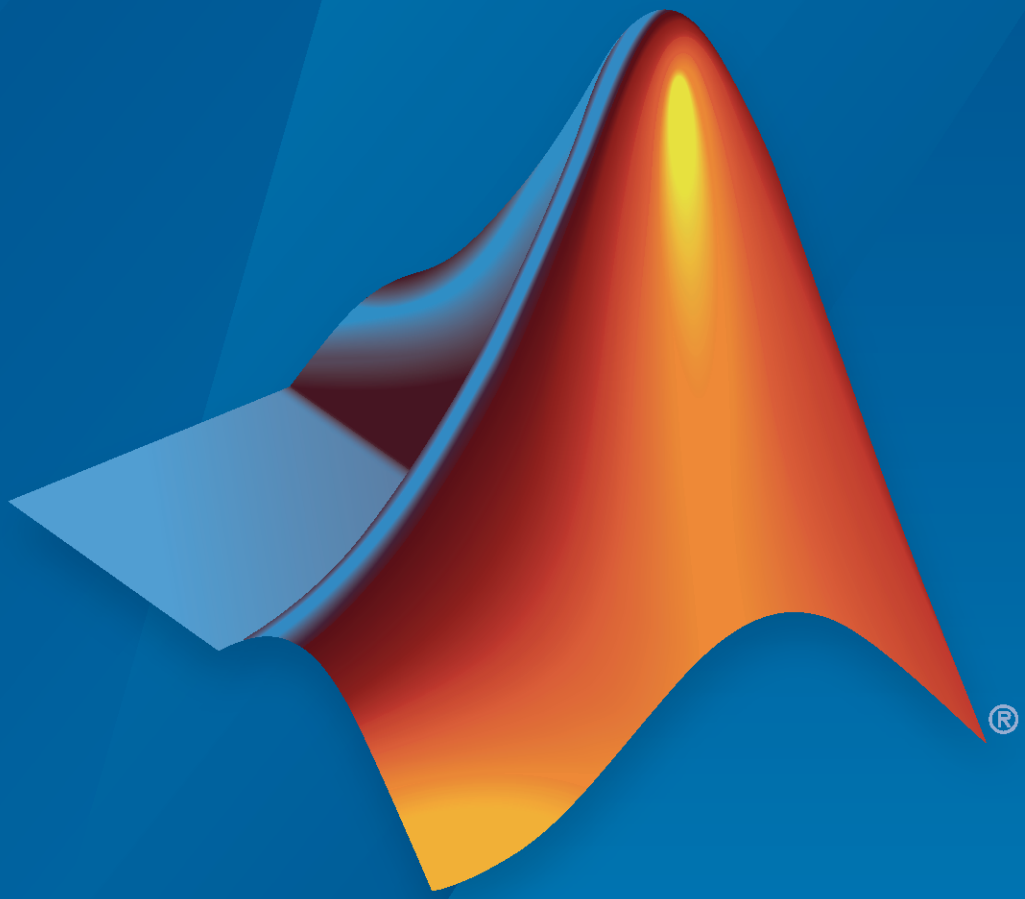
MATLAB® Compiler SDK™

Java User's Guide



MATLAB®

R2020a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

MATLAB® Compiler SDK™ Java User's Guide

© COPYRIGHT 2006–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2015	Online only	New for Version 6.0 (Release 2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)
September 2018	Online only	Revised for Version 6.6 (Release R2018b)
March 2019	Online only	Revised for Version 6.6.1 (Release R2019a)
September 2019	Online only	Revised for Version 6.7 (Release R2019b)
March 2020	Online only	Revised for Version 6.8 (Release R2020a)

Contents

1 Overview

2 Programming

3 Distribute Integrated Java Applications

4 Distribute to End Users

5 Sample Java Applications

6 Working with MATLAB Figures and Images

7 Creating Scalable Web Applications Using RMI

8 Troubleshooting

9 Reference Information for Java

10 Functions

Overview

Product Overview

In this section...

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 MATLAB Runtime must be installed, on the target machine. You must also install the compiler generated packages.

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

Limitations of Support

MATLAB Compiler SDK 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.

Configure Your Java Environment

In this section...

Before you can 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 a version of the Java Developer’s Kit (JDK™) that is compatible with MATLAB.
- `JAVA_HOME` is set to the folder containing the system’s JDK installation.
- `CLASSPATH` contains all of the MATLAB library JAR files and the JAR files 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 https://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 a Java Developer's Kit (JDK) with the same major version from https://adoptopenjdk.net/.
3. Install the JDK.

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. From the system command prompt, set the system environment variable, `JAVA_HOME`, to point to your JDK installation.

   For example on Windows® enter `set JAVA_HOME=`*`path_to_Java_install`*.
2. If you are compiling MATLAB code, verify that MATLAB is reading the correct value of `JAVA_HOME`.

   At the MATLAB command prompt, type `getenv JAVA_HOME` to display the value of `JAVA_HOME` in use by MATLAB.

3 Verify that the folder containing your Java installation has been added to your system `PATH` environment variable.

   For example on Windows enter `set PATH=%PATH%;`*`path_to_Java_install`*.

Set the CLASSPATH

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

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

Java classes compiled with MATLAB Compiler SDK 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	*`matlabroot`*`/toolbox/javabuilder/jar`
MATLAB Runtime installed on your system	*`mcrroot`*`/toolbox/javabuilder/jar`

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 MATLAB Runtime is installed.

In addition, you need to add to the JAR files created by the compiler 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 MATLAB Runtime 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.

Programming

Note For examples of these tasks, see the sample Java applications in this documentation.

For information about deploying your application after you complete these tasks, see “How Does Java Package Deployment Work?” on page 1-2.

MATLAB Compiler SDK and the JVM

Packages produced by MATLAB Compiler SDK use Java Native Interface (JNI) to interact with the MATLAB Runtime.

When the first MATLAB Compiler SDK object is instantiated:

1. Dependent MATLAB Compiler SDK classes are loaded.
2. A series of shared libraries forming the JNI bridge from the generated package to the MATLAB Runtime are loaded.
3. The MATLAB Runtime is initialized by creating an instance of a C++ class called `mcrInstance`.
4. The MATLAB-Java interface establishes a connection to the 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.
6. The MATLAB Runtime C++ core allocates resources for the arrays created by the Java API.

As you create `MWArray` objects to interact with the MATLAB Runtime, the JVM creates a wrapper object for the MATLAB `mxArray` object. The MATLAB Runtime C++ core allocates the actual resources to store the `mxArray` object. This has an impact on how the JVM frees up resources used by your application. Most of the resources used when interacting with MATLAB are created by the MATLAB Runtime C++ core. The JVM only knows about the MATLAB resources through the JNI wrappers created for them. Because of this, the JVM does not know the size of the resources being consumed and cannot effectively manage them using its built in garbage collector. The JVM also does not manage the threads used by the MATLAB Runtime and cannot clean them up.

All of the MATLAB Compiler SDK classes have static methods to properly dispose of their resources. The disposal methods trigger the freeing of the underlying resources in the MATLAB Runtime C++ core. Not properly disposing of MATLAB Compiler SDK objects can result in unpredictable behavior and may look like your application has a memory leak.

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 Runtime and generated JAR files 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 MATLAB Runtime from the web and places the generated JAR files onto your computer.
   - Manually install the MATLAB Runtime and the generated JAR files onto your development system.

     You can download the MATLAB Runtime installer from https://www.mathworks.com/products/compiler/mcr. The generated JAR files are located in the MATLAB deployment project's `for_testing` folder.
2. In the folder containing the generated JAR files, 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 MATLAB Runtime 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 Enter the `javac` command on a single line.

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

`mcrroot` is the path to where the MATLAB Runtime 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 .;"mcrroot\toolbox\javabuilder\jar\javabuilder.jar";
     .\makesqr.jar getmagic 5
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

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 Enter the `java` command on a single line.

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

`mcrroot` is the path to where the MATLAB Runtime 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.

About the MATLAB Compiler SDK Java API

In this section...

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 MATLAB Runtime required to run the MATLAB functions.

The compiler 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 Runtime 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 Compiler SDK Java 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 Compiler SDK 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 compiler converts it to an instance of the appropriate `MWArray` class before it is passed to the method. The compiler 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 “Rules for Data Conversion Between Java and MATLAB” on page 9-3 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 Cell and Struct Arrays” on page 7-10 for information and comprehensive examples.

Understanding Function Signatures Generated by the MATLAB Compiler SDK 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 compiler generates a

single overloaded Java method that accommodates any number of input arguments. This behavior is an enhancement over previous versions of `varargin` support that only handled a limited number of arguments.

Note In addition to handling optional function arguments, the overloaded Java methods that wrap MATLAB functions handle data conversion. See “Automatic Conversion to MATLAB Types” on page 2-8 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 Compiler SDK 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 compiler 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 generated for the C/C++ MATLAB Compiler SDK target.

These overloaded methods are called the standard interface and the `mlx` interface. See “Programming Interfaces Generated MATLAB Compiler SDK” on page 9-8 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.

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 Cell and Struct Arrays” on page 7-10 for information and comprehensive examples.

Pass Arguments To and From Java

In this section...

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 compiler. For class reference and usage information, see the `com.mathworks.toolbox.javabuilder` package.

Using MWNumericArray

The Magic Square example (“Integrate a Java Package into an Application” on page 2-3) 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);
```

The Java bridge 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 “Rules for Data Conversion Between Java and MATLAB” on page 9-3.

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);
```

The compiler converts the `MWNumericArray` object to a MATLAB scalar `double` to pass to the MATLAB function.

Calling MWArray Methods

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

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

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

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

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

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

Changing the Default by Specifying the Type

When calling an `MWArray` class method constructor, supplying a specific data type causes the MATLAB Compiler SDK 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
```

Specify 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)

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`.

Pass Variable Numbers of Inputs

The MATLAB Compiler SDK product generates a Java interface to this function as follows:

```
/* mlx interface - List version*/
public void mysum(List lhs, List rhs)
                          throws MWException
{
```

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

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

/* standard interface - variable inputs */
public Object[] mysum(int nargout, Object varargin)
                          throws MWException
{
   (implementation omitted)
}
```

In all cases, the `varargin` argument is passed as type `Object`. This lets you provide any number of inputs in the form of an array of Object, that is `Object[]`, and the contents of this array are passed to the compiled 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 MWException
{
   myclass cls = null;
   Object[] x = {vals};
   Object[] y = null;

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

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

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

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

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

Pass 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.

Pass 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

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)
}
```

Pass 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.

Handle 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 `to`*`Type`*`Array` methods.

Use Java Reflection to Determine Type and Dimensionality

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.

Use MWArray Query to Determine Type and Dimensionality

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[][])((MWArray)y[0]).toArray();

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

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

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

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

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

Use toTypeArray Methods to Determine Type and Dimensionality

The next example demonstrates how you can coerce or force data to a specified numeric type by invoking any of the to*Type*Array methods. These methods return an array of Java elements of the 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);
        }
```

Pass Java Objects by Reference

In this section...

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.

Wrap and Pass Objects to MATLAB

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 SDK product. The examples in this section present some common use cases.

Passing a Java Object into a MATLAB Compiler SDK Java Method

To pass an object into a MATLAB Compiler SDK Java 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 a MATLAB function that lists its methods, etc */
result = theComponent.displayObj(1, origRef);
MWArray.disposeArray(origRef);
```

Cloning an Object

You can also use `MWJavaObjectRef` to clone an object. Continuing with the example in "Passing a Java Object into a MATLAB Compiler SDK Java Method" on page 2-21, 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)

% 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 Compiler SDK Java Method” on page 2-21, 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 a 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)

% 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 wrappering), 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 “Pass Java Objects to MATLAB” on page 5-22.

Use Multiple Packages in Single Application

In this section...

When developing applications that use multiple MATLAB packages, consider that the following cannot be shared between assemblies:

- MATLAB function handles
- MATLAB figure handles
- MATLAB objects
- C, Java, and .NET objects
- Executable data stored in cell arrays and structures

Work with MATLAB Function Handles

MATLAB function handles can be passed between an application and the MATLAB Runtime instance from which it originated. However, a MATLAB function handle cannot be passed into a MATLAB Runtime instance other than the one in which it originated. For example, suppose you had two MATLAB functions, `get_plot_handle` and `plot_xy`, and `plot_xy` used the function handle created by `get_plot_handle`.

```
% Saved as get_plot_handle.m
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end

% Saved as plot_xy.m
function plot_xy(x, y, h)
h(x, y);
end
```

If you compiled them into two packages, the call to `plot_xy` would throw an exception.

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


class plottter
{
   public static void main(String[] args)
   {
     MWArray h = null;
```

```
      try
      {
        plotter_handle = new get_plot_handle.Class1();
        plot = new plot_xy.Class1();

        h = plotter_handle.get_plot_handle(1,'--rs',2.0,'k','g',10);
        double[] x = {1,2,3,4,5,6,7,8,9};
        double[] y = {2,6,12,20,30,42,56,72,90};
        plot.plot_xy(x, y, h);
      }
      catch (Exception e)
      {
         System.out.println("Exception: " + e.toString());
      }
      finally
      {
         MWArray.disposeArray(h);
         plot.dispose();
         plotter_handle.dispose();
      }
   }
}
```

The correct way to handle the situation is to compile both functions into a single package.

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

class plotter
{
   public static void main(String[] args)
   {
     MWArray h = null;

      try
      {
        plot_funcs = new Class1();

        h = plot_funcs.get_plot_handle(1, '--rs', 2.0, 'k', 'g', 10);
        double[] x = {1,2,3,4,5,6,7,8,9};
        double[] y = {2,6,12,20,30,42,56,72,90};
        plot_funcs.plot_xy(x, y, h);
      }
      catch (Exception e)
      {
         System.out.println("Exception: " + e.toString());
      }
      finally
      {
         MWArray.disposeArray(h);
         plot_funcs.dispose();
      }
   }
}
```

Note You could also handle this situation by using a singleton MATLAB Runtime.

Work with Objects

MATLAB Compiler SDK enables you to return the following types of objects from the MATLAB Runtime to your application code:

- MATLAB
- C++
- .NET
- Java

However, you cannot pass an object created in one MATLAB Runtime instance into a different MATLAB Runtime instance. This conflict can happen when a function that returns an object and a function that manipulates that object are compiled into different packages.

For example, you develop two functions. The first creates a bank account for a customer based on some set of conditions. The second transfers funds between two accounts.

```
% Saved as account.m
classdef account < handle

    properties
        name
    end

    properties (SetAccess = protected)
        balance = 0
        number
    end

    methods
        function obj = account(name)
            obj.name = name;
            obj.number = round(rand * 1000);
        end

        function deposit(obj, deposit)
            new_bal = obj.balance + deposit;
            obj.balance = new_bal;
        end

        function withdraw(obj, withdrawl)
            new_bal = obj.balance - withdrawl;
            obj.balance = new_bal;
        end

    end
end

% Saved as open_acct .m
function acct = open_acct(name, open_bal )

    acct = account(name);

    if open_bal > 0
        acct.deposit(open_bal);
    end
```

```
end

% Saved as transfer.m
function transfer(source, dest, amount)

    if (source.balance > amount)
        dest.deposit(amount);
        source.withdraw(amount);
    end

end
```

If you compiled `open_acct.m` and `transfer.m` into separate packages, you could not transfer funds using accounts created with `open_acct`. The call to `transfer` throws an exception. One way of resolving this is to compile both functions into a single package. You could also refactor the application such that you are not passing MATLAB objects to the functions.

Note You could also handle this situation by using a singleton MATLAB Runtime.

See Also

More About

- “Share MATLAB Runtime Instances” on page 9-11

Error Handling

In this section...

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.

Handling Checked Exceptions

Checked exceptions must be declared as thrown by a method using the `throws` clause. MATLAB Compiler SDK Java packages support the `com.mathworks.toolbox.javabuilder` exception.`MWException`. This exception class inherits from `java.lang.Exception` and is thrown by every MATLAB Compiler SDK 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 `MWException`s.

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 “Manage MATLAB Resources” on page 2-34.

Handling an Exception in the Calling Function

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

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

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

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

Handling Unchecked Exceptions

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

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

- `java.lang.RuntimeException`
- `java.lang.ArrayStoreException`
- `java.lang.NullPointerException`
- `java.lang.IndexOutOfBoundsException`
- `java.lang.NegativeArraySizeException`

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

Catching General Exceptions

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

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

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

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

```
   }

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

Catching Multiple Exception Types

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

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

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

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

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

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

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

Alternatives to Using of System.exit

Any Java application that uses a class generated using MATLAB Compiler SDK 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`.

Manage MATLAB Resources

In this section...

Why MATLAB Resources Need to be Managed

MATLAB Compiler SDK uses a Java Native Interface (JNI) wrapper connecting your Java application to the C++ MATLAB Runtime. As a result, most of the resources consumed by the MATLAB Compiler SDK portions of your Java application are created by the MATLAB Runtime. Resource created by the MATLAB Runtime are not visible to the JVM. The JVM’s garbage collector cannot effectively manage resources that it cannot see.

All of the MATLAB Compiler SDK Java classes have hooks that free the MATLAB resources when the JVM garbage collects the wrapper objects. However, the JVM’s garbage collection is unreliable because the JVM only sees the small wrapper object. The garbage collector can decide that it is not worth wasting CPU cycles to actually delete the wrapper object. Until the Java wrapper object is deleted, the resources allocated in the MATLAB Runtime are also not deleted. This behavior can result in conditions that look like memory leaks and rapidly consume resources.

To avoid this situation:

- Never create anonymous MATLAB objects.
- Always dispose of MATLAB objects using their `dispose()` method.

Creating MATLAB Objects

All of the MATLAB objects supported by MATLAB Compiler SDK have standard Java constructors as described in the `com.mathworks.toolbox.javabuilder` Javadoc.

When creating MATLAB objects, **always** assign them names. To create a 5x5 cell array:

```
MWCellArray myCA = new MWCellArray(5, 5);
```

The Java object `myCA` is a wrapper that points to a 5x5 `mxCellArray` object in the MATLAB Runtime. `myCA` can be added to other MATLAB arrays or manipulated in your Java application. When you are finished with `myCA`, you can clean up the 5x5 `mxCellArray` using the object’s `dispose()` method.

The semantics of the API allows you create anonymous MATLAB objects and store them in named MATLAB objects, but you should **never** do this in practice. You have **no way** to manage the MATLAB resources created by the anonymous MATLAB object.

The following code creates a MATLAB array, data, and populates it with an anonymous MATLAB object:

```
MWStructArray data = new MWStructArray(1, KMAX, FIELDS);
data.set(FIELDS[0], k + 1, new MWNumericArray(k * 1.13));
```

Two MATLAB objects are created. Both objects have a Java wrapper and a MATLAB array object in the MATLAB Runtime. When you dispose of `data`, all of the resources for it are cleaned up. However,

the anonymous object created by `new MWNumericArray(k * 1.13)` is just marked for deletion by the JVM. However, because the Java wrapper consumes a tiny amount of space, the garbage collector is likely to leave it around. Because the JVM never cleans up the wrapper object, the MATLAB Runtime never cleans up the resources it has allocated.

The MATLAB object's `set()` methods accept native Java types:

```
MWStructArray data = new MWStructArray(1, KMAX, FIELDS);
data.set(FIELDS[0], k + 1, k * 1.13);
```

In this instance, only one MATLAB object is created. When its `dispose()` method is called all of the resources are cleaned up.

Disposing of MATLAB Objects

There are two ways of cleaning up MATLAB objects:

- the object's `dispose()` method
- the static `MWArray.disposeArray()` method

Both methods release all of the resources associated with the MATLAB object. The Java wrapper object is deleted. If there are no other references to the MATLAB Runtime `mxArray` object, it is also deleted.

The following code disposes of a MATLAB object using its `dispose()` method.

```
MWCellArray myCA = new MWCellArray(5, 5);
...
myCA.dispose();
```

The following code disposes of a MATLAB object using the `MWArray.disposeArray()` method.

```
MWCellArray myCA = new MWCellArray(5, 5);
...
MWArray.disposeArray(myCA);
```

MATLAB Runtime User Data Interface

This feature provides a lightweight interface for accessing MATLAB Runtime data. It allows data to be shared between a MATLAB Runtime instance, the MATLAB code running on that MATLAB Runtime, and the wrapper code that created the MATLAB Runtime. Through calls to the MATLAB Runtime User Data interface API, you access MATLAB Runtime data through creation of a per-instance associative array of `mxArray`s, 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 on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles.
- You want to initialize the MATLAB Runtime 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 Compiler SDK software supports per-run-time instance state access through an object-oriented API. Access to a per-run-time 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 2-37.

For more information, see “Using the MATLAB Runtime User Data Interface”

Supply Run-Time Profile Information for Parallel Computing Toolbox Applications

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

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;
parpool;
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']);
delete(gcp);
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 parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Normal loop times: 0.7587,parallel loop time: 2.9988
parallel speedup:  0.253 times faster than normal
Parallel pool using the 'local' profile is shutting down.
done

a =

    0.2530
```

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 SDK.

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
    setmcruserdata('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 Library Compiler App or the Command Line Compiler

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
```

For an example on how to create a Java package using the **Library Compiler** app, see “Generate a Java Package and Build a Java Application”.

Use the following information for your project:

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

When the compilation finishes, a new folder with the same name as the project is created. This folder contains the following subfolders:

- `for_redistribution`
- `for_redistribution_files_only`
- `for_testing`

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 the Library Compiler app, click **Add files/directories** on the **Build** tab.

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

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 runtime 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 PROFILE file)
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Normal loop times: 0.7587,parallel loop time: 2.9988
parallel speedup:  0.253 times faster than normal
Parallel pool using the 'local' profile is shutting down.
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
```

See Also

More About

- “MATLAB Runtime User Data Interface” on page 2-36

Dynamically Specify Options to the MATLAB Runtime

In this section...

What Options Can You Specify?

You can pass MATLAB Runtime options `-nojvm`, `-nodisplay`, and `-logfile` to MATLAB Compiler SDK from the client application using two classes in `javabuilder.jar`:

- `MWApplication`
- `MWMCROption`

Setting and Retrieving MATLAB Runtime Option Values Using MWApplication

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

MWApplication Static Methods	Purpose
`MWApplication.initialize(MWMCROption... options);`	Passes MATLAB Runtime run-time options (see “Specifying Run-Time Options Using MWMCROption” on page 2-41)
`MWApplication.isMCRInitialized();`	Returns `true` if the MATLAB Runtime run-time is initialized; otherwise returns `false`
`MWApplication.isMCRJVMEnabled();`	Returns `true` if the MATLAB Runtime run-time is launched with JVM; otherwise returns `false`
`MWApplication.isMCRNoDisplaySet();`	Returns `true` if `MWMCROption.NODISPLAY` is used in `MWApplication.initialize` **Note** `false` is always returned on Windows systems since the `-nodisplay` option is not supported on Windows systems.
`MWApplication.getMCRLogfileName();`	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 MATLAB Runtime with the following default values.

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

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

MATLAB Runtime Run-Time Option	Default Values
`-nojvm`	`false`
`-logfile`	`null`
`-nodisplay`	`false`

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

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

MWMCROption Static Members	Purpose
`MWMCROption.NOJVM`	Launches the MATLAB Runtime without a 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 MATLAB Runtime should attach itself to the JVM launched by the client application.
`MWMCROption.NODISPLAY`	Launches the MATLAB Runtime without display functionality.
`MWMCROption.logFile("`*`logfile.dat`*`")`	Allows you to specify a log file name (must be passed with a log file name).

Passing and Retrieving MATLAB Runtime Option Values from a Java Application

Following is an example of how MATLAB Runtime 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

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

Data Conversion Between Java and MATLAB

In this section...

Overview

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

For example, consider the following Java statement:

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

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

See “Rules for Data Conversion Between Java and MATLAB” on page 9-3 for a complete list of rules to convert between Java and MATLAB Compiler SDK data types.

Call MWArray Methods

The conversion rules apply not only when calling your own methods, but also when calling constructors and factory methods belonging to the `MWArray` classes. For example, the following code calls the constructor for the `MWNumericArray` class with a Java `double` input. The MATLAB Compiler SDK 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

To specify the MATLAB to Java type conversion, you supply a specific data type in the constructor. The MATLAB Compiler SDK product converts to the specified type rather than following the default conversion rules.

The following 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);
```

Create 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`.

Set Java Properties

In this section...

How to Set Java System Properties

Set Java system properties in one of two ways:

- *In the `Java` statement*. Use the syntax: `java -D`*`propertyname`*`=`*`value`*, where *`propertyname`* is the name of the Java system property you want to set and *`value`* is the value to which you want the property set.
- *In the Java code*. Insert the following statement in your Java code near the top of the `main` function, before you initialize any Java 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 Compiler SDK 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 2-45:

- In the `java` statement:

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

- In the Java code:

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

Execution of Applications that Create Figures

The following example illustrates using `waitForFigures` from a Java application. The example uses a Java class created by the MATLAB Compiler SDK 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 compiler 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.drawplot();
           p.waitForFigures();
         }
         finally
         {
           p.dispose();
         }
       }
       catch (MWException e)
       {
         e.printStackTrace();
       }
     }
   }
   ```
5. Compile the application with the `javac` command.

   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

Compiled MATLAB code 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 MATLAB Runtime installed.

However, if your compiled MATLAB code 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 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`*`.mexw64` and then again on a Linux 64-bit machine to get *`my_mex`*`.mexa64`).
2. Compile the 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`*`.mexw64` (providing you are running on Windows). This example assumes that *`my_mex`*`.mexw64` 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 generated 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.
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.

Caution Toolbox functionality that runs seamlessly across platforms when executed from within the MATLAB desktop environment will continue to run seamlessly across platforms when deployed. However, if a particular toolbox functionality is designed to run on a specific platform, then that functionality will only run on that specific platform when deployed. For example, functionality from the Data Acquisition Toolbox™ only runs on Windows.

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

Deployable Archive Embedding and Extraction

In this section...

Overview

Deployable archive data is extracted from the JAR file with no separate deployable archive or *packageName*`mcr` folder needed on the target machine. This behavior is helpful when storage space on a file system is limited.

If you don't want deployable archive data extracted by default, use either the `MWComponentOptions` class, or use environment variables, to specify how deployable archive data extraction and utilization is handled.

Use MWComponentOptions Class to Indicate Extraction Options

Selecting Options

Choose from the following `CtfSource` or `ExtractLocation` instantiation options to customize how to manage deployable archive content with `MWComponentOptions`:

- `CtfSource` — This option specifies where the deployable archive may be found for an extracted component. It defines a binary data stream comprised of the bits of the deployable archive. The following values are objects of some type extending `MWCtfSource`:
  - `MWCtfSource.NONE` — Indicates that no deployable archive is to be extracted. This implies that the extracted deployable archive data is already accessible somewhere on your file system. This is a public, static, final instance of `MWCtfSource`.
  - `MWCtfFileSource` — Indicates that the deployable archive 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 deployable archive bits are loaded from that file. This class takes a `java.io.File` object in its constructor.
  - `MWCtfStreamSource` — Allows deployable archive 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 deployable archive content is to be located. Since the MATLAB Runtime requires all deployable archive 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 deployable archive content is to be placed in the MATLAB Runtime component cache. This is the default setting for R2007a and forward.
- `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 deployable archive data is extracted into the same folder. This option most closely matches the behavior of R2007a and previous releases.

Note Deployable archives are extracted by default to *`temp`*`\`*`user_name`*`\mcrCache`*`n.nn`*.

Setting Options

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

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

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

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

Enabling MATLAB Runtime Component Cache, Utilizing Deployable Archive Content Already on Your System

If you want to enable the MATLAB Runtime Component Cache for a generated Java class utilizing deployable archive 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);
```

Use 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 deployable archive to be extracted, this variable overrides the default per-user component cache location. This is true for embedded `.ctf` files only.	Does not apply
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 `.max_size`, which resides in the file designated by running the `mcrcachedir` command, with the desired cache size limit.

Overriding Default Behavior

To extract the deployable archive, compile using the `-C` option when calling `mcc`.

You can also implement this override by entering `-C` in the **Settings** editor of the Library Compiler app.

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

For More Information

For more information about the deployable archive, see “Deployable Archive” (MATLAB Compiler).

Explore the Javadoc

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

Distribute Integrated Java Applications

Package Java Applications

1. Gather and package the following files for installation on end user computers:
   - MATLAB Runtime installer

     See “Install and Configure the MATLAB Runtime” on page 3-4.
   - MATLAB generated Java package
   - JAR files for the application
2. Include directions for installing the MATLAB Runtime.

   See “Install and Configure the MATLAB Runtime” on page 3-4.
3. Include directions for adding the required JAR files to the Java CLASSPATH.

   At a minimum, the CLASSPATH must include:
   - *mcrroot*`/toolbox/javabuilder/jar/javabuilder.jar`
   - MATLAB generated Java package
   - JAR files for the application

About the MATLAB Runtime

In this section...

The MATLAB Runtime is a standalone set of shared libraries, MATLAB code, and other files that enables the execution of MATLAB files on computers without an installed version of MATLAB. Applications that use artifacts built with MATLAB Compiler SDK require access to an appropriate version of the MATLAB Runtime to run.

End-users of compiled artifacts without access to MATLAB must install the MATLAB Runtime on their computers or know the location of a network-installed MATLAB Runtime. The installers generated by the compiler apps may include the MATLAB Runtime installer. If you compiled your artifact using `mcc`, you should direct your end-users to download the MATLAB Runtime installer from the website `https://www.mathworks.com/products/compiler/mcr`.

See “Install and Configure the MATLAB Runtime” on page 3-4 for more information.

How is the MATLAB Runtime Different from MATLAB?

The MATLAB Runtime differs from MATLAB in several important ways:

- In the MATLAB Runtime, MATLAB files are encrypted and immutable.
- MATLAB has a desktop graphical interface. The MATLAB Runtime has all the MATLAB functionality without the graphical interface.
- The MATLAB Runtime is version-specific. You must run your applications with the version of the MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using version 6.3 (R2016b) of MATLAB Compiler, users who do not have MATLAB installed must have version 9.1 of the MATLAB Runtime installed. Use `mcrversion` to return the version number of the MATLAB Runtime.
- The MATLAB paths in a MATLAB Runtime instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Performance Considerations and the MATLAB Runtime

MATLAB Compiler SDK was designed to work with a large range of applications that use the MATLAB programming language. Because of this, run-time libraries are large.

Since the MATLAB Runtime technology provides full support for the MATLAB language, including the Java programming language, starting a compiled application takes approximately the same amount of time as starting MATLAB. The amount of resources consumed by the MATLAB Runtime is necessary in order to retain the power and functionality of a full version of MATLAB.

Calls into the MATLAB Runtime are serialized so calls into the MATLAB Runtime are threadsafe. This can impact performance.

Install and Configure the MATLAB Runtime

In this section...

Download the MATLAB Runtime Installer from the Web

Download the MATLAB® Runtime from the website at https://www.mathworks.com/products/compiler/matlab-runtime.html.

Install the MATLAB Runtime Interactively

To install the MATLAB Runtime:

1 Unzip/Extract the archive containing the MATLAB Runtime installer.

Platform	Steps
Windows	Unzip the MATLAB Runtime installer. To unzip the installer: • Right click the zip file `MATLAB_Runtime_R2020a_win64.zip` • Select **Extract All**, and then follow the instructions.
Linux	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2020a MATLAB Runtime installer, at the Terminal, type: `unzip MATLAB_Runtime_R2020a_glnxa64.zip`
macOS	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2020a MATLAB Runtime installer, at the Terminal, type: `unzip MATLAB_Runtime_R2020a_maci64.zip`

Note The release part of the installer filename (*_R2020a_*) will change from one release to the next.

2 Start the MATLAB Runtime installer.

Platform	Steps
Windows	Double-click the file `setup.exe` from the extracted files to start the installer.

Linux	At the Terminal, type:
	`sudo ./install`
	Note On Debian® based Linux distributions, you will need to type:
	`gksudo ./install`

macOS	At the Terminal, type: `./install` **Note** You may need to enter an administrator username and password after you run `./install`.

3. When the MATLAB Runtime installer starts, it displays a dialog box. Read the information and then click **Next** to proceed with the installation.
4. Specify the folder in which you want to install the MATLAB Runtime in the **Folder Selection** dialog box.

   Note On Windows systems, you can have multiple versions of the MATLAB Runtime on your computer but only one installation for any particular version. If you already have an existing installation, the MATLAB Runtime installer does not display the **Folder Selection** dialog box because you can only overwrite the existing installation in the same folder.
5. Confirm your choices and click **Next**.

   The MATLAB Runtime installer starts copying files into the installation folder.
6. On Linux and macOS platforms, after copying files to your disk, the MATLAB Runtime installer displays the **Product Configuration Notes** dialog box. This dialog box contains information necessary for setting your path environment variables. Copy the path information from this dialog box and then click **Next**.
7. Click **Finish** to exit the installer.

Install the MATLAB Runtime Non-Interactively

To install the MATLAB Runtime without having to interact with the installer dialog boxes, use one of the MATLAB Runtime installer's non-interactive modes:

- silent—the installer runs as a background task and does not display any dialog boxes
- automated—the installer displays the dialog boxes but does not wait for user interaction

When run in silent or automated mode, the MATLAB Runtime installer uses default values for installation options. You can override these defaults by using MATLAB Runtime installer command-line options or an installer control file.

Note When running in silent or automated mode, the installer overwrites the default installation location.

Running the Installer in Silent Mode

To install the MATLAB Runtime in silent mode:

1. Extract the contents of the MATLAB Runtime installer file to a temporary folder, called `$temp` in this documentation.

   Note On Windows systems, **manually** extract the contents of the installer file.
2. Run the MATLAB Runtime installer, specifying the `-mode` option and `-agreeToLicense yes` on the command line.

   Note On most platforms, the installer is located at the root of the folder into which the archive was extracted. On Windows 64, the installer is located in the archives `bin` folder.

Platform	Command
Windows	`setup -mode silent -agreeToLicense yes`
Linux	`./install -mode silent -agreeToLicense yes`
macOS	`./install -mode silent -agreeToLicense yes`

   Note If you do not include the `-agreeToLicense yes` the installer will not install the MATLAB Runtime.
3. View a log of the installation.

   On Windows systems, the MATLAB Runtime installer creates a log file, named `mathworks_`*`username`*`.log`, where *`username`* is your Windows log-in name, in the location defined by your `TEMP` environment variable.
4. On Linux and macOS systems, specify the path variable. The MATLAB Runtime installer displays the log information for Linux and macOS systems at the command prompt, unless you redirect it to a file.

Customizing a Non-Interactive Installation

When run in one of the non-interactive modes, the installer will use the default values unless told to do otherwise. Like the MATLAB installer, the MATLAB Runtime installer accepts a number of command line options that modify the default installation properties.

Option	Description
`-destinationFolder`	Specifies where the MATLAB Runtime will be installed.
`-outputFile`	Specifies where the installation log file is written.
`-automatedModeTimeout`	Specifies how long, in milliseconds, that the dialog boxes are displayed when run in automatic mode.

Option	Description
`-inputFile`	Specifies an installer control file with the values for all of the above options.

Note The MATLAB Runtime installer archive includes an example installer control file called `installer_input.txt`. This file contains all of the options available for a full MATLAB installation. Only the options listed in this section are valid for the MATLAB Runtime installer.

Install the MATLAB Runtime without Administrator Rights

To install the MATLAB Runtime as a user without administrator rights on Windows:

1. Use the MATLAB Runtime installer to install it on a Windows machine where you have administrator rights.
2. Copy the folder where the MATLAB Runtime was installed to the machine without administrator rights. You can compress the folder into zip file and distribute to multiple users.
3. On the machine without administrator rights, add the *`mcr_root`*`\runtime\`*`arch`* directory onto the user's path environment variable.

   Note You don't need administrator rights for adding directories to a user's path environment variable.

Multiple MATLAB Runtime Versions on Single Machine

`MCRInstaller` supports the installation of multiple versions of the MATLAB Runtime on a target machine. This allows applications compiled with different versions of the MATLAB Runtime to execute side by side on the same machine.

If you do not want multiple MATLAB Runtime versions on the target machine, you can remove the unwanted ones. On Windows, run **Add or Remove Programs** from the Control Panel to remove any of the previous versions. On Linux, you manually delete the unwanted MATLAB Runtime. You can remove unwanted versions before or after installation of a more recent version of the MATLAB Runtime, as versions can be installed or removed in any order.

MATLAB and MATLAB Runtime on Same Machine

You do not need to install MATLAB Runtime on your machine if your machine has MATLAB installed. The version of MATLAB should be the same as the version of MATLAB that was used to create the compiled MATLAB code. Also, to act as the MATLAB Runtime replacement, the MATLAB installation must include MATLAB Compiler.

You can, however, install the MATLAB Runtime for debugging purposes.

Modifying the Path

If you install MATLAB Runtime on a machine that already has MATLAB on it, you must adjust the library path according to your needs.

- **Windows**

To run deployed MATLAB code against MATLAB Runtime install, *mcr_root*\ver\runtime\win64 must appear on your system path before *matlabroot*\runtime\win64.

If *mcr_root*\ver\runtime*arch* appears first on the compiled application path, the application uses the files in the MATLAB Runtime install area.

If *matlabroot*\runtime*arch* appears first on the compiled application path, the application uses the files in the MATLAB installation area.

- **Linux**

  To run deployed MATLAB code against MATLAB Runtime on Linux, the folder <mcr_root>/runtime/<arch> must appear on your LD_LIBRARY_PATH before *matlabroot*/runtime/<arch>.

- **macOS**

  To run deployed MATLAB code on macOS, the <mcr_root>/runtime folder must appear on your DYLD_LIBRARY_PATH before *matlabroot*/runtime/<arch>.

  To run MATLAB on macOS or Intel® Mac, *matlabroot*/runtime/<arch> must appear on your DYLD_LIBRARY_PATH before the <mcr_root>/bin folder.

Uninstall MATLAB Runtime

The method you use to uninstall MATLAB Runtime from your computer varies depending on the type of computer.

Windows

1. Start the uninstaller.

   From the Windows Start menu, search for the **Add or Remove Programs** control panel, and double-click MATLAB Runtime in the list.

   You can also start the MATLAB Runtime uninstaller from the *mcr_root*\uninstall\bin*arch* folder, where *mcr_root* is your MATLAB Runtime installation folder and *arch* is an architecture-specific folder, such as win64.
2. Select the MATLAB Runtime from the list of products in the Uninstall Products dialog box.
3. Click **Next**.
4. Click **Finish**.

Linux

1. Exit the application.
2. Enter this command at the Linux prompt:

   ```
   rm -rf mcr_root
   ```

   where *mcr_root* represents the name of your top-level MATLAB installation folder.

macOS

1. Exit the application.
2. Navigate to your MATLAB Runtime installation folder. For example, the installation folder might be named MATLAB_Compiler_Runtime.app in your Applications folder.

3 Drag your MATLAB Runtime installation folder to the trash, and then select **Empty Trash** from the Finder menu.

Distribute to End Users

MATLAB Runtime Path Settings for Development and Testing

In this section...

Path for Java Development on All Platforms

There are additional requirements when programming in the Java programming language. For more information see “Configure Your Java Environment” on page 1-3.

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:

```
matlabroot\sys\java\jre\arch\jre\bin\JavaAccessBridge.dll
matlabroot\sys\java\jre\arch\jre\bin\WindowsAccessBridge.dll
```

You may not be able to use such technologies without doing so.

Windows Settings for Development and Testing

When programming with compiled MATLAB code, add the following folder to your system `PATH` environment variable:

```
matlabroot\runtime\win32|win64
```

Linux Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

```
setenv LD_LIBRARY_PATH
   matlabroot/runtime/glnxa64:
   matlabroot/bin/glnxa64:
   matlabroot/sys/os/glnxa64:
   matlabroot/sys/opengl/lib/glnxa64
```

OS X Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

```
setenv DYLD_LIBRARY_PATH
   matlabroot/runtime/maci64:
   matlabroot/bin/maci64:
   matlabroot/sys/os/maci64:
```

MATLAB Runtime Path Settings for Run-Time Deployment

In this section...

General Path Guidelines

Regardless of platform, be aware of the following guidelines with regards to placing specific folders on the path:

- Always avoid including `arch` on the path. Failure to do so may inhibit ability to run multiple MATLAB Runtime instances.
- Ideally, set the environment in a separate shell script to avoid run-time errors caused by path-related issues.

Path for Java Applications on All Platforms

When your users run applications that contain compiled MATLAB code, you must instruct them to set the path so that the system can find the MATLAB Runtime.

Note When you deploy a Java application to end users, they must set the class path on the target machine.

The system needs to find `.jar` files containing the MATLAB libraries. To tell the system how to locate the `.jar` files it needs, specify a `classpath` either in the `javac` command or in your system environment variables.

Windows Path for Run-Time Deployment

The following folder should be added to the system path:

*mcr_root**version*\runtime\win64

mcr_root refers to the complete path where the MATLAB Runtime library archive files are installed on the machine where the application is to be run.

mcr_root is version specific; you must determine the path after you install the MATLAB Runtime.

Note If you are running the MATLAB Runtime installer on a shared folder, be aware that other users of the share may need to alter their system configuration.

Linux Paths for Run-Time Deployment

Use these `setenv` commands to set your MATLAB Runtime paths.

```
setenv LD_LIBRARY_PATH
   mcr_root/version/runtime/glnxa64:
   mcr_root/version/bin/glnxa64:
   mcr_root/version/sys/os/glnxa64:
   mcr_root/version/sys/opengl/lib/glnxa64
```

OS X Paths for Run-Time Deployment

Use these `setenv` commands to set your MATLAB Runtime paths.

```
setenv DYLD_LIBRARY_PATH
   mcr_root/version/runtime/maci64:
   mcr_root/version/bin/maci64:
   mcr_root/version/sys/os/maci64
```

Sample Java Applications

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

Display a MATLAB Plot in a Java Application

In this section...

Purpose

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

- Use the MATLAB Compiler SDK 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 “Configure Your Java Environment” on page 1-3.
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 Library Compiler app:

   ```
   libraryCompiler
   ```
5. You create a Java package by using the Library Compiler app to build a Java class that wraps around your MATLAB code. For an example, see “Generate a Java Package and Build a Java Application”.

   Use the following information for your project:

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

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

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

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

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

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

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

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

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

   The program does the following:

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

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

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

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

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

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

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 major version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “Limitations of the MATLAB Compiler SDK Java Target” on page 9-2 for more specific information.

The `createplot` program should display the output.

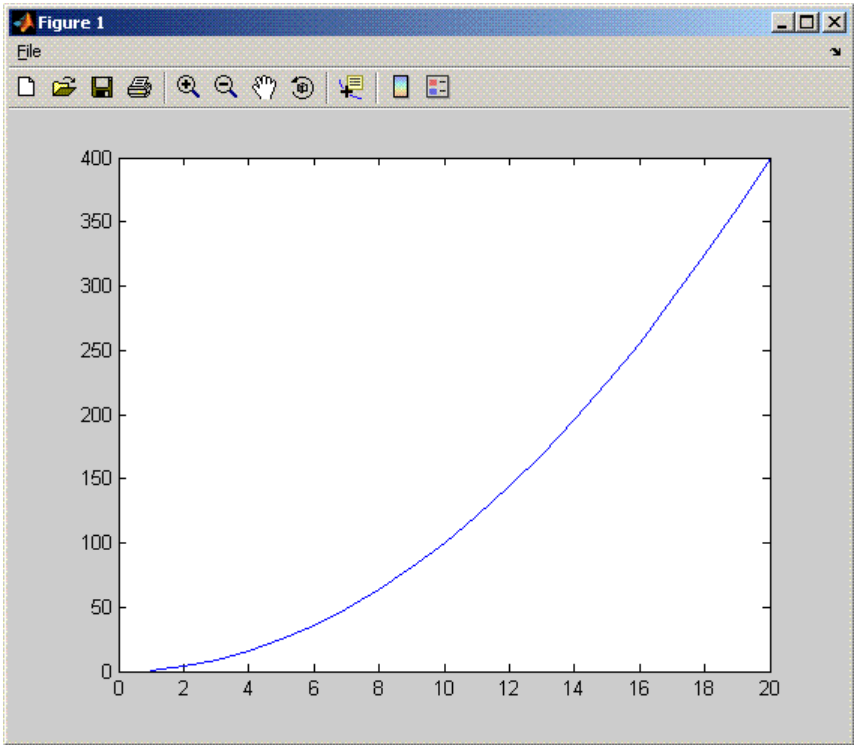


Figure 1
File
400
350
300
250
200
150
100
50
0
0
2
4
6
8
10
12
14
16
18
20

Create a Java Application with Two MATLAB Functions

In this section...

Purpose

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

- How to use the MATLAB Compiler SDK product to create a package containing a class that has a private method that is automatically encapsulated.
- How to access the MATLAB functions in a Java application, including use of the `MWArray` class hierarchy to represent data

  Note For complete reference information about the `MWArray` class hierarchy, see the `com.mathworks.toolbox.javabuilder` package.
- How to build and run the application

The `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 “Configure Your Java Environment” on page 1-3.
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. Open the Library Compiler app.
5. Create the Java package by using the Library Compiler app to build a Java class that wraps your MATLAB code. For an example on how to create a Java package using the **Library Compiler** app, see “Generate a Java Package and Build a Java Application”.

   Use the following information for your project:

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

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

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

      try
      {
         /*
          * Construct input data as sin(2*PI*15*t) +
          * sin(2*PI*40*t) plus a random signal.
          *    Duration = 10
          *    Sampling interval = 0.01
          */
         int[] dims = {1, nSamples};
         data = MWNumericArray.newInstance(dims, MWClassID.DOUBLE,
                                           MWComplexity.REAL);
         for (int i = 1; i <= nSamples; i++)
         {
            double t = (i-1)*interval;
            double x = Math.sin(2.0*Math.PI*15.0*t) +
               Math.sin(2.0*Math.PI*40.0*t) +
               Math.random();
            data.set(i, x);
         }

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

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

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

      finally
      {
```

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

The program does the following:

- Constructs an input array with values representing a random signal with two sinusoids at 15 and 40 Hz embedded inside of it
- Creates an `MWNumericArray` array that contains the data, as shown:

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

- Instantiates a `fourier` object
- Calls the `plotfft` method, which calls `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 root MATLAB 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:

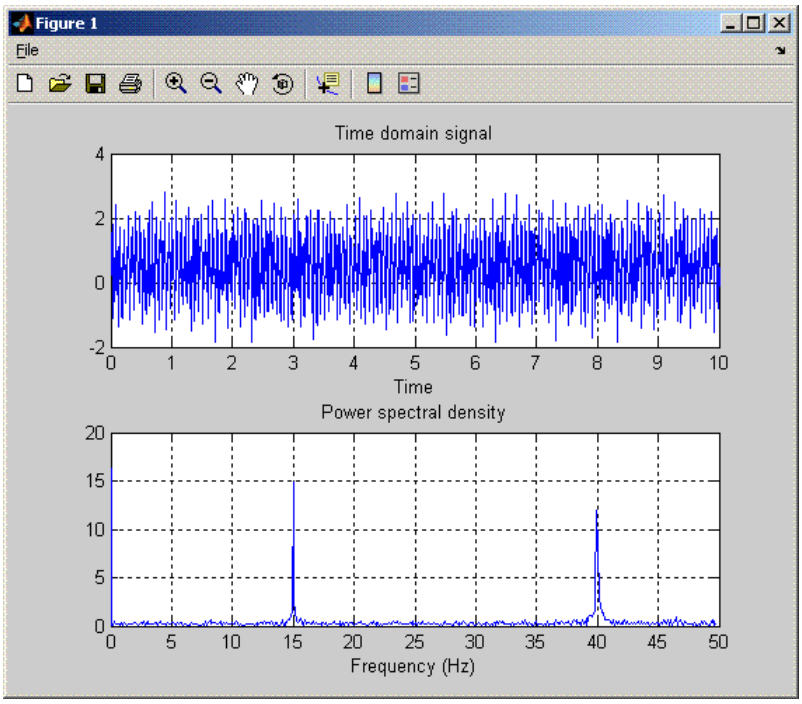
  ```
  java -classpath
  .:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
  ./distrib/spectralanalysis.jar
  powerspect
  ```

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

```
version -java
```

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “Limitations of the MATLAB Compiler SDK Java Target” on page 9-2 for more specific information.

The `powerspect` program should display the output:



Matrix Math

In this section...

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

You supply the size of the matrix on the command line, and the program constructs the matrix and performs the three factorizations. The original matrix and the results are printed to standard output. You may optionally perform the calculations using a sparse matrix by specifying the string "sparse" as the second parameter on the command line.

MATLAB Functions to Be Encapsulated

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

- `cholesky.m`

  ```
  function [L] = cholesky(A)

  L = chol(A);
  ```

- `ludecomp.m`

  ```
  function [L,U] = ludecomp(A)

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

- `qrdecomp.m`

```
function [Q,R] = qrdecomp(A)

[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 “Configure Your Java Environment” on page 1-3.

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

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

4 Select Library Compiler app.

5 Create the Java package using the Library Compiler app to build a Java class that wraps around your MATLAB code. For an example, see “Generate a Java Package and Build a Java Application”.

Use the following information for your project:

Project Name	`factormatrix`
Class Name	`factor`
Files to compile	`cholesky.m`, `ludecomp.m`, and `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

```
/* 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 major version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Note If you are running on theMac 64-bit platform, you must add the `-d64` flag in the Java command. See “Limitations of the MATLAB Compiler SDK Java Target” on page 9-2 for more specific information.

Output for the Matrix Math Example

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


LU factorization:
L matrix:
    1.0000         0         0         0
   -0.5000    1.0000         0         0
         0   -0.6667    1.0000         0
         0         0   -0.7500    1.0000


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


QR factorization:
Q matrix:
   -0.8944   -0.3586   -0.1952    0.1826
```

```
    0.4472   -0.7171   -0.3904    0.3651
         0    0.5976   -0.5855    0.5477
         0         0    0.6831    0.7303


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

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

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

Output for a Sparse Matrix

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


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


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


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

```
   (3,4)      -1.0000
   (4,4)       1.2500


QR factorization:
Q matrix:
   (1,1)       0.8944
   (2,1)      -0.4472
   (1,2)       0.3586
   (2,2)       0.7171
   (3,2)      -0.5976
   (1,3)       0.1952
   (2,3)       0.3904
   (3,3)       0.5855
   (4,3)      -0.6831
   (1,4)       0.1826
   (2,4)       0.3651
   (3,4)       0.5477
   (4,4)       0.7303


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

Phone Book

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 “Configure Your Java Environment” on page 1-3.
3. Write the `makephone` function as you would any MATLAB function.

   The following code defines the `makephone` function:

   ```
   function book = makephone(friends)

   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. Select Library Compiler app.
5. Create a Java Package using the Library Compiler app to build a Java class that wraps around your MATLAB code. For an example, see “Generate a Java Package and Build a Java Application”.

   Use the following information for your project:

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

```
/* 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 major version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

```
version -java
```

Note If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “Limitations of the MATLAB Compiler SDK Java Target” on page 9-2 for more specific information.

The `getphone` program should display the output:

```
Friends:
2x2 struct array with fields:
    name
    phone
Result:
2x2 struct array with fields:
    name
    phone
    external
Result record 2:
Mary Smith
3912
(508) 555-3912

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

```
Element 3
   name: Stacy Flora
   phone: 3238
   external: (508) 555-3238
Element 4
   name: Harry Alpert
   phone: 3077
   external: (508) 555-3077
```

Pass Java Objects to MATLAB

In this section...

Purpose

The purpose of this example is to show you how to:

- Use the MATLAB Compiler SDK product to create a package that applies MATLAB optimization routines to objective functions implemented as Java objects.
- Access the MATLAB functions in a Java application , including use of the `MWJavaObjectRef` class to create a reference to a Java object 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 MATLAB `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.
- The 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`.

Prerequisites

- Start this example by creating a new work folder that is visible to the MATLAB search path. In this example, we will use the following folders:

  Windows: `c:\matlab\work`

  Linux: `~/matlab/work`

- If you have not already done so, set the environment variables that are required on a development machine. For more information, see “Configure Your Java Environment” on page 1-3.
- Copy the following files to the `work` folder:

File(s)	From	To
`javabuilder.jar`	*`matlabroot`*`\toolbox\javabuilder\jar\win64`	**Windows:** `c:\matlab\work` **Linux:** `~/matlab/work`

`doOptim.m`
 	
`displayObj.m`
	`matlabroot``\toolbox\javabuilder\Examples\ObjectRefExample\ObjectRefDemoComp`	**Windows:** `c:\matlab\work`
		
		Linux: `~/matlab/work`

`BananaFunction.java`
 	
`PerformOptim.java`
	`matlabroot``\toolbox\javabuilder\Examples\ObjectRefExample\ObjectRefDemoJavaApp`	**Windows:** `c:\matlab\work`
		
		Linux: `~/matlab/work`

Your `work` folder should now have the following five files:

```
c:\matlab\work

javabuilder.jar
doOptim.m
displayObj.m
BananaFunction.java
PerformOptim.java
```

Procedure

1. Open MATLAB and `cd` to the `work` folder you created in the prerequisite section.
2. Write the MATLAB code you want to access from Java. This example uses `doOptim.m` and `displayObj.m`, which are already in your `work` folder.

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

   ```
   function [x,fval] = doOptim(h, x0)
   directEval = h.evaluateFunction(x0)
   wrapperEval = mWrapper(x0)
   [x,fval] = fminsearch(mWrapper,x0)
   ```

   For reference, the code from `displayObj.m` is displayed here:

   ```
   function className = displayObj(h)
   h
   className = class(h)
   whos('h')
   methods(h)
   ```
3. At the MATLAB command prompt, type `libraryCompiler` to open the **Library Compiler** app.
4. Use the **Library Compiler** app to create a Java package from the MATLAB functions `doOptim.m` and `displayObj.m`. Use the following naming conventions for your package:

Save the project in your `work` folder using the name:	`OptimDemo`
Library Name:	`OptimDemo`
Class Name:	`Optimizer`
Exported Functions (files to compile) :	`doOptim.m` and `displayObj.m`

The Java classes created by the **Library Compiler** app serve as wrapper around the MATLAB code.

The **Library Compiler** app will create a folder by the name `OptimDemo` in the `c:\matlab\work` directory. The `OptimDemo` folder contains the following sub folders:

```
for_redistribution
for_redistribution_files_only
for_testing
```

The files that you will need for the rest of the example are located in the `for_testing` folder.

For more information on working with the **Library Compiler** app, see “Generate a Java Package and Build a Java Application”.

5 Write source code for a class that implements an object function to optimize. The code for this example is in file `BananaFunction.java`. The program listing is shown here:

```
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 MATLAB `fminsearch` documentation.

6 Write source code for an application that accesses the MATLAB functions. The code for this example is in the file `PerformOptim.java`. The program listing is shown here:

```
/* 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 Java 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 accomplishes 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.

7 Compile the `PerformOptim.java` application and `BananaFunction.java` helper class using the Java command `javac`. When entering this command, ensure there are no spaces between path names separated by a semicolon (`;`).

1 Open a Command Prompt or Terminal window and `cd` to the `work` folder.

2 Compile the application according to which operating system you are running on:

Windows

To compile `BananaFunction.java`, type:

```
javac -classpath .;
c:\matlab\work\javabuilder.jar;
c:\matlab\work\OptimDemo\for_testing\OptimDemo.jar BananaFunction.java
```

To compile `PerformOptim.java`, type:

```
javac -classpath .;
c:\matlab\work\javabuilder.jar;
c:\matlab\work\OptimDemo\for_testing\OptimDemo.jar PerformOptim.java
```

Linux

To compile `BananaFunction.java`, type:

```
javac -classpath .:
~/matlab/work/javabuilder.jar:
~/matlab/work/OptimDemo/for_testing/OptimDemo.jar BananaFunction.java
```

To compile `PerformOptim.java`, type:

```
javac -classpath .:
~/matlab/work/javabuilder.jar:
~/matlab/work/OptimDemo/for_testing/OptimDemo.jar PerformOptim.java
```

8 Execute the `PerformOptim` class file as follows:

On Windows, type:

```
java -classpath .;
c:\matlab\work\javabuilder.jar;
c:\matlab\work\OptimDemo\for_testing\OptimDemo.jar PerformOptim -1.2 1.0
```

On Linux, type:

```
java -classpath .:
~/matlab/work/javabuilder.jar:
~/matlab/work/OptimDemo/for_testing/OptimDemo.jar PerformOptim -1.2 1.0
```

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

```
version -java
```

> **Note** If you are running on the Mac 64-bit platform, you must add the `-d64` flag in the Java command. See “Limitations of the MATLAB Compiler SDK Java Target” on page 9-2 for more specific information.

When run successfully, the `PerformOptim` program should display the following output:

```
Using x0 =
-1.2000    1.0000
*****************************************************
** Properties of Java object                       **
*****************************************************

h =

BananaFunction@1766806

className =

BananaFunction

  Name      Size            Bytes  Class             Attributes

  h         1x1                    BananaFunction

Methods for class BananaFunction:

 BananaFunction    getClass          notifyAll
equals            hashCode          toString
evaluateFunction  notify            wait

** Finished DISPLAYOBJ ******************************
*****************************************************
** Performing unconstrained nonlinear optimization **
*****************************************************

directEval =

   24.2000

 wrapperEval =

   24.2000

 x =

    1.0000    1.0000

 fval =

   8.1777e-10

Optimization successful
** Finished DOOPTIM *********************************
Location of minimum:
1.0000    1.0000
Function value at minimum:
```

```
8.1777e-10
```

Display a MATLAB Plot on the Web using a Java Servlet

In this section...

Overview

This example demonstrates how to display a plot created by a Java servlet calling a class created with the MATLAB Compiler SDK 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 “Specify Optional Arguments” on page 2-13.

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 5-30
- “Ensure Your Web Server Is Java Compliant” on page 5-30
- “Install the javabuilder.jar Library” on page 5-31

Ensure You Have the Required Products

The following products must be installed at their recommended release levels.

- MATLAB, MATLAB Compiler, MATLAB Compiler SDK
- 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 major version of Java that ships with MATLAB. To find out what version of Java MATLAB is running, enter the following MATLAB command:

  ```
  version -java
  ```

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.

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.

Locating and Copying the Example Files

The example files are located in the *`matlabroot`*`\toolbox\javabuilder\Examples\java_web_vararg_demo` folder.

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 5-32.

Copy the Example Files

Copy the folder `java_web_vararg_demo` containing example files to a local folder. Failure to do so may lead to errors.

Build Your Java Package

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

1. Open the Library Compiler app.
2. Create the Java package using the Library Compiler app to build a Java class that wraps around your MATLAB code. For an example on how to create a Java package using the **Library Compiler** app, see "Generate a Java Package and Build a Java Application"

   Use the following information for your project:

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` — included with MATLAB Compiler SDK
- `vararg_java.jar` — the JAR file you just built
- `servlet-api.jar` — included as part of the servlet container

For more details about using `javac`, see the Oracle website.

Generating the Web Archive (WAR) File

Web archive or WAR files are a type of Java Archive used to deploy J2EE and JSP servlets. To run this example you will need to use the `jar` command to generate the final WAR file that runs the application. To do this, follow these steps:

1. Copy the JAR file created using the MATLAB Compiler SDK product into the `JavaCode\build\WEB-INF\classes\VarArg` example folder.
2. Copy the compiled Java class to the `JavaCode\build\WEB-INF\classes\VarArg` example folder.
3. 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.

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

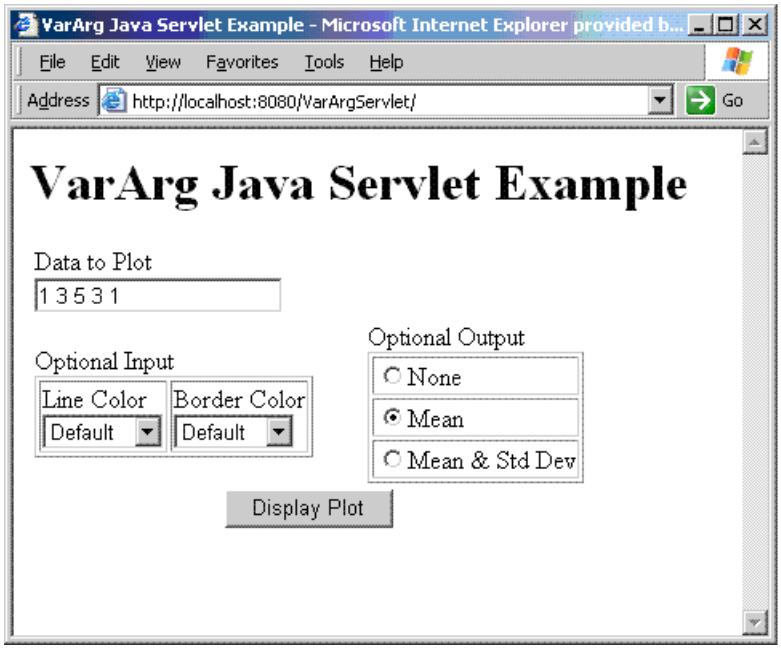
Running the Web Deployment Example

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

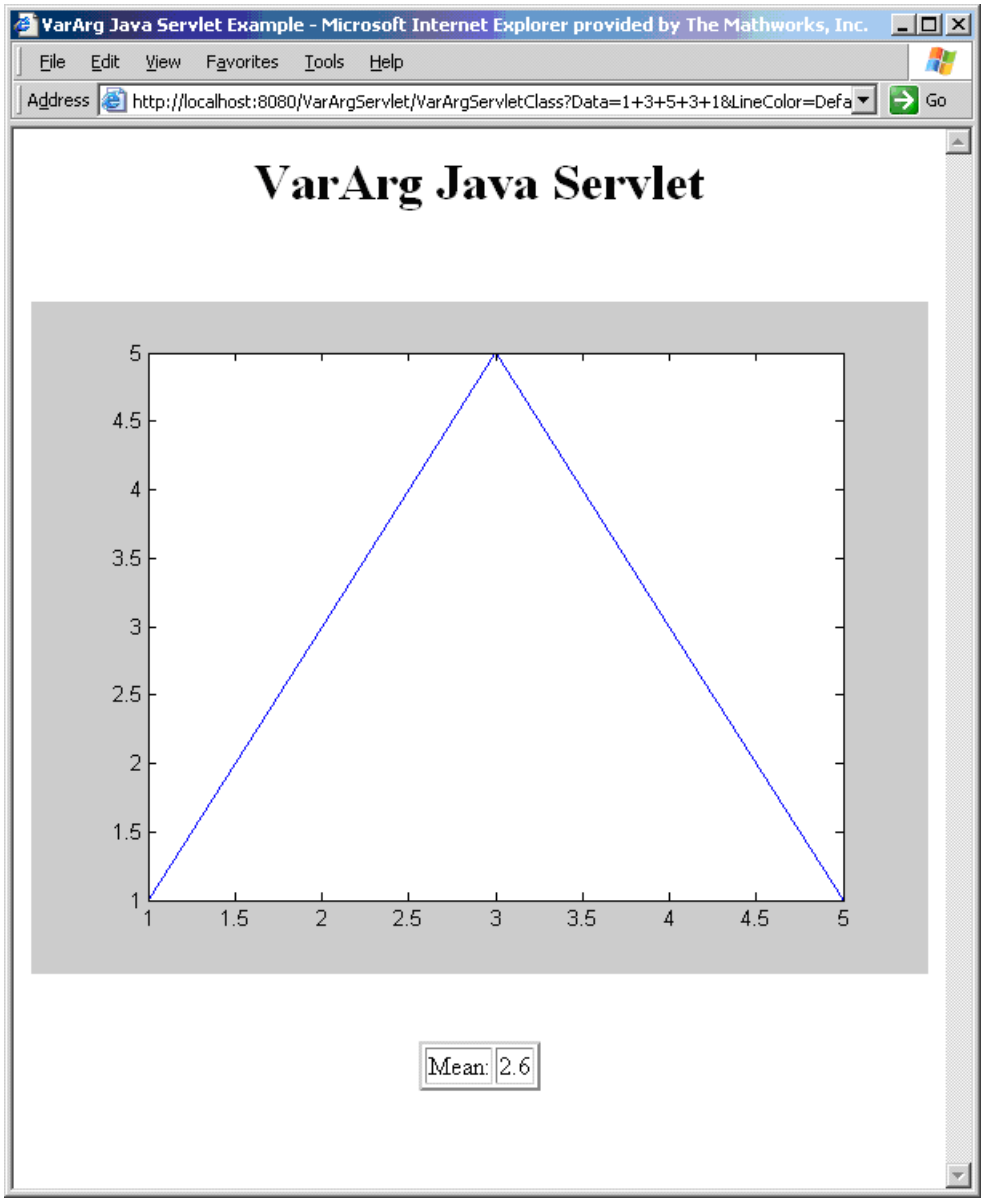
1. Install the `VarArgServlet.war` file into your Web server's `webapps` folder.
2. Run the application by entering `http://localhost:`*`port_number`*`/VarArgServlet` in the address field of your Web browser, where *`port_number`* is the port that your Web server is configured to use (usually `8080`).

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.


VarArg Java Servlet Example - Microsoft Internet Explorer provided by The Mathworks, Inc.
File Edit View Favorites Tools Help
Address http://localhost:8080/VarArgServlet/VarArgServletClass?Data=1+3+5+3+1&LineColor=Defa
Go
VarArg Java Servlet
5
4.5
4
3.5
3
2.5
2
1.5
1
1 1.5 2 2.5 3 3.5 4 4.5 5
Mean: 2.6

6

Working with MATLAB Figures and Images

Roles 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

Work with MATLAB Image Data

In this section...

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 *“Use MATLAB Compiler SDK Web Example Guide”*.

Working with Images

Getting Encoded Image Bytes from an Image in a Component

```
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.getByteData();
    }
    finally
    {
        MWArray.disposeArray(byteImageOutput);
    }
}
```

Getting a Buffered Image in a Component

```
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.getByteData();
    }
    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

Use Remote Method Invocation (RMI)

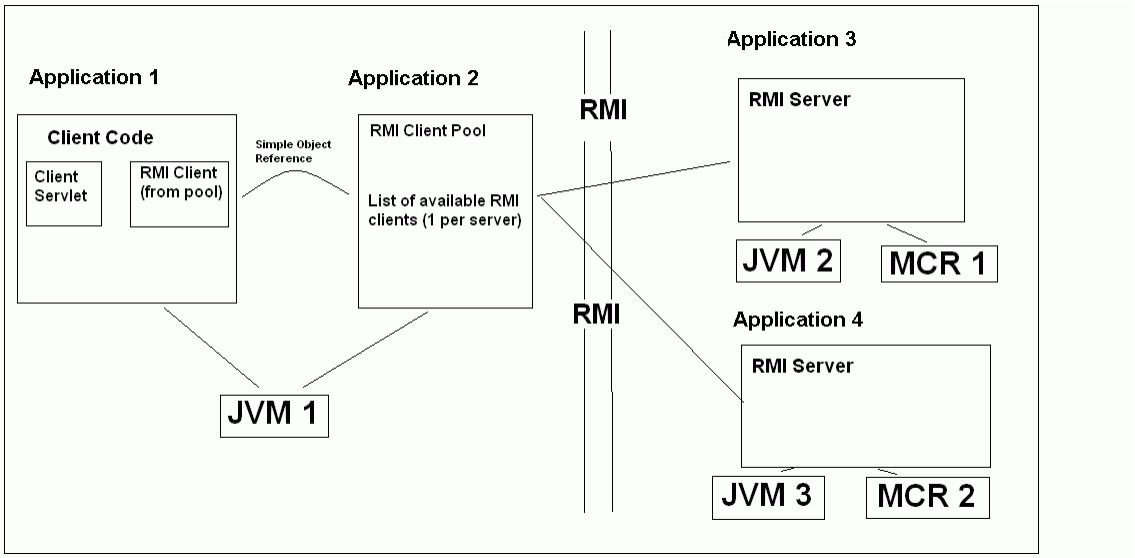
You can expand your application's throughput capacity by taking advantage of RMI, the Java native remote procedure call (RPC) mechanism. The way the MATLAB Compiler SDK product implements RMI technology provides for automatic generation of interface code to enable components to start in separate processes, on one or more computers, making your applications scalable and adaptable to future performance demands.

You can use RMI in the following ways:

- Run a client and server on a single machine
- Run 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

This section describes the prerequisites needed prior to using RMI.

Ensure You Have the Required Products

The following products must be installed at their recommended release levels.

- MATLAB, MATLAB Compiler, MATLAB Compiler SDK
- Java Development Kit (JDK)

  Ensure you have a JDK installed on your system. You can download OpenJDK from https://adoptopenjdk.net/.

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

```
version -java
```

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.

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.

Run Client and Server on Same Machine

The following example shows how to run two separate processes to initialize MATLAB struct arrays.

Note You do not need the MATLAB Runtime installed on the client side. Return values from the MATLAB Runtime 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 Java package by issuing the following command at the MATLAB command prompt:

   ```
   mcc -W 'java:dataTypesComp,dataTypesClass' createEmptyStruct.m updateField.m -v
   ```

   createEmptyStruct.m

   ```
   function PartialStruct = createEmptyStruct(field_names)

   fprintf('EVENT 1: Initializing the structure in MATLAB and sending it to JAVA client:\n');

   PartialStruct = struct();

   for i=1:length(field_names)
       PartialStruct.(field_names{i}) = [];
   end

   fprintf('         Initialized empty structure:\n\n');
   disp(PartialStruct);
   fprintf('\n###################################\n');
   ```

   updateField.m

   ```
   function FinalStruct = updateField(st,field_name)

   fprintf('\nEVENT 3: Partially initialized structure as received by MATLAB:\n\n');
   disp(st);
   fprintf('Address field as initialized from the client:\n\n');
   disp(st.Address);
   fprintf('################################\n');

   fprintf(['\nEVENT 4: Updating ''', field_name, ''' field before sending the structure back to the JAVA client:\n\n']);
   st.(field_name) = 'MathWorks';
   FinalStruct = st;
   disp(FinalStruct);
   fprintf('\n###################################\n');
   ```

2. Compile the server Java code by issuing the following `javac` command at your system command prompt. Ensure there are no spaces between `javabuilder.jar;` and *`directory_containing_package`*.

   `javac -classpath` *`matlabroot`*`\toolbox\javabuilder\jar\javabuilder.jar; directory_containing_pac`

   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 at your system command prompt. Ensure there are no spaces between `javabuilder.jar;` and *`directory_containing_package`*.

   `javac -classpath` *`matlabroot`*`\toolbox\javabuilder\jar\javabuilder.jar; directory_containing_pac`

4. Run the client and server as follows:

   a. Open two command windows.

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 at the system command prompt. Ensure there are no spaces between dataTypesComp.jar; and *matlabroot*.

```
java -classpath .;directory_containing_package\dataTypesComp.jar; matlabroot\toolbox\java
```

d Run the client by issuing the following *java* command at the system command prompt. Ensure there are no spaces between dataTypesComp.jar; and *matlabroot*.

```
java -classpath .;directory_containing_package\dataTypesComp.jar; matlabroot\toolbox\java
```

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 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 Client and Server on Same Machine” on page 7-4.

Why Use Native Cell Arrays and Struct Arrays?

In Java, there is no direct representation available for MATLAB struct arrays 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 Compiler SDK packages 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) subtype 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 MATLAB Runtime on your client machines

Using Native Types Does Not Require a Client-Side MATLAB Runtime

The classes in the `com.mathworks.extern.java` package (in `javabuilder.jar`) do not need a MATLAB Runtime. 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 a MATLAB Runtime.

The availability of a MATLAB Runtime 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 a MATLAB Runtime. If the client machine does not have a MATLAB Runtime 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 Data Marshaling Prerequisites

Even though client machines don’t need to have a MATLAB Runtime, 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 Compiler SDK Java packages.

Use Native Java Cell and Struct Arrays

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 Client and Server on Separate Machines” on page 7-7 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 MATLAB Runtime. The client does not need access to the MATLAB Runtime.
- 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 “MATLAB Runtime Path Settings for Run-Time Deployment” on page 4-4 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_package\dataTypesComp.jar NativeCellStructServer.java
```

3. Compile the client's Java code:

```
javac -classpath matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
directory_containing_package\dataTypesComp.jar
NativeCellStructClient.java
```

4. Prepare to run the server and client applications by opening two command windows—one for client and one for server.
5. Run the server:

```
java -classpath .;directory_containing_package\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 Compiler SDK Error Messages

Exception in thread "main" java.lang.UnsatisfiedLinkError: Failed to find the library mclmcrrt712.dll, required by MATLAB Compiler SDK, on java.library.path

Install the MATLAB Runtime or add it to the MATLAB path.

Failed to find the library <library_name>, required by MATLAB Compiler SDK, on java.library.path.

This error commonly occurs on Linux or Mac systems if the `LD_LIBRARY_PATH` variable is not set.

See “MATLAB Runtime Path Settings for Development and Testing” on page 4-2 and “MATLAB Runtime Path Settings for Run-Time Deployment” on page 4-4.

`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.

Java packages generated using the LibraryCompiler app that serialize and deserialize MathWorks Java classes will throw an exception or hang when using serialization filtering in Java 8.

MathWorks® Java classes need to be on the filter pattern list of the serialization filtering feature of Java 8 so that they can be passed to the method `java.io.ObjectInputStream.filterCheck()`. This will prevent an application using the Java package from thowing an exception or from hanging. To fix the issue, set the following system properties at the command line:

```
jdk.serialFilter=com.mathworks.**
sun.rmi.registry.registryFilter=com.mathworks.**
```

Reference Information for Java

Requirements for the MATLAB Compiler SDK Java Target

In this section...

System Requirements

System requirements and restrictions on use of the MATLAB Compiler SDK Java target are as follows:

- The MATLAB Compiler product must be installed.
- Java Development Kit must be installed.
- Java Runtime Environment that is used by MATLAB and MATLAB Runtime

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

```
version -java
```

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:

matlabroot\sys\java\jre*arch*\jre\bin\JavaAccessBridge.dll
matlabroot\sys\java\jre*arch*\jre\bin\WindowsAccessBridge.dll

You may not be able to use such technologies without doing so.

Limitations of the MATLAB Compiler SDK Java Target

MATLAB Java External Interface

JAR files created by MATLAB Compiler SDK cannot be loaded back into MATLAB with the MATLAB Java External Interface.

MATLAB Objects

The MATLAB Compiler SDK product's Java target does not support MATLAB object data types. MATLAB objects cannot "pass" the MATLAB/Java boundary. However, you are free to use objects in your MATLAB code.

Rules for Data Conversion Between Java and MATLAB

In this section...

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 compiler 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`
`boolean`	`logical`
`java.lang.Double`	`double`
`java.lang.Float`	`single`
`java.lang.Byte`	`int8`
`java.lang.Integer`	`int32`
`java.lang.Long`	`int64`
`java.lang.Short`	`int16`
`java.lang.Number`	`double` **Note** Subclasses of `java.lang.Number` not listed above are converted to `double`.
`java.lang.Boolean`	`logical`
`java.lang.Character`	`char`
`java.lang.String`	`char` **Note** A Java string is converted to a `1`-by-`N` array of `char` with `N` equal to the length of the input string. An array of Java strings (`String[]`) is converted to an `M`-by-`N` array of `char`, with `M` equal to the number of elements in the input array and `N` equal to the maximum length of any of the strings in the array. Higher dimensional arrays of `String` are converted similarly. In general, an `N`-dimensional array of `String` is converted to an `N+1` dimensional array of `char` with appropriate zero padding where supplied strings have different lengths.

MATLAB to Java Conversion

The following table lists the data conversion rules for converting MATLAB data types to Java types.

Note The conversion rules apply to scalars, vectors, matrices, and multidimensional arrays of the types listed.

MATLAB to Java Conversion Rules

MATLAB Type	Java Type (Primitive)	Java Type (Object)
`cell`	Not applicable	`Object` **Note** Cell arrays are constructed and accessed as arrays of `MWArray`.
`structure`	Not applicable	`Object` **Note** Structure arrays are constructed and accessed as arrays of `MWArray`.
`char`	`char`	`java.lang.Character`
`double`	`double`	`java.lang.Double`
`single`	`float`	`java.lang.Float`
`int8`	`byte`	`java.lang.Byte`
`int16`	`short`	`java.lang.Short`
`int32`	`int`	`java.lang.Integer`
`int64`	`long`	`java.lang.Long`
`uint8`	`byte`	`java.lang.Byte` Java 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.short` Java 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.Integer` Java 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.Long` Java 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	

MATLAB Type	Java Type (Primitive)	Java Type (Object)
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 MATLAB Compiler SDK

In this section...

APIs Based on MATLAB Function Signatures

The compiler 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	`function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)`
API if there are no input arguments	`public Object[] foo(int numArgsOut)`
API if there is one input argument	`public Object[] foo(int numArgsOut, Object In1)`

API if there are two to *N* input arguments	`public Object[] foo(`
	`int numArgsOut,`
	`Object In1,`
	`Object In2,`
	`... Object InN`
	`)`

API if there are optional arguments, represented by the `varargin` argument	`public Object[] foo(`
	`int numArgsOut,`
	` Object in1,`
	` Object in2,`
	` ..., Object InN,`
	`Object varargin`
	`)`

Details about the arguments for these samples of standard signatures are shown in the following table.

Argument	Description	Details About Argument
numArgsOut	Number of outputs	An integer indicating the number of outputs you want the method to return. To return no arguments, omit this argument. The value of *numArgsOut* must be less than or equal to the MATLAB function `nargout`. The *numArgsOut* argument must always be the first argument in the list.
In1, In2, ...InN	Required input arguments	All arguments that follow *numArgsOut* in the argument list are inputs to the method being called. Specify all required inputs first. Each required input must be of class `MWArray` or any class derived from `MWArray`.
varargin	Optional inputs	You can also specify optional inputs if your MATLAB code uses the `varargin` input: list the optional inputs, or put them in an `Object[]` argument, placing the array last in the argument list.
Out1, Out2, ...OutN	Output arguments	With the standard calling interface, all output arguments are returned as an array of `MWArray`s.

mlx API

For a function with the following structure:

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

The compiler 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 compiler generates three overloaded methods in Java.

When you add `myprimes` to the class `myclass` and build the class, the compiler 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 9-8 for details about the interfaces.

Share MATLAB Runtime Instances

In this section...

What Is a Singleton MATLAB Runtime?

You create an instance of the MATLAB Runtime that can be shared among all subsequent class instances within a component. This is commonly called a shared MATLAB Runtime instance or a Singleton runtime.

Advantages and Disadvantages of Using a Singleton

In most cases, a singleton MATLAB Runtime will provide many more advantages than disadvantages. Following are examples of when you might and might not create a shared MATLAB Runtime 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 MATLAB Runtime start-up or initialization time

When You Might Avoid Using a Singleton

Using a singleton may not benefit you if your application uses a large number of global variables. This causes crosstalk.

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.

10

Functions

mcrinstaller

Display version and location information for MATLAB Runtime installer corresponding to current platform

Syntax

```
[INSTALLER_PATH, MAJOR, MINOR, PLATFORM] = mcrinstaller;
```

Description

Displays information about available MATLAB Runtime installers using the format: `[INSTALLER_PATH, MAJOR, MINOR, PLATFORM] = mcrinstaller;` where:

- *`INSTALLER_PATH`* is the full path to the installer for the current platform.
- *`MAJOR`* is the major version number of the installer.
- *`MINOR`* is the minor version number of the installer.
- *`PLATFORM`* is the name of the current platform (returned by `COMPUTER(`*`arch`*`)`).

If no MATLAB Runtime installer is found, you are prompted to download an installer using the command `compiler.runtime.download`.

Note You must distribute the MATLAB Runtime library to your end users to enable them to run applications developed with MATLAB Compiler or MATLAB Compiler SDK.

See “Install and Configure the MATLAB Runtime”for more information about the MATLAB Runtime installer.

Examples

Find MATLAB Runtime Installer Location

Display the location of MATLAB Runtime installers for a particular platform. This example shows output for a `win64` system. The release number is called `R20xxx` indicating the release for which the MATLAB Runtime installer has been downloaded.

```
mcrinstaller
```

```
C:\Program Files\MATLAB\R20xxx\toolbox\compiler\deploy\win64\MCR_R20xxx_win64_installer.exe
```

For example, for R2018b, the path would be:

```
C:\Program Files\MATLAB\R2018b\toolbox\compiler\deploy\win64\MCR_R2018b_win64_installer.exe
```

Introduced in R2009a

mcrversion

Determine version of installed MATLAB Runtime

Syntax

```
[major, minor] = mcrversion;
```

Description

The MATLAB Runtime version number consists of two digits, separated by a decimal point. This function returns each digit as a separate output variable: `[major, minor] = mcrversion;` Major and minor are returned as integers.

If the version number ever increases to three or more digits, call `mcrversion` with more outputs, as follows:

```
[major, minor, point] = mcrversion;
```

At this time, all outputs past “minor” are returned as zeros.

Typing only `mcrversion` will return the major version number only.

Examples

```
mcrversion
ans =
     7
```

Introduced in R2008a

waitForFigures

Block execution of a calling program as long as figures created in encapsulated MATLAB code are displayed

Syntax

```
objName.waitForFigures();
```

Description

`waitForFigures()` blocks 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 Compiler SDK 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.

See Also

Topics

“Execution of Applications that Create Figures” on page 2-46

Introduced before R2006a