR Paths on Mac or Linux with Scripts” on page A-12

Overview

If you use MATLAB Compiler on Linux or Macintosh systems, use this appendix as a quick reference to common tasks.

Installing MATLAB Compiler on Mac or Linux

In this section...

“Installing MATLAB® Compiler™” on page A-3

“Selecting Your gcc Compiler” on page A-3

“Custom Configuring Your Options File” on page A-3

“Install Apple Xcode from DVD on Maci64” on page A-3

Installing MATLAB Compiler

See “Supported ANSI® C and C++ UNIX Compilers” for general installation instructions and information about supported compilers.

Selecting Your gcc Compiler

Run `mbuild -setup` to select your gcc compiler . See the “UNIX” configuration instructions for more information about running `mbuild` .

Custom Configuring Your Options File

To modify the current linker settings, or disable a particular set of warnings, locate your options file for your “UNIX Operating System”, and view instructions for “Changing the Options File”.

Install Apple Xcode from DVD on Maci64

When installing on 64-bit Macintosh systems, install the Apple Xcode from the installation DVD.

Writing Applications for Mac or Linux

In this section...
“Objective-C/C++ Applications for Apple’s Cocoa API” on page A-4
“Where’s the Example Code?” on page A-4
“Preparing Your Apple Xcode Development Environment” on page A-4
“Build and Run the Sierpinski Application” on page A-5
“Running the Sierpinski Application” on page A-7

Objective-C/C++ Applications for Apple’s Cocoa API

Apple Xcode, implemented in the Objective-C language, is used to develop applications using the Cocoa framework, the native object-oriented API for the Mac OS X operating system.

This article details how to deploy a graphical MATLAB application with Objective C and Cocoa, and then deploy it using MATLAB Compiler.

Where’s the Example Code?

You can find example Apple Xcode, header, and project files in *matlabroot/extern/examples/compiler/xcode*.

Preparing Your Apple Xcode Development Environment

To run this example, you should have prior experience with the Apple Xcode development environment and the Cocoa framework.

The example in this article is ready to build and run. However, before you build and run your own applications, you must do the following (as has been done in our example code):

- 1 Build the shared library with MATLAB Compiler using either the Deployment Tool or `mcc`.

- 2 Compile application code against the component's header file and link the application against the component library and `libmwmlmcrtrt`. See “Set MCR Paths on Mac or Linux with Scripts” on page A-12 and “Solving Problems Related to Setting MCR Paths on Mac or Linux” on page A-12 for information about MCR paths and `libmwmlmcrtrt`.
- 3 In your Apple Xcode project:
 - Specify `mcc` in the project target (Build Component Library in the example code).
 - Specify target settings in `HEADER_SEARCH_PATHS`.
 - Specify directories containing the component header.
 - Specify the path `matlabroot/extern/include`.
 - Define `MWINSTALL_ROOT`, which establishes the install route using a relative path.
 - Set `LIBRARY_SEARCH_PATHS` to any directories containing the component's shared library, as well as to the path `matlabroot/runtime/maci64`.

Build and Run the Sierpinski Application

In this example, you deploy the graphical Sierpinski function (`sierpinski.m`, located at `matlabroot/extern/examples/compiler`).

```
function [x, y] = sierpinski(iterations, draw)
% SIERPINSKI Calculate (optionally draw) the points
% in Sierpinski's triangle

% Copyright 2004 The MathWorks, Inc.

% Three points defining a nice wide triangle
points = [0.5 0.9 ; 0.1 0.1 ; 0.9 0.1];

% Select an initial point
current = rand(1, 2);

% Create a figure window
if (draw == true)
    f = figure;
    hold on;
```

```
end

% Pre-allocate space for the results, to improve performance
x = zeros(1,iterations);
y = zeros(1,iterations);

% Iterate
for i = 1:iterations

    % Select point at random
    index = floor(rand * 3) + 1;

    % Calculate midpoint between current point and random point
    current(1) = (current(1) + points(index, 1)) / 2;
    current(2) = (current(2) + points(index, 2)) / 2;

    % Plot that point
    if draw, line(current(1),current(2));, end
    x(i) = current(1);
    y(i) = current(2);

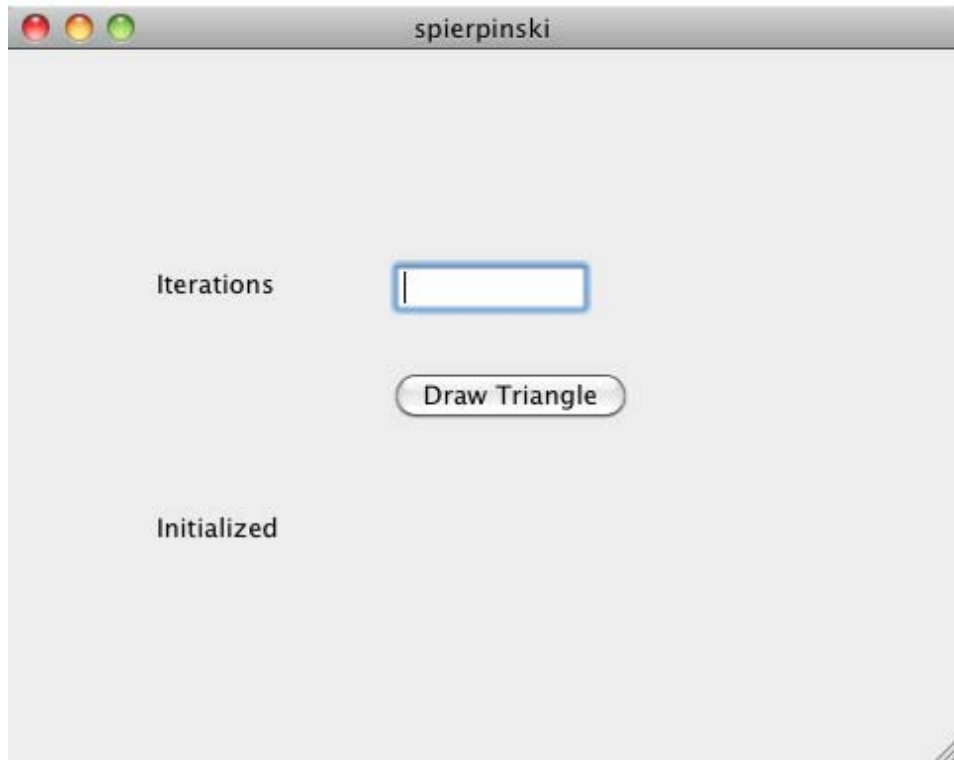
end

if (draw)
    drawnow;
end
```

- 1** Using the Mac Finder, locate the Apple Xcode project (*matlabroot/extern/examples/compiler/xcode*). Copy files to a working directory to run this example, if needed.
- 2** Open *sierpinski.xcodeproj*. The development environment starts.
- 3** In the **Groups and Files** pane, select **Targets**.
- 4** Click **Build and Run**. The make file runs that launches MATLAB Compiler (*mcc*).

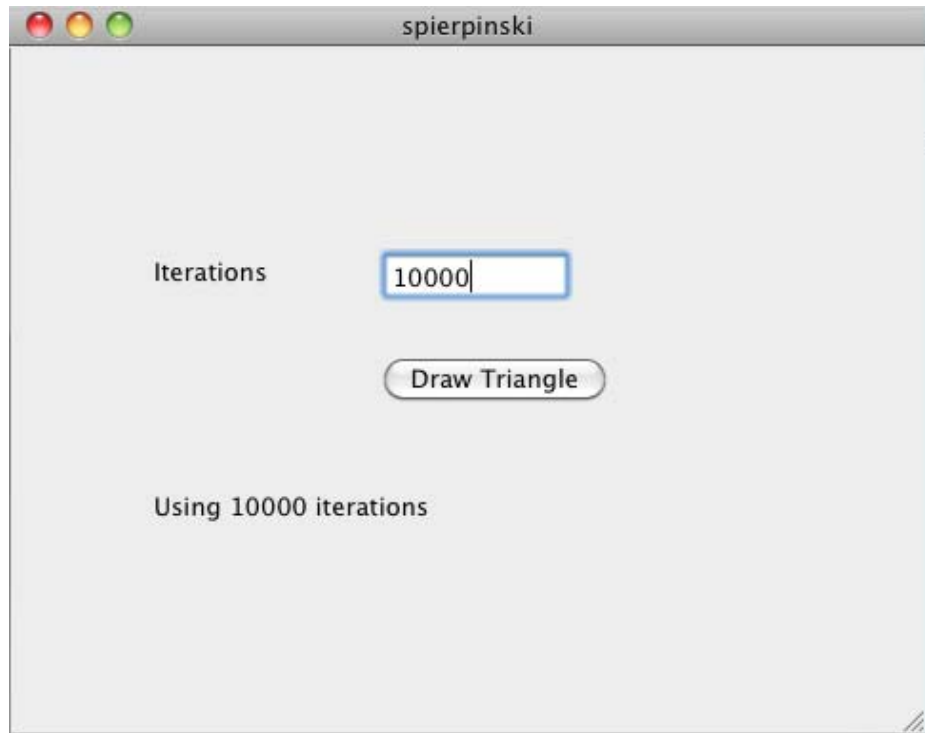
Running the Sierpinski Application

Run the **Sierpinski** application from the build output directory. The following GUI appears:

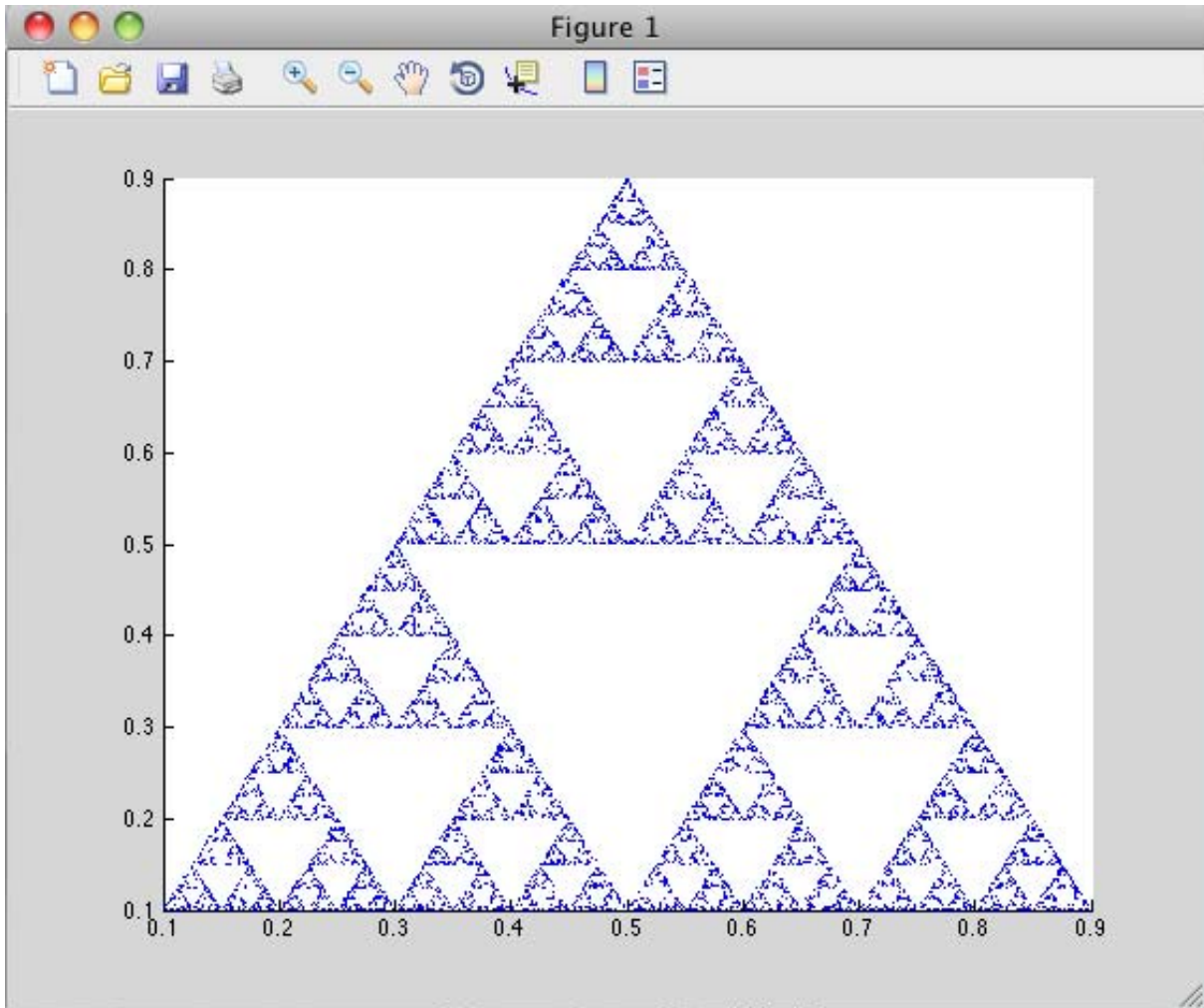


MATLAB Sierpinski Function Implemented in the Mac Cocoa Environment

1 In the **Iterations** field, enter an integer such as 10000:



2 Click **Draw Triangle**. The following figure appears:



Building Your Application on Mac or Linux

In this section...
“Compiling Your Application with the Deployment Tool” on page A-10
“Compiling Your Application with the Command Line” on page A-10

Compiling Your Application with the Deployment Tool

When running a graphical interface (such as XWindows) from your Mac or Linux desktop, use as an end-to-end template for building a standalone or shared library with the Deployment Tool (`deploytool`).

See for information on invoking `deploytool` from the command line.

Compiling Your Application with the Command Line

For compiling your application at the command line, there are separate Macintosh and non-Macintosh instructions for Mac or Linux platforms.

On Non-Mac i64 Platforms

Use the section “Input and Output Files” for lists of files produced and supplied to `mcc` when building a “Standalone Executable”, “C Shared Library”, or “C++ Shared Library”.

On Maci64

Use the section “Input and Output Files” for lists of files produced and supplied to `mcc` when building a “Macintosh 64 (Maci64)” application.

Testing Your Application on Mac or Linux

On Windows, deployed applications automatically modify the system PATH variable.

On Mac OS X or Linux, deployed applications do not modify the system PATH variable. You must perform this step manually.

Set MCR Paths on Mac or Linux with Scripts

When you build applications, associated shell scripts (*run_application.sh*) are automatically generated in the same folder as your binary. By running these scripts, you can conveniently set the path to your MCR location.

These paths can be found in the article .

Solving Problems Related to Setting MCR Paths on Mac or Linux

Use the following to solve common problems and issues:

I tried running SETENV on Mac and the command failed

If the `setenv` command fails with a message similar to `setenv: command not found` or `setenv: not found`, you are not using a C Shell command interpreter (such as `csh` or `tcsh`).

Set the environment variables using the `export` command using the format `export my_variable=my_value`.

For example, to set `DYLD_LIBRARY_PATH`, run the following command:

```
export DYLD_LIBRARY_PATH = mcr_root/v711/runtime/maci64:mcr_root/  
...
```

My Mac application fails with “Library not loaded” or “Image not found” even though my EVs are set

If you set your environment variables, you may still receive the following message when you run your application:

```
imac-joe-user:~ joeuser$ /Users/joeuser/Documents/MATLAB/Dip/Dip ; exit;  
dyld: Library not loaded: @loader_path/libmwmclmcr7.11.dylib  
Referenced from: /Users/joeuser/Documents/MATLAB/Dip/Dip  
Reason: image not found  
Trace/BPT trap  
logout
```


You may have set your environment variables initially, but they were not set up as persistent variables. Do the following:

- 1** In your home directory, open a file such as `.bashrc` or `.profile` file in your log-in shell.
- 2** In either of these types of log-in shell files, add commands to set your environment variables so that they persist. For example, to set `DYLD_LIBRARY_PATH` or `XAPPLRESDIR` in this manner, you enter the following in your file:

```
# Setting PATH for MCR

DYLD_LIBRARY_PATH=/Users/joeuser/Desktop/mcr/v711/runtime/maci64:
/Users/joeuser/Desktop/mcr/v711/sys/os/maci64:/Users/joeuser/Desktop/
mcr//v711/bin/maci64
export DYLD_LIBRARY_PATH

?
```

Note The `DYLD_LIBRARY_PATH=` statement is one statement that must be entered as a single line. The statement is shown on different lines, in this example, for readability only.

Symbols and Numerics

64-bit Mac Applications

how to run 2-25

A

Accessibility

DLLs to add to path enabling 12-2

Advanced Encryption Standard (AES)

cryptosystem 2-9

API

data conversion classes 6-13

arguments

optional 6-16

standard interface 6-20

optional inputs 6-17

optional outputs 6-19

passing 6-13

array inputs

passing 6-19

Assistive technologies

DLLs to add to path enabling 12-2

B

build process 2-5

C

calling interface

standard 12-9

calling methods 2-24

checked exceptions

exceptions

checked 6-33

in called function 6-34

in calling function 6-35

classes

API utility 6-13

calling methods of a 2-24

creating an instance of 6-9

integrating 2-23

classpath variable 1-8

command line

differences between command-line and

apps 2-5

Compiler

security 2-9

Component Technology File (CTF) 2-9

concepts

data conversion classes 2-25

converting characters to MATLAB char

array 12-5

converting data 6-14

Java to MATLAB 12-4

MATLAB to Java 12-6

converting strings to MATLAB char array 12-5

create phonebook example 7-28

create plot example 7-2

creating objects 6-9

CTF (Component Technology File) archive 2-9

CTF Archive

Controlling management and storage

of. 6-64

Embedding in component 6-64

CTF file 2-9

D

data conversion 6-14

characters, strings 12-5

example of 7-28

Java to MATLAB 12-4

MATLAB to Java 12-6

rules for Java components 12-4

unsigned integers 12-7

data conversion rules 6-55

debugging

-G option flag 13-18

Dependency Analysis Function 2-5 2-8

Deployment A-1
deploytool
 differences between command-line and 2-5
deploytool function 13-2
DLLs 2-9
 depfun 2-9

E

environment variables
 classpath 1-8
 ld_library_path 1-9
 path 1-9
error handling 6-33
example applications
 Java 7-1
examples 7-28
 Java create plot 7-2
exceptions 6-33
 catching 6-37
 checked
 in called function 6-34
 in calling function 6-35
 general 6-37
 unchecked 6-36

F

Figures
 Keeping open by blocking execution of
 console application 6-59
finalization 6-43
freeing native resources
 try-finally 6-42

G

-G option flag 13-18
garbage collection 6-40

I

images 6-56
integrating classes 2-23

J

Java application
 sample application
 usemyclass.java 6-11
 writing 7-1
Java component
 Java examples 7-1
Java packages
 instantiating classes 6-9
Java reflection 6-22
Java to MATLAB data conversion 12-4
JAVA_HOME
 Setting on Windows and UNIX 1-7
JVM 6-40

L

ld_library_path variable 1-9
Library Compiler
 differences between command-line and 2-5
limitations 12-3
 platform-specific 6-8
Limitations 2-4
Load function 3-9
loadlibrary
 (MATLAB function)
 Use of 3-7

M

-M option flag 13-23
Mac Deployment A-1
.mat file
 How to use with compiled applications 3-9
MAT file

- How to explicitly include in depfun
 - analysis 3-9
 - How to force MATLAB Compiler to inspect
 - for dependencies 3-9
 - How to use with compiled applications 3-9
 - MATLAB Builder JA
 - introduction 2-2
 - MATLAB Compiler
 - build process 2-5
 - Building on Mac or Linux A-10
 - Compiling on Mac or Linux A-10
 - Installing on Mac or Linux A-3
 - Testing on Mac or Linux A-11
 - Using with Mac and Linux A-1
 - MATLAB Compiler Runtime (MCR)
 - defined 2-21
 - MATLAB Component Runtime (MCR)
 - Administrator Privileges, requirement
 - of 2-22
 - Version Compatibility with MATLAB 2-22
 - MATLAB data files 3-9
 - MATLAB file
 - encrypting 2-9
 - MATLAB file method
 - myprimes.m 6-11
 - MATLAB objects
 - no support for 2-4
 - MATLAB to Java data conversion 12-6
 - MATLAB® Builder™ JA
 - system requirements 12-2
 - matrix math example
 - Java 7-16
 - mcc 13-4
 - differences between command-line and
 - apps 2-5
 - MCR 2-21
 - MCR Component Cache
 - How to use 6-64
 - Overriding CTF embedding 6-64
 - MCR Installer 2-21 to 2-22
 - MCR Instance
 - Sharing of 6-53
 - Sharing one 6-53
 - memory
 - preserving 6-40
 - memory management
 - native resources 6-40
 - method signatures
 - standard interface
 - method signatures 6-15 12-8
 - methods
 - adding 7-9
 - calling 2-24
 - error handling 6-33
 - of MWArray 6-16 6-55
 - MEX-files 2-5 2-8 to 2-9
 - depfun 2-9
 - MWArray 6-13
 - MWArray methods 6-16 6-55
 - MWArray query
 - return values 6-23 6-25
 - MWComponentOptions 6-64
 - mwjavaobjectref 6-27
 - MWStructArray
 - example of returning from generated
 - component 7-28
 - myprimes.m 6-11
- N**
- native resources
 - dispose 6-41
 - finalizing 6-43
- O**
- objects
 - creating 6-9
 - operating system issues 6-8
 - optional arguments 6-16

- input 6-17
- output 6-19
- standard interface 6-20

P

- Parallel Computing Toolbox
 - Example using 6-45
 - Supplying run-time profiles for 6-45
- pass through
 - M option flag 13-23
- passing arguments 6-13
- passing data
 - matlab to java 6-7
- path variable 1-9
- Platform independence
 - MEX files 6-62
- platform issues 6-8
- portability 6-8
 - ensuring for MEX files 6-62

R

- Renderers
 - in WebFigures 8-2
- requirements
 - system 12-2
- resource management 6-40
- return values
 - handling 6-21
 - java reflection 6-22
 - MWArray query 6-23 6-25

S

- Save function 3-9

- security 2-9
- shared libraries 2-9
 - depfun 2-9
- shared library 2-9
- Singleton MCR
 - Creating 6-53
- Standalone App Compiler
 - differences between command-line and 2-5
- standard interface 12-9
 - passing optional arguments 6-20
- Structs StructArrays
 - Adding fields 6-7
- system requirements 12-2
- System.exit
 - Alternatives to using 6-39
 - Reasons to avoid using 6-39

T

- try-finally 6-42

U

- unchecked exceptions 6-36
- usage information
 - getting started 1-1 2-1
 - sample Java applications 7-1

W

- waitForFigures 6-59
- Web Figure
 - WebFigure 8-2
- WebFigures
 - Supported renderers 8-2