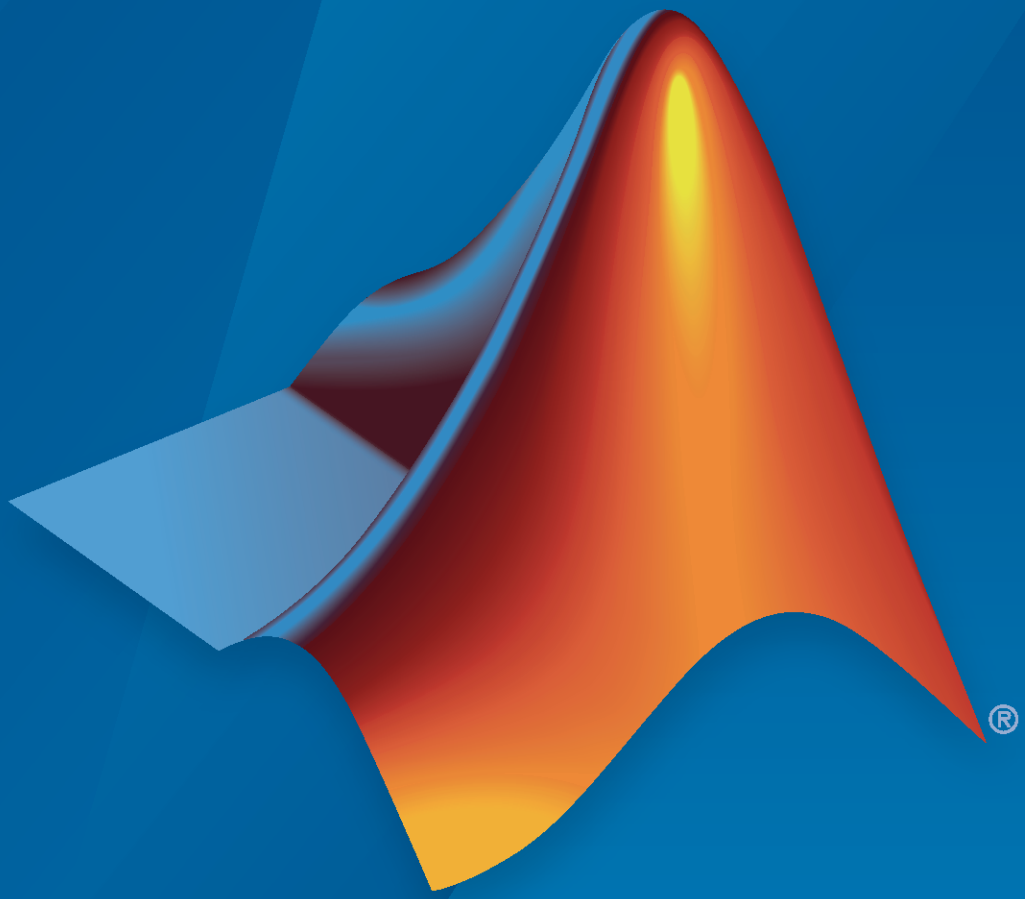
MATLAB® Compiler SDK™

Java User's Guide



MATLAB®

R2021b

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

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)
September 2020	Online only	Revised for Version 6.9 (Release R2020b)
March 2021	Online only	Revised for Version 6.10 (Release R2021a)
September 2021	Online only	Revised for Version 6.11 (Release R2021b)

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

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 MATLAB Runtime must be installed on the target machine. You must also install or distribute 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.

Configure Your Java Environment for Generating Packages

In this section...

Before you can generate Java packages using MATLAB Compiler SDK or run Java applications, ensure that your Java environment is properly configured. You must verify that:

- Your development system uses a version of the Java Developer’s Kit (JDK™) that is compatible with MATLAB. For updated Java system requirements, see MATLAB Interfaces to Other Languages.
- The `JAVA_HOME` environment variable is set to the folder containing your Java installation.
- `CLASSPATH` contains all of the MATLAB library JAR files and the JAR files for the packages containing your compiled MATLAB code.
- Your target machine has MATLAB or MATLAB Runtime installed. For instructions on how to install MATLAB Runtime, see “Install and Configure MATLAB Runtime”.

Install JDK or JRE

To develop Java applications, you must install the proper version of the Java Developer's Kit (JDK). If you are not compiling MATLAB code or developing Java applications, you can install the Java Runtime Environment (JRE™) instead of the JDK to run Java applications.

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

   ```
   version -java
   ```
2. Download and install the JDK with the same major version from https://adoptopenjdk.net/. For example, if `version -java` returns `Java 1.8.X`, install OpenJDK 8.

   In Windows®, you can automatically set the `JAVA_HOME` environment variable during installation by selecting the **Set JAVA_HOME variable** option on the **Custom Setup** screen.

Set JAVA_HOME Environment Variable

After you install the JDK or the JRE, set the system environment variable `JAVA_HOME` to your Java installation folder if you have not already done so during installation.

1. Use the following table to set `JAVA_HOME` according to your operating system.

Operating System	Procedure
Windows	1 Run `C:\Windows\System32\SystemPropertiesAdvanced.exe` and click the **Environment Variables...** button. 2 Select the system variable `JAVA_HOME` and click **Edit...**. If you do not have administrator rights on the machine, select the user variable `JAVA_HOME` instead of the system variable. 3 Click **New** and add the path to your Java installation folder. For example, `C:\Program Files\AdoptOpenJDK\jdk-8.0.282.8-hotspot`. 4 Click **OK** to apply the change.

Linux®	In a Bash shell, enter the following commands:
	`echo "export JAVA_HOME=<path_to_Java_install>" >> ~/.bashrc`
	`source ~/.bashrc`

macOS (Mojave 10.14 or Earlier)	In a Bash shell, enter the following commands:
	`echo "export JAVA_HOME=<path_to_Java_install>" >> ~/.profile`
	`source ~/.profile`

macOS (Catalina 10.15 or Later)	In a Zsh shell, enter the following commands:
	`echo "setenv JAVA_HOME <path_to_Java_install>" >> ~/.zshrc`
	`source ~/.zshrc`

2 If you are compiling MATLAB code, verify that MATLAB reads the correct value of `JAVA_HOME`.

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

Set CLASSPATH

To build and run a Java application that uses a component generated by MATLAB Compiler SDK, the class path must include:

- Classes in the `com.mathworks.toolbox.javabuilder` package, which is located in *`matlabroot`*`/toolbox/javabuilder/jar/javabuilder.jar`, where *`matlabroot`* represents your MATLAB or MATLAB Runtime installation folder.
- Java packages that you have developed.

When you compile a Java application, you must specify a `classpath` either in the `javac` command or in the `CLASSPATH` system environment variable. Similarly, when you deploy a Java application, the end user must specify a `classpath` either in the `java` command or in the `CLASSPATH` system

environment variable. For an example on setting the class path, see “Compile and Run MATLAB Generated Java Application”.

Set Shared Library Path Variable

Add the `bin` subfolder of your Java installation to your shared library path environment variable.

Use the following table to set the library path according to your operating system.

Operating System	Procedure
Windows	The OpenJDK installer for Windows automatically sets the library path during installation. If you do not use the installer, complete the following steps to set the `PATH` environment variable permanently. **1** Run `C:\Windows\System32\SystemPropertiesAdvanced.exe` and click the **Environment Variables...** button. **2** Select the system variable `Path` and click **Edit...**. If you do not have administrator rights on the machine, select the user variable `Path` instead of the system variable. **3** Click **New** and add the path to the folder `<path_to_Java_install>\bin`. **4** Click **OK** to apply the change.
Linux	In a Bash shell, enter the following command: `export JAVA_HOME=<path_to_Java_install>/bin`
macOS (Mojave 10.14 or Earlier)	In a Bash shell, enter the following command: `export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:<path_to_Java_install>/bin`
macOS (Catalina 10.15 or Later)	In a Zsh shell, enter the following command: `setenv DYLD_LIBRARY_PATH $DYLD_LIBRARY_PATH:<path_to_Java_install>/bin`

Note In order to run Java applications that contain compiled MATLAB code, you must include the MATLAB or MATLAB Runtime library folders in your system library path. For details, see “Set MATLAB Runtime Path for Deployment”.

See Also

Related Examples

- “Install and Configure MATLAB Runtime”
- “Set MATLAB Runtime Path for Deployment”
- “Generate Java Package and Build Java Application”

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.

Integrate Simple MATLAB Function Into Java Application

This example shows how to invoke a MATLAB method that generates a magic square in a Java application.

Files

MATLAB Function Location	*matlabroot*\toolbox\javabuilder\Examples\MagicSquareExample\MagicDemoComp\makesqr.m
Java Code Location	*matlabroot*\toolbox\javabuilder\Examples\MagicSquareExample\MagicDemoJavaApp\getmagic.java

Procedure

1. Copy the `MagicSquareExample` folder that ships with MATLAB to your work folder:

   ```
   copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','MagicSquareExample'))
   ```

   At the MATLAB command prompt, navigate to the new `MagicSquareExample\MagicDemoComp` subfolder in your work folder.
2. Examine the `makesqr.m` function.

   ```
   function y = makesqr(x)
   y = magic(x);
   disp(y);
   ```

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

   The output is a 5-by-5 matrix.

   ```
       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
   ```
3. Create a Java package that encapsulates `makesqr.m` by using the **Library Compiler** app or `compiler.build.javaPackage`.

   Use the following information for your project:

Package Name	`magicsquare`
Class Name	`magic`
File to Compile	`makesqr.m`

   For example, if you are using `compiler.build.javaPackage`, type:

   ```
   buildResults = compiler.build.javaPackage('makesqr.m', ...
       'PackageName','magicsquare', ...
       'ClassName','magic');
   ```

   For more details, see the instructions in “Generate Java Package and Build Java Application”.
4. Write source code for a Java application that accesses the MATLAB function.

   The sample application for this example is in `MagicSquareExample\MagicDemoJavaApp\getmagic.java`.

getmagic.java

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

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

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

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

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

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

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

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

The program does the following:

- Creates an `MWNumericArray` array to store the input data
- Instantiates a `magic` object
- Calls the `makesqr` method, where the first parameter specifies the number of output arguments and the following parameters are passed to the function in order as input arguments
- Uses a `try-catch` block to handle exceptions
- Frees native resources using `MWArray` methods

5. In MATLAB, navigate to the `MagicDemoJavaApp` folder.
6. Copy the generated `magicsquare.jar` package into this folder.

   - If you used `compiler.build.javaPackage`, type:

     ```
     copyfile(fullfile('..','MagicDemoComp','magicsquarejavaPackage','magisquare.jar'))
     ```

   - If you used the Library Compiler, type:

```
copyfile(fullfile('..','MagicDemoComp','magicsquare','for_testing','magicsquare.jar'))
```

7 In a system command window, navigate to the `PlotDemoJavaApp` folder.

8 Compile the Java application using `javac`.

- On Windows, execute this command:

```
javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\magicsquare.jar getmagic.java
```

- On UNIX®, execute this command:

```
javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./magicsquare.jar getmagic.java
```

Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Windows, the path may be `C:\Program Files\MATLAB\R2021b`.

For more details, see “Compile and Run MATLAB Generated Java Application”.

9 From the system command prompt, run the application.

- On Windows, type:

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

- On UNIX, type:

```
java -classpath .:"matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./makesqr.jar getmagic 5
```

The application outputs a 5-by-5 magic square in the command window.

```
    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
```

10 To follow up on this example:

- Try running the generated application on a different computer.
- Try building an installer for the package using `compiler.package.installer`.
- Try integrating a package that consists of multiple functions.

See Also

`libraryCompiler` | `compiler.build.javaPackage` | `compiler.package.installer` | `mcc` | `deploytool`

Related Examples

- “Generate Java Package and Build Java Application”
- “Create Java Application with Multiple MATLAB Functions” on page 5-6
- “Display MATLAB Plot in Java Application” on page 5-2

How MATLAB Compiler SDK Java Integration Works

In this section...

When you create Java packages using MATLAB Compiler SDK, the compiler encrypts your MATLAB functions and generates 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 `javabuilder` package that is provided with MATLAB and MATLAB Runtime contains the `MWArray` classes that manage data that passes between Java and MATLAB.

MWArray Data Conversion Classes

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

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

MWArray 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.

The `MWArray` classes 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

Note 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.

Automatic and Manual Conversion to MATLAB Types

If your Java code uses a native Java primitive or array as an input parameter, the compiler automatically 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 a multidimensional array of these types to an appropriate `MWArray` type.

In contrast, you can manually convert Java data types to one of the standard MATLAB data types using the `MWArray` data conversion classes. When you pass an `MWArray` instance as an input argument, the encapsulated MATLAB array is passed directly to the method being called. For more details and examples, see “Convert Data Between Java and MATLAB” on page 2-27.

For a list of all the data types that are supported along with their equivalent types in MATLAB, see “Rules for Data Conversion Between Java and MATLAB” on page 9-3 .

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.

When you pass an argument only a few times, it is usually just as efficient to pass a primitive Java type or object, which the calling mechanism automatically converts into an equivalent MATLAB type.

Function Signatures Generated by MATLAB Compiler SDK

The Java programming language supports optional function arguments in the way that MATLAB does with `varargin` and `varargout`. To support this MATLAB feature, the compiler generates a single overloaded Java method that accommodates any number of input arguments.

MATLAB Function Signatures

A 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 MATLAB Compiler SDK 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 of the MATLAB function. This method of encapsulating the information about return values resembles the `mlx` interface generated for the C/C++ MATLAB Compiler SDK target.

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

Note 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.

Interaction Between MATLAB Compiler SDK and JVM

Packages produced by MATLAB Compiler SDK use Java Native Interface (JNI) to interact with 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 MATLAB Runtime are loaded.
3. 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 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 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.

Limitations on Multiple Packages in Single Java Application

In this section...

When developing Java applications that use multiple MATLAB packages, consider that the following types of data cannot be shared between packages:

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

To work around these limitations, you can combine multiple Java packages into a single package.

Combining Packages 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 then `plot_xy` uses 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 compile 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 way to correct 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();
      }
   }
}
```

You could also correct this situation by using a singleton MATLAB Runtime. For more information, see “Share MATLAB Runtime Instances” on page 9-11.

Combining Packages with Objects

MATLAB Compiler SDK enables you to return the following types of objects from 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 conflict is to compile both functions into a single package. You could also refactor the application so that you are not passing MATLAB objects to the functions. You could also use a singleton MATLAB Runtime. See “Share MATLAB Runtime Instances” on page 9-11.

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.

Handle 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.

Handle Exception in 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, Double.valueOf((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 in JVM” on page 2-17.

Handle Exception in 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, Double.valueOf((double)n));
      return (double[])((MWArray)y[0]).getData();
   }

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

Handle 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, Double.valueOf((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, Double.valueOf((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 never executes.

Alternatives to Using 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 JVM

In this section...

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 MATLAB Runtime. Resources created by MATLAB Runtime are not visible to the JVM. The JVM garbage collector cannot effectively manage resources that it cannot see.

All of the MATLAB Compiler SDK Java classes have hooks that free MATLAB resources when the JVM garbage collector collects the wrapper objects. However, JVM garbage collection is unreliable because the JVM sees only the small wrapper object. The garbage collector can forgo spending CPU cycles to delete the small wrapper object. Until the Java wrapper object is deleted, the resources allocated in 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.

For information about the interaction between the interface for MATLAB and Java and the JVM, see “Interaction Between MATLAB Compiler SDK and JVM” on page 2-7.

Name MATLAB Objects for Resource Maintenance

All of the MATLAB objects supported by MATLAB Compiler SDK have standard Java constructors as described in the Java API documentation in *`matlabroot`*`/help/toolbox/javabuilder/MWArrayAPI`.

When creating MATLAB objects, always assign them names. For example, create a 5-by-5 cell array.

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

The Java object `myCA` is a wrapper that points to a 5-by-5 `mxCellArray` object in 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 5-by-5 `mxCellArray` by 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.

Consider the following code that 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 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. Because the Java wrapper consumes a tiny amount of space, the garbage collector is likely to leave it around. Since the JVM never cleans up the wrapper object, MATLAB Runtime never cleans up the resources it has allocated.

Now consider the following code, where 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.

Release Resources of MATLAB Objects

Clean up MATLAB objects by using the:

- Object's `dispose()` method
- 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);
```

See Also

Related Examples

- "How MATLAB Compiler SDK Java Integration Works" on page 2-5
- "Interaction Between MATLAB Compiler SDK and JVM" on page 2-7
- "Convert Data Between Java and MATLAB" on page 2-27

Interaction Between MATLAB Compiler SDK and JVM

Packages produced by MATLAB Compiler SDK use Java Native Interface (JNI) to interact with 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 MATLAB Runtime are loaded.
3. 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 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 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.

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 instance. Through calls to 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 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 2-21.

For more information, see “Using 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 Java Package and Build 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 Java Application

Write the following Java 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 2-20

Dynamically Specify Options to 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`

Sett and Retrieve 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 2-25)
`MWApplication.isMCRInitialized();`	Returns `true` if MATLAB Runtime is initialized; otherwise returns `false`
`MWApplication.isMCRJVMEnabled();`	Returns `true` if MATLAB Runtime 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 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 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 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 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).

Pass and Retrieve MATLAB Runtime Option Values from 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();
```

Convert Data Between Java and MATLAB

In this section...

When you invoke a MATLAB method from a generated class in your Java application, the input arguments received by the method must be in the MATLAB internal array format. You can either use manual conversion within the calling program by using instances of the `MWArray` classes, or rely on automatic conversion by storing your data using Java classes and data types, which are then automatically converted by the calling mechanism. Most likely, you will use a combination of manual and automatic conversion.

For example, consider the following Java statement:

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

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

Automatic Conversion to MATLAB Types

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

For 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.

Use MWNumericArray

The `getmagic` program (“Integrate Simple MATLAB Function Into Java Application” on page 2-2) converts a `java.lang.Double` argument 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);
```

Pass Java Double Object

This 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 single data input to `myprimes`.

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

The compiler converts the `java.lang.Double` argument to a MATLAB 1-by-1 `double` array.

Pass 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 `x` to a MATLAB scalar `double` and passes it to the MATLAB function.

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 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());

Array A is of type double
```

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.

Return 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. For more details, see “Use toTypeArray Methods to Specify Type and Dimensionality” on page 2-35.

Note For information on how to work directly with cell arrays and data structures in native Java, see “Represent Native Java Cell and Struct Arrays” on page 7-7.

Manual Conversion of Data Types

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

Change Default by Specifying Type

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

Array A is of type int16
```

Pass Variable Numbers of Inputs

Consider the following MATLAB function that returns the sum of the inputs:

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

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`.

MATLAB Compiler SDK generates the following Java interface to this function:

```
/* 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`, which lets you provide any number of inputs in the form of an array of Object (`Object[]`). 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, you create an `Object` array of length 1 and initialize it 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

This more general 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.

```
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();
   }
}
```

Pass Variable Number of Outputs

`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`. MATLAB Compiler SDK generates the following Java interface to this function:

```
/* 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 Standard Interface

Here is one way to use the `randvectors` method in a Java program:

getrandvectors.java

```
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 Of Unknown Type

The previous examples used the fact that you knew the type and dimensionality of the output argument. Sometimes, in MATLAB programming, this information is unknown or can vary. In this case, 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:

- 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.
- Specify type using the `to`*`Type`*`Array` methods.

Use Java Reflection to Determine Type and Dimensionality

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.

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

`getprimes.java` Using Reflection

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

   try
   {
      cls = new myclass();
      y = cls.myprimes(1, Double.valueOf((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");
      }
   }
```

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.

`getprimes.java` Using `classID`

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

   try
   {
      cls = new myclass();
      y = cls.myprimes(1, Double.valueOf((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 Specify 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.

Specify Type Using to*Type*Array Method

```
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

You can create a MATLAB code wrapper around Java objects using `MWJavaObjectRef`, a special subclass of `MWArray`. 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 MATLAB Compiler SDK. The examples in this section present some common use cases.

Pass Java Object into 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);
```

Clone an Object

You can also use `MWJavaObjectRef` to clone an object by doing 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());
```

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);
```

Pass Date into Method and Get Date from Method

In addition to passing in created objects, you can also use `MWJavaObjectRef` to pass utility objects such as `java.util.date`.

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());
```

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();
```

Return Java Objects Using unwrapJavaObjectRefs

If you want actual Java objects returned from a method, 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 fragments show two examples of calling `unwrapJavaObjectRefs`:

Return Single Reference or Reference to Array of Objects Using `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))
```

This code produces the following output:

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

Return Array of References Using `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]);
```

This code 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.

See Also

`varargin` | `varargout`

Related Examples

- “How MATLAB Compiler SDK Java Integration Works” on page 2-5
- “Programming Interfaces Generated by MATLAB Compiler SDK” on page 9-8
- “Rules for Data Conversion Between Java and MATLAB” on page 9-3
- “Manage MATLAB Resources in JVM” on page 2-17

Set Java Properties

In this section...

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 “Set Java System Properties” on page 2-39:

- 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();
               }
           }
  ```

Block Console Display When Creating Figures in Java

This example shows how to use `waitForFigures` from a Java application that you create using MATLAB Compiler SDK. The object encapsulates MATLAB code that draws a simple plot.

1. Create a MATLAB function named `drawplot.m` with the following code:

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

2. Build the Java package with the **Library Compiler** app or `compiler.build.javaPackage` using the following information:

Field	Value
Library Name	`examples`
Class Name	`Plotter`
File to Compile	`drawplot.m`

   For example, if you are using `compiler.build.javaPackage`, type:

   ```
   buildResults = compiler.build.javaPackage('drawplot.m', ...
   'PakageName','examples', ...
   'ClassName','Plotter');
   ```

   For more details, see the instructions in “Generate Java Package and Build Java Application”.

3. 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();
       }
     }
   }
   ```

4. In MATLAB, copy the generated `examples.jar` package into your current folder.

   - If you used `compiler.build.javaPackage`, type:

     ```
     copyfile(fullfile('examplesjavaPackage','examples.jar'))
     ```

   - If you used the Library Compiler, type:

     ```
     copyfile(fullfile('examples','for_testing','examples.jar'))
     ```

5. In a command prompt window, navigate to your work folder.

6. Compile the application using `javac`.

   - On Windows, type:

     ```
     javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\examples.jar runplot.java
     ```

   - On UNIX, type:

     ```
     javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./examples.jar runplot.java
     ```

   Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Windows, the path may be `C:\Program Files\MATLAB\R2021b`.

7. Run the application.

   - On Windows, type:

     ```
     java -classpath .;"matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\examples.jar runplot
     ```

   - On UNIX, type:

     ```
     java -classpath .:"matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./examples.jar runplot
     ```

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

   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 immediately closes as the application exits.

Ensure Multiplatform Portability for Java

Compiled MATLAB code containing only MATLAB files are platform independent, as are Java `.jar` files. You can use these files on any platform, provided that the platform has either MATLAB or 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 the application on a Windows machine, and you want it to 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 `-a` flag of `mcc` or the `AdditionalFiles` option in a `compiler.build` function to include the MEX file compiled on the other platform(s).

   In the case above, you run `mcc` on Windows and include the `-a` flag to include *`my_mex`*`.mexa64`. It is not necessary to explicitly include *`my_mex`*`.mexw64`. In this step , the `mcc` command would be:

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

   For example, if you are running `mcc` 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
   ```

   `callyprime.m` can be a simple MATLAB function, as follows:

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

   Ensure that 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, or enable the `Verbose` option in a `compiler.build` function.
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, or use the `AdditionalFiles` options in a `compiler.build` function.

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 run only on that specific platform when deployed. For example, functionality from the Data Acquisition Toolbox™ runs only on Windows.

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

Define Embedding and Extraction Options for Deployable Java Archive

In this section...

When you deploy a Java archive, by default, the 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 to extract deployable archive data by default, you can use either the `MWComponentOptions` class or environment variables to specify options for extraction and utilization of the deployable archive data.

Extraction Options Using MWComponentOptions Class

Select Options

Choose from the following `CtfSource` or `ExtractLocation` instantiation options to customize how MATLAB Runtime manages 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`:

Value	Description
`MWCtfSource.NONE`	Indicates that no deployable archive is to be extracted. This option implies that the extracted deployable archive data is already accessible somewhere in your file system. This object 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 that 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 MATLAB Runtime requires all deployable archive content be located somewhere in 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`. You can create an instance of this class by passing a `java.io.File` or `java.lang.String` into the constructor to specify the file system location to be used, or you can use one of these predefined, static final instances:

Value	Descriptin
`MWCtfExtractLocation.EXTRACT_TO_CACHE`	Use to indicate that the deployable archive content is to be placed in the MATLAB Runtime component cache. This option 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 where the component has been loaded. If the location is found (e.g., it is in 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`*.

Set 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()`

Enable MATLAB Runtime Component Cache

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` by 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);
```

Extraction Options Using Environment Variables

Use the following environment variables to change the default settings for the cache size and location of the deployable archive extraction.

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.	On macOS, this variable is ignored in MATLAB R2020a and later. The app bundle contains the files necessary for runtime.
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.

See Also

Related Examples

- “Deployable Archive”

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 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 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 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 MATLAB Runtime

Supported Platforms: Windows, Linux, macOS

MATLAB Runtime contains the libraries needed to run MATLAB applications on a target system without a licensed copy of MATLAB.

Download MATLAB Runtime Installer

Download MATLAB Runtime using one of the following options:

- Download the MATLAB Runtime installer at the latest update level for the selected release from the website at https://www.mathworks.com/products/compiler/matlab-runtime.html. This option is best for end users who want to run deployed applications.
- Use the MATLAB function `compiler.runtime.download` to download the MATLAB Runtime installer matching the version and update level of MATLAB from where the command is executed. If the installer has already been downloaded to the machine, it returns the path to the MATLAB Runtime installer. If the machine is offline, it returns a URL to the MATLAB Runtime installer. This option is best for developers who want to create application installers that contain MATLAB Runtime.

Install MATLAB Runtime Interactively

To install MATLAB Runtime:

1 Extract the archive containing the MATLAB Runtime installer.

Platform	Steps
Windows	Unzip the MATLAB Runtime installer. Right-click the ZIP file `MATLAB_Runtime_R2021b_win64.zip` and select **Extract All**.
Linux	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2021b MATLAB Runtime installer, at the terminal, type: `unzip MATLAB_Runtime_R2021b_glnxa64.zip`
macOS	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2021b MATLAB Runtime installer, at the terminal, type: `unzip MATLAB_Runtime_R2021b_maci64.zip`

Note The release part of the installer file name (*_R2021b_*) changes 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 -H ./install` **Note** The `-H` option sets the `HOME` environment variable to the home directory of the root user and should be used for graphical applications such as installers.
macOS	At the terminal, type: `./install` **Note** You may need to enter an administrator user name and password after you run `./install`.

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.

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. In the **Folder Selection** dialog box, specify the folder in which you want to install MATLAB Runtime.

   Note You can have multiple versions of 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 it overwrites 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, save it to a text file, and then click **Next**. For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.
7. Click **Finish** to exit the installer.

The default MATLAB Runtime installation directory for **R**2021**b** is specified in the following table:

Operating System	MATLAB Runtime Installation Directory
Windows	`C:\Program Files\MATLAB\MATLAB Runtime\v911`
Linux	`/usr/local/MATLAB/MATLAB_Runtime/v911`
macOS	`/Applications/MATLAB/MATLAB_Runtime/v911`

Install MATLAB Runtime Noninteractively

To install MATLAB Runtime without having to interact with the installer dialog boxes, use one of these noninteractive 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 values 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 installation location.

Run Installer in Silent Mode

To install MATLAB Runtime in silent mode:

1. Extract the contents of the MATLAB Runtime installer archive to a temporary folder.
2. In your system command prompt, navigate to the folder where you extracted the installer.
3. Run the MATLAB Runtime installer, specifying the `-mode silent` and `-agreeToLicense yes` options 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 64-bit Windows, the installer is located in the archive `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` option, the installer does not install MATLAB Runtime.
4. 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 login name, in the location defined by your `TEMP` environment variable.
5. On Linux and macOS systems, the MATLAB Runtime installer displays the log information at the command prompt and also saves it to a file if you use the `-outputFile` option.

Customize a Noninteractive Installation

When run in one of the noninteractive modes, the installer uses the default values unless you specify 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 MATLAB Runtime is installed.
`-outputFile`	Specifies where the installation log file is written.
`-tmpdir`	Specifies where temporary files are stored during installation. **Caution** The installer deletes everything inside the specified folder.
`-automatedModeTimeout`	Specifies how long, in milliseconds, that each dialog box is displayed when run in automatic mode.
`-inputFile`	Specifies an installer control file that contains your command-line options and values. Omit the dashes and put each option and value pair on a separate line.

Note The MATLAB 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. The options listed in this section are valid for the MATLAB Runtime installer.

Install MATLAB Runtime without Administrator Rights

To install 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 MATLAB Runtime was installed to the machine without administrator rights. You can compress the folder into a zip file for distribution.
3. On the machine without administrator rights, add the *<MATLAB_RUNTIME_INSTALL_DIR>*`\runtime\`*arch* directory to the user's `PATH` environment variable. For more information, see "Set MATLAB Runtime Path for Deployment".

Install Multiple MATLAB Runtime Versions on Single Machine

`MCRInstaller` supports the installation of multiple versions of MATLAB Runtime on a target machine. This capability allows applications compiled with different versions of 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 a specific version. On Linux, manually delete the unwanted MATLAB Runtime directories. You can remove unwanted versions before or after installation of a more recent version of MATLAB Runtime because versions can be installed or removed in any order.

Note Installing multiple versions of MATLAB Runtime on the same machine is not supported on macOS.

Install MATLAB and MATLAB Runtime on Same Machine

To test your deployed component on your development machine, you do not need an installation of MATLAB Runtime. The MATLAB installation that you use to compile the component can act as a MATLAB Runtime replacement.

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

Modify Path

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

To run deployed MATLAB code against MATLAB Runtime rather than MATLAB, ensure that your library path lists the MATLAB Runtime directories before any MATLAB directories.

For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.

Uninstall MATLAB Runtime

The method you use to uninstall MATLAB Runtime from your computer varies depending on your platform.

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 `<MATLAB_RUNTIME_INSTALL_DIR>\uninstall\bin\<arch>` folder, where `<MATLAB_RUNTIME_INSTALL_DIR>` is your MATLAB Runtime installation folder and `<arch>` is an architecture-specific folder, such as `win32` or `win64`.
2. Select MATLAB Runtime from the list of products in the Uninstall Products dialog box and click **Next**.
3. Click **Finish**.

Linux

1. Close all instances of MATLAB and MATLAB Runtime.
2. Enter this command at the Linux terminal:

   ```
   rm -rf <MATLAB_RUNTIME_INSTALL_DIR>
   ```

   Caution Be careful when using the `rm` command, as deleted files cannot be recovered.

macOS

1. Close all instances of MATLAB and MATLAB Runtime.
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.

See Also

`compiler.runtime.download`

More About

- About MATLAB Runtime
- “MATLAB Runtime Startup Options”
- “Set MATLAB Runtime Path for Deployment”

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 for Generating Packages” 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:
```

Set MATLAB Runtime Path for Deployment

In this section...

After you install MATLAB Runtime, add the run-time directories to the system library path according to the instructions for your operating system and shell environment.

Note

- Ensure that the MATLAB Runtime directories are not already present in the path before adding them.
- Save the value of your current library path as a backup before modifying it.
- Your library path may contain multiple versions of MATLAB Runtime. Applications launched without using the shell script use the first version listed in the path.

Windows

The MATLAB Runtime installer for Windows automatically sets the library path during installation. If you do not use the installer, complete the following steps to set the `PATH` environment variable permanently.

Graphical Approach

1. Run `C:\Windows\System32\SystemPropertiesAdvanced.exe` and click the **Environment Variables...** button.
2. Select the system variable `Path` and click **Edit...**.

   Note If you do not have administrator rights on the machine, select the user variable `Path` instead of the system variable.
3. Click **New** and add the directory *<MATLAB_RUNTIME_INSTALL_DIR>*`\runtime\`*<arch>*. For example, if you are using MATLAB Runtime R2021b located in the default installation directory on 64-bit Windows, add `C:\Program Files\MATLAB\MATLAB Runtime\v911\runtime\win64`.
4. Click **OK** to apply the change.

Command-Line Approach Using PowerShell

1. Execute the following command using Windows PowerShell with elevated privileges to save the current system path as a variable named `mypath` and display it in the console.

   ```
   ($mypath = (Get-Item "HKLM:\System\CurrentControlSet\Control\Session Manager\Environment").Ge
   ```
2. Append your MATLAB Runtime directory to the `mypath` variable. If there is a semicolon at the end of your current path or the path is empty, then delete the semicolon from the following command.

```
$mypath = "${mypath};<MATLAB_RUNTIME_INSTALL_DIR>\runtime\<arch>"
```

For example, if you are using MATLAB Runtime R2021b located in the default installation directory on 64-bit Windows, use the following command:

```
$mypath = "${mypath};C:\Program Files\MATLAB\MATLAB Runtime\v911\runtime\win64"
```

3. Display the value of `mypath` to ensure the path is correct.

```
echo %mypath%
```

4. Set the system path to the value of `mypath`.

```
Set-ItemProperty -Path 'Registry::HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Sessior
```

5. Sign out of Windows to apply the change.

Linux

In the terminal, display the `SHELL` variable to determine your current shell environment.

```
echo $SHELL
```

For information on setting environment variables in shells other than Bash, see your shell documentation.

Bash Shell

1. Save your current library path as a variable named `mypath` and display it in the console.

```
mypath=$LD_LIBRARY_PATH && echo $mypath
```

2. Append the MATLAB Runtime directories to the `mypath` variable. The following command must be entered as a single line.

```
mypath="${mypath:+${mypath}:}
<MATLAB_RUNTIME_INSTALL_DIR>/runtime/glnxa64:
<MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64:
<MATLAB_RUNTIME_INSTALL_DIR>/sys/os/glnxa64:
<MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/glnxa64"
```

Note If you are using Mesa OpenGL®, replace `/extern/bin/glnxa64` with `/sys/opengl/lib/glnxa64`.

For example, if you are using MATLAB Runtime R2021b located in the default installation directory, use the following command entered on a single line:

```
mypath="${mypath:+${mypath}:}/usr/local/MATLAB/MATLAB_Runtime/v911/runtime/glnxa64:
/usr/local/MATLAB/MATLAB_Runtime/v911/bin/glnxa64:/usr/local/MATLAB/MATLAB_Runtime/v911/exter
```

3. Display the value of `mypath` to ensure the path is correct.

```
echo $mypath
```

4. Set the `LD_LIBRARY_PATH` variable for the current session.

```
export LD_LIBRARY_PATH=$mypath
```

5. To set the variable permanently, append the `export` command to a file in your home directory named `.bashrc` using the following command:

```
echo "export LD_LIBRARY_PATH=$mypath" >> ~/.bashrc
```

6. To apply changes to the current session, type `source ~/.bashrc`.
7. Type `ldd --version` to check your version of GNU® C library (`glibc`). If the version displayed is 2.17 or lower, add *<MATLAB_RUNTIME_INSTALL_DIR>*`/bin/glnxa64/glibc-2.17_shim.so` to the `LD_PRELOAD` environment variable using the above procedure.

macOS

In the terminal, display the `SHELL` variable to determine your current shell environment.

```
echo $SHELL
```

Zsh Shell (macOS Catalina 10.15 or Later)

1. Save your current library path as a variable named `mypath` and display it in the console.

   ```
   mypath=$DYLD_LIBRARY_PATH && echo $mypath
   ```

2. Append the MATLAB Runtime directories to the `mypath` variable. The following command must be entered as a single line.

   ```
   mypath="${mypath:+${mypath}:}
   <MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64:
   <MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64:
   <MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64"
   ```

   For example, if you are using MATLAB Runtime R2021b located in the default installation directory, use the following command entered on a single line:

   ```
   mypath="${mypath:+${mypath}:}/Applications/MATLAB/MATLAB_Runtime/v911/runtime/maci64:
   /Applications/MATLAB/MATLAB_Runtime/v911/bin/maci64:/Applications/MATLAB/MATLAB_Runtime/v911/
   ```

3. Display the value of `mypath` to ensure the path is correct.

   ```
   echo $mypath
   ```

4. Set the `DYLD_LIBRARY_PATH` variable for the current session.

   ```
   export DYLD_LIBRARY_PATH=$mypath
   ```

5. To set the variable permanently, append the `setenv` command to a file in your home directory named `.zshrc` using the following command:

   ```
   echo "export DYLD_LIBRARY_PATH=$mypath" >> ~/.zshrc
   ```

6. To apply changes to the current session, type `source ~/.zshrc`.

Bash Shell (macOS Mojave 10.14 or Earlier)

1. Save your current library path as a variable named `mypath` and display it in the console.

   ```
   mypath=$DYLD_LIBRARY_PATH && echo $mypath
   ```

2. Append the MATLAB Runtime directories to the `mypath` variable. The following command must be entered as a single line.

   ```
   mypath="${mypath:+${mypath}:}
   <MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64:
   <MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64:
   <MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64"
   ```

For example, if you are using MATLAB Runtime R2021b located in the default installation directory, use the following command entered as a single line:

```
mypath="${mypath:+${mypath}:}/Applications/MATLAB/MATLAB_Runtime/v911/runtime/maci64
:/Applications/MATLAB/MATLAB_Runtime/v911/bin/maci64:/Applications/MATLAB/MATLAB_Runtime/v911
```

3. Display the value of `$mypath` to ensure the path is correct.

```
echo $mypath
```

4. Set the `DYLD_LIBRARY_PATH` variable for the current session.

```
export DYLD_LIBRARY_PATH=$mypath
```

5. To set the variable permanently, append the `export` command to a file in your home directory named `.profile` using the following command:

```
echo "export DYLD_LIBRARY_PATH=$mypath" >> ~/.profile
```

6. To apply changes to the current session, type `source ~/.profile`.

See Also

More About

- “Install and Configure MATLAB Runtime”
- “Change Environment Variable for Shell Command”

Sample Java Applications

Note Remember to double-quote all parts of the `java` command paths that contain spaces.

Display MATLAB Plot in Java Application

In this section...

In this example, you integrate a MATLAB function into a Java application by performing these steps:

1. 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`).
2. 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.
3. Build and run the `createplot.java` application.

Files

MATLAB Function Location	*matlabroot*`\toolbox\javabuilder\Examples\PlotExample\PlotDemoComp\drawplot.m`
Java Code Location	*matlabroot*`\toolbox\javabuilder\Examples\PlotExample\PlotDemoJavaApp\createplot.java`

Procedure

1. Copy the `PlotExample` folder that ships with MATLAB to your work folder:

   ```
   copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','PlotExample'),'PlotExample')
   ```

   At the MATLAB command prompt, navigate to the new `PlotExample\PlotDemoComp` subfolder in your work folder.
2. Examine the `drawplot.m` function.

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

   The function displays a plot of input parameters `x` and `y`.
3. Create a Java package by using the **Library Compiler** app or `compiler.build.javaPackage` using the following information:

Project Name	`plotdemo`
Class Name	`plotter`
File to Compile	`drawplot.m`

   For example, if you are using `compiler.build.javaPackage`, type:

   ```
   buildResults = compiler.build.javaPackage('drawplot.m', ...
   'PackageName','plotdemo', ...
   'ClassName','plotter');
   ```

For more details, see the instructions in “Generate Java Package and Build Java Application”.

4 Write source code for a Java application that accesses the MATLAB function.

The sample application for this example is in `PlotExample\PlotDemoJavaApp\createplot.java`.

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 `x` and `y` using `MWNumericArray` to represent the equation $y = x^2$
- Instantiates the `plotter` class as `thePlot` object

  ```
  thePlot = new plotter();
  ```
- Calls the `drawplot` method to plot a simple parabola using the MATLAB `plot` function

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

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

5 In MATLAB, navigate to the `PlotDemoJavaApp` folder.

6 Copy the generated `plotdemo.jar` package into this folder.

- If you used `compiler.build.javaPackage`, type:

```
copyfile(fullfile('..','PlotDemoComp','plotdemojavaPackage','plotdemo.jar'))
```

- If you used the Library Compiler, type:

```
copyfile(fullfile('..','PlotDemoComp','plotdemo','for_testing','plotdemo.jar'))
```

7 In a command prompt window, navigate to the `PlotDemoJavaApp` folder where you copied `plotdemo.jar`.

8 Compile the `createplot` application using `javac`.

- On Windows, execute this command:

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

- On UNIX, execute this command:

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

Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Windows, the path may be `C:\Program Files\MATLAB\R2021b`.

9 Run the `createplot` application.

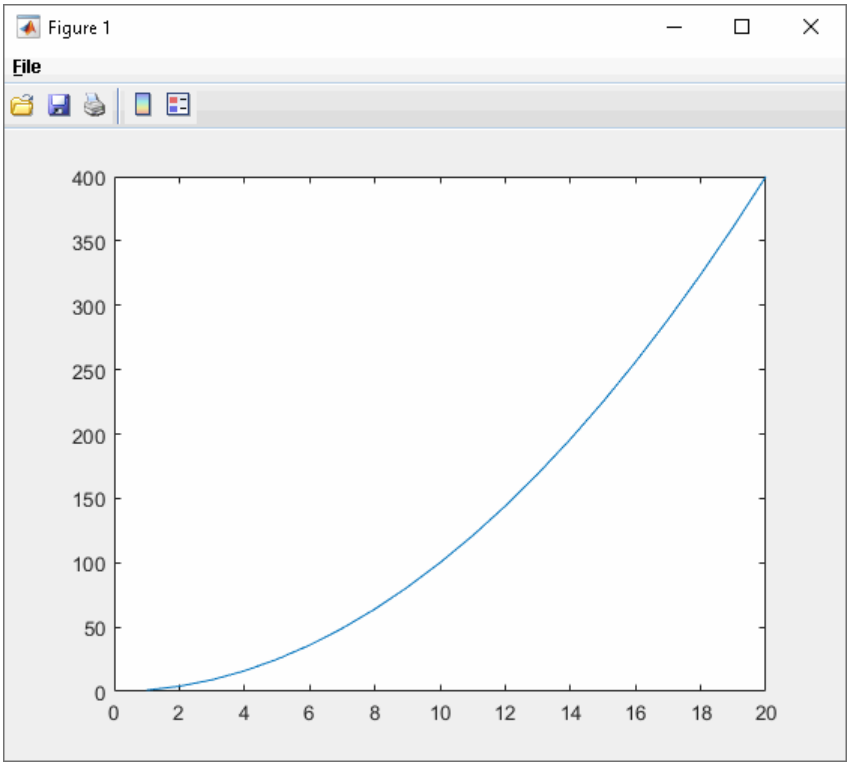
- On Windows, type:

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

- On UNIX, type:

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

The `createplot` program displays the following output.



See Also

`libraryCompiler` | `compiler.build.javaPackage`

Related Examples

- “Create Java Phone Book Application Using Structure Array” on page 5-18

Create Java Application with Multiple MATLAB Functions

In this section...

This example shows how to create a Java application that uses multiple MATLAB functions to analyze a signal and then graph the result.

In this example, you perform the following steps:

1. Use MATLAB Compiler SDK to create a package containing a class that has a private method that is automatically encapsulated.
2. Access the MATLAB functions in a Java application, including use of the `MWArray` class hierarchy to represent data.
3. Build and run the application.

spectralanalysis Application

The `spectralanalysis` application 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.

Files

MATLAB Functions	`computefft.m` `plotfft.m`
MATLAB Function Location	*`matlabroot`*`\toolbox\javabuilder\Examples\SpectraExample\SpectraDemoComp`
Java Code Location	*`matlabroot`*`\toolbox\javabuilder\Examples\SpectraExample\SpectraDemoJavaApp\powerspect.java`

Procedure

1. Copy the `SpectraExample` folder that ships with MATLAB to your work folder:

   ```
   copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','SpectraExample'),'SpectraExa
   ```

   At the MATLAB command prompt, navigate to the new `SpectraExample\SpectraDemoComp` subfolder in your work folder.
2. Examine the MATLAB functions `computefft.m` and `plotfft.m`.

   computefft.m

   ```
   function [fftdata, freq, powerspect] = computefft(data, interval)
   if (isempty(data))
   ```

```
    fftdata = [];
    freq = [];
    powerspect = [];
    return;
end

if (interval <= 0)
    error('BuilderJA:Examples:samplingInterval','Sampling interval must be greater then zero')
    return;
end

fftdata = fft(data);
freq = (0:length(fftdata)-1)/(length(fftdata)*interval);
powerspect = abs(fftdata)/(sqrt(length(fftdata)));
```

plotfft.m

```
function [fftdata, freq, powerspect] = plotfft(data, interval)
[fftdata, freq, powerspect] = computefft(data, interval);
len = length(fftdata);
if (len <= 0)
    return;
end

t = 0:interval:(len-1)*interval;
subplot(2,1,1), plot(t, data)
xlabel('Time'), grid on
title('Time domain signal')
subplot(2,1,2), plot(freq(1:len/2), powerspect(1:len/2))
xlabel('Frequency (Hz)'), grid on
title('Power spectral density')
```

3 Build the Java package with the **Library Compiler** app or `compiler.build.javaPackage`.

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 call `computefft` directly. The `computefft` method is required only by the `plotfft` method. You do not need to manually add the `computefft` function to the package, as the compiler automatically includes it during dependency analysis.

For example, if you are using `compiler.build.javaPackage`, type:

```
buildResults = compiler.build.javaPackage('plotfft.m', ...
'PackageName','spectralanalysis', ...
'ClassName','fourier');
```

For more details, see the instructions in “Generate Java Package and Build Java Application”.

4 Write source code for a Java application that accesses the MATLAB functions.

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

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();

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

      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

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

- Instantiates a `fourier` object
- Calls the `plotfft` method, which calls `computeftt` and plots the data

- Uses a `try-catch` block to handle exceptions
- Frees native resources using `MWArray` methods

5. In MATLAB, navigate to the `SpectraDemoJavaApp` folder.
6. Copy the generated `spectralanalysis.jar` package into this folder.

   - If you used `compiler.build.javaPackage`, type:

     ```
     copyfile(fullfile('..','SpectraDemoComp','spectralanalysisjavaPackage','spectralanalysis.jar'))
     ```

   - If you used the Library Compiler, type:

     ```
     copyfile(fullfile('..','SpectraDemoComp','spectralanalysis','for_testing','spectralanalysis.jar'))
     ```

7. Open a command prompt window and navigate to the `SpectraDemoJavaApp` folder.
8. Compile the `powerspect.java` application using `javac`.

   - On Windows, execute the following command:

     ```
     javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\spectralanalysis.j
     ```

   - On UNIX, execute the following command:

     ```
     javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./spectralanalysis.j
     ```

   Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Windows, the path may be `C:\Program Files\MATLAB\R2021b`.
9. Run the `powerspect` application.

   - On Windows, execute the following command:

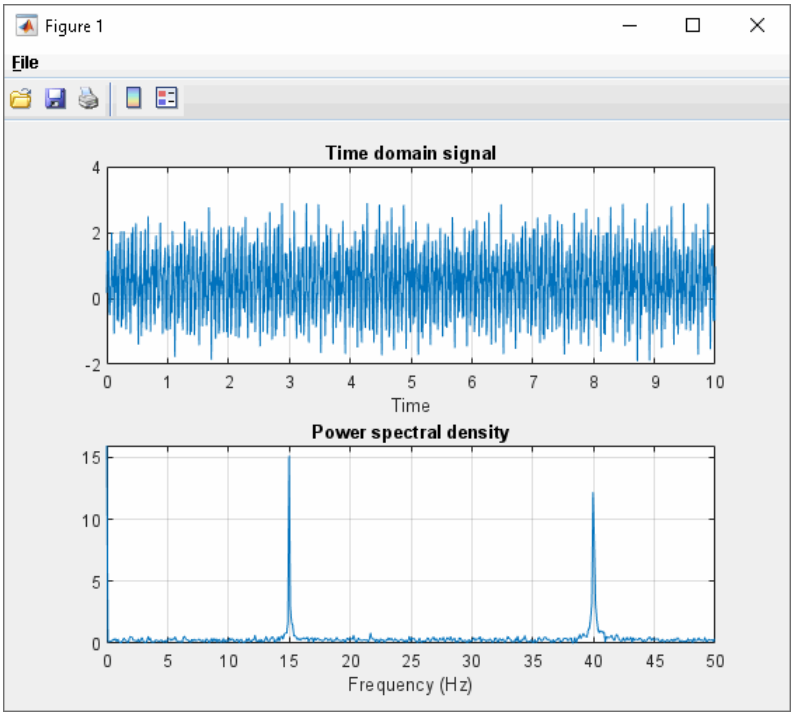
     ```
     java -classpath .;"matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\spectralanalysis.
     ```

   - On UNIX, execute the following command:

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

Note If you are running the application on the Mac 64-bit platform, you must add the `-d64` flag in the Java command.

The `powerspect` program displays the following output:



See Also

`libraryCompiler` | `compiler.build.javaPackage`

Related Examples

- “Assign Multiple MATLAB Functions to Java Class” on page 5-11

Assign Multiple MATLAB Functions to Java Class

In this section...

This example shows you how to create a Java matrix math program using multiple MATLAB functions.

In this example, you perform the following steps:

1. Assign more than one MATLAB function to a generated class.
2. Manually handle native memory management.
3. Access the MATLAB functions in a Java application (`getfactor.java`) by instantiating `Factor` and using the `MWArray` class library to handle data conversion.
4. Build and run the `MatrixMathDemoApp` application.

MatrixMathApp Application

The `MatrixMathApp` application 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.

Files

MATLAB Functions	`cholesky.m`
	`ludecomp.m`
	`qrdecomp.m`

MATLAB Function Location	*`matlabroot`*`\toolbox\javabuilder\Examples\MatrixMathExample\MatrixMathDemoComp\`
Java Code Location	*`matlabroot`*`\toolbox\javabuilder\Examples\MatrixMathExample\MatrixMathDemoJavaApp\getfactor.java`

Procedure

1. Copy the `MatrixMathExample` folder that ships with MATLAB to your work folder:

   ```
   copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','MatrixMathExample'),'MatrixM
   ```

At the MATLAB command prompt, navigate to the new `MatrixMathExample\MatrixMathDemoComp` subfolder in your work folder.

2. If you have not already done so, set up your Java development environment. For details, see "Configure Your Java Environment for Generating Packages" on page 1-3.
3. Examine the MATLAB functions `cholesky.m`, `ludecomp.m`, and `qrdecomp.m`.

```
function [L] = Cholesky(A)
    L = chol(A);

function [L,U] = LUDecomp(A)
    [L,U] = lu(A);

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

4. Build the Java package with the **Library Compiler** app or `compiler.build.javaPackage` using the following information:

Field	Value
Library Name	`factormatrix`
Class Name	`factor`
Files to Compile	`cholesky ludecomp qrdecomp`

For example, if you are using `compiler.build.javaPackage`, type:

```
buildResults = compiler.build.javaPackage(["cholesky.m","ludecomp.m","qrdecomp.m"], ...
'PackageName','factormatrix', ...
'ClassName','factor');
```

For more details, see the instructions in "Generate Java Package and Build Java Application".

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

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

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();
     }
  }
}
```

This statement creates an instance of the class `factor`:

```
theFactor = new 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);
...
```

6. In MATLAB, navigate to the `MatrixMathDemoJavaApp` folder.
7. Copy the generated `factormatrix.jar` package into this folder.

   - If you used `compiler.build.javaPackage`, type:

     ```
     copyfile(fullfile('..','MatrixMathDemoComp','factormatrixjavaPackage','factormatrix.jar'))
     ```

   - If you used the Library Compiler, type:

     ```
     copyfile(fullfile('..','MatrixMathDemoComp','factormatrix','for_testing','factormatrix.jar'))
     ```

8. In a command prompt window, `cd` to the`MatrixMathDemoJavaApp` folder.
9. Compile the `getfactor` application using `javac`.

   - On Windows, type:

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

   - On UNIX, type:

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

   Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Linux, the path may be `/usr/local/MATLAB/R2021b`.
10. Run the `getfactor` application using a nonsparse matrix.

    - On Windows, type:

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

    - On UNIX, type:

      ```
      java -classpath .:"matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./factormatrix.jar getfactor 4
      ```

Note If you are running the application on the Mac 64-bit platform, you must add the `-d64` flag in the Java command.

Output for Nonsparse Matrix

```
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` at the end. For example, on Windows, type:

```
java -classpath .;"matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\factormatrix.jar getfac
```

Output for 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
```

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, 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.

See Also

`libraryCompiler` | `compiler.build.javaPackage`

Related Examples

- “Use MATLAB Class in Java Application” on page 5-28

Create Java Phone Book Application Using Structure Array

In this example, you create a Java package that calls a MATLAB function to modify a structure array and implement a phone book application.

Files

MATLAB Function	`makephone.m`
MATLAB Function Location	*`matlabroot`*`\toolbox\javabuilder\Examples\PhoneExample\PhoneDemoComp\`
Java Code Location	*`matlabroot`*`\toolbox\javabuilder\Examples\PhoneExample\PhoneDemoJavaApp\getphone.java`

Procedure

1. Copy the `PhoneExample` folder that ships with MATLAB to your work folder:

   ```
   copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','PhoneExample'),'PhoneExample
   ```

   At the MATLAB command prompt, navigate to the new `PhoneExample\PhoneDemoComp` subfolder in your work folder.
2. Examine the `makephone.m` function.

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

   The function takes a structure array as an input, modifies it, and supplies the modified array as an output.
3. Build the Java package with the **Library Compiler** app or `compiler.build.javaPackage` using the following information:

Field	Value
Library Name	`phonebookdemo`
Class Name	`phonebook`
File to Compile	`makephone.m`

   For example, if you are using `compiler.build.javaPackage`, type:

   ```
   buildResults = compiler.build.javaPackage('makephone.m', ...
   'PackageName','phonebookdemo', ...
   'ClassName','phonebook');
   ```

   For more details, see the instructions in “Generate Java Package and Build Java Application”.
4. Write source code for an application that accesses the MATLAB functions.

   The sample application for this example is in `PhoneExample\PhoneDemoJavaApp\getphone.java`.

   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:

  ```
  thePhonebook = new phonebook();
  ```

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

  ```
  result = thePhonebook.makephone(1, friends);
  ```

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

5. In MATLAB, navigate to the `PhoneExample\PhoneDemoJavaApp` folder.
6. Copy the generated `phonebookdemo.jar` package into this folder.

   - If you used `compiler.build.javaPackage`, type:

     ```
     copyfile(fullfile('..','PhoneDemoComp','phonebookdemojavaPackage','phonebookdemo.jar'))
     ```

   - If you used the Library Compiler, type:

     ```
     copyfile(fullfile('..','PhoneDemoComp','phonebookdemo','for_testing','phonebookdemo.jar'))
     ```

7. In a command prompt window, `cd` to the`PhoneDemoJavaApp` folder.
8. Compile the `getphone` application using `javac`.

   - On Windows, type:

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

   - On UNIX, type:

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

   Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Linux, the path may be `/usr/local/MATLAB/R2021b`.
9. Run the `getphone` application.

   - On Windows, type:

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

   - On UNIX, type:

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

> **Note** If you are running the application on the Mac 64-bit platform, you must add the `-d64` flag in the Java command.

The `getphone` program displays the following 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
```

See Also

`libraryCompiler` | `compiler.build.javaPackage`

Related Examples

- “Create Java Application with Multiple MATLAB Functions” on page 5-6

Pass Java Objects to MATLAB

In this section...

Overview

This example shows you how to create a Java application that finds a local minimum of an objective function using the MATLAB optimization function `fminsearch` and the `MWJavaObjectRef` class.

In this example, you perform the following steps:

1. Use MATLAB Compiler SDK to create a package that applies MATLAB optimization routines to objective functions implemented as Java objects.
2. 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.
3. 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 Optimizer class 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`.

Files

MATLAB Functions	`doOptim.m` `displayObj.m`
MATLAB Function Location	*`matlabroot`*`\toolbox\javabuilder\Examples\ObjectRefExample\ObjectRefDemoComp`
Java Code Location	*`matlabroot`*`\toolbox\javabuilder\Examples\ObjectRefExample\ObjectRefDemoJavaApp`
`javabuilder.jar`	*`matlabroot`*`\toolbox\javabuilder\jar\win64`

Procedure

1. Copy the `ObjectRefExample` folder that ships with MATLAB to your work folder:

   ```
   copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','ObjectRefExample'),'ObjectRe
   ```

   At the MATLAB command prompt, navigate to the new `ObjectRefExample\ObjectRefDemoComp` subfolder in your work folder.
2. Examine the MATLAB code you want to access from Java. This example uses `doOptim.m` and `displayObj.m`.

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

   ```
   function className = displayObj(h)
   h
   className = class(h)
   whos('h')
   methods(h)
   ```
3. Build the Java package with the **Library Compiler** app or `compiler.build.javaPackage` using the following information:

Field	Value
Library Name	`OptimDemo`
Class Name	`Optimizer`
Files to Compile	`doOptim.m` `displayObj.m`

   For example, if you are using `compiler.build.javaPackage`, type:

   ```
   buildResults = compiler.build.javaPackage(["doOptim.m","displayObj.m"], ...
   'PackageName','OptimDemo', ...
   'ClassName','Optimizer');
   ```

   For more details, see the instructions in “Generate Java Package and Build Java Application”.
4. Write source code for a class that implements an object function to optimize. The code for this example is in the file `BananaFunction.java`.

   BananaFunction.java

   ```
   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.
5. Write source code for an application that accesses the MATLAB functions. The code for this example is in the file `PerformOptim.java`.

PerformOptim.java

```
/* 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 does the following:

- Instantiates an object of the `BananaFunction` class above to be optimized.
- Creates an `MWJavaObjectRef` that references the `BananaFunction` object, as shown:

  ```
  origRef = new MWJavaObjectRef(objectiveFunction);
  ```

  .
- Instantiates an Optimizer object.

- Calls the `displayObj` method to verify that the Java object is being passed correctly.
- Calls the `doOptim` method, which uses `fminsearch` to find a local minimum of the objective function.
- Uses a `try/catch` block to handle exceptions.
- Frees native resources using `MWArray` methods.

6 In MATLAB, navigate to the `ObjectRefDemoJavaApp` folder.

7 Copy the generated `OptimDemo.jar` package into this folder.

- If you used `compiler.build.javaPackage`, type:

```
copyfile(fullfile('..','ObjectRefDemoComp','OptimDemojavaPackage','OptimDemo.jar'))
```

- If you used the Library Compiler, type:

```
copyfile(fullfile('..','ObjectRefDemoComp','OptimDemo','for_testing','OptimDemo.jar'))
```

8 Open a command prompt window and navigate to the `ObjectRefDemoJavaApp` folder where you copied `OptimDemo.jar`.

9 Compile the `PerformOptim.java` application and `BananaFunction.java` helper class using `javac`.

- **Windows**

  To compile `BananaFunction.java`, type:

```
javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\OptimDemo.jar BananaFunction.java
```

  To compile `PerformOptim.java`, type:

```
javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\OptimDemo.jar PerformOptim.java
```

- **UNIX**

  To compile `BananaFunction.java`, type:

```
javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./OptimDemo.jar BananaFunction.java
```

  To compile `PerformOptim.java`, type:

```
javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./OptimDemo.jar PerformOptim.java
```

Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Windows, the path may be `C:\Program Files\MATLAB\R2021b`.

10 Run the `PerformOptim` application.

On Windows, type:

```
java -classpath .;"matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\OptimDemo.jar PerformOptim -1.2 1.0
```

On Linux, type:

```
java -classpath .:"matlabroot/toolbox/javabuilder/jar/javabuilder.jar":.\OptimDemo.jar PerformOptim -1.2 1.0
```

Note If you are running the application on the Mac 64-bit platform, you must add the `-d64` flag in the Java command.

The `PerformOptim` program displays 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
```

See Also

`libraryCompiler` | `compiler.build.javaPackage`

Related Examples

- “Block Console Display When Creating Figures in Java” on page 2-40

Use MATLAB Class in Java Application

In this section...

Overview

This example shows you how to create a Java application that calls MATLAB wrapper functions for a MATLAB class.

In this example, you perform the following steps:

1. Use MATLAB Compiler SDK to create a package that uses MATLAB wrapper functions to access a MATLAB class.
2. Call the MATLAB wrapper functions in a Java application.
3. Build and run the application.

Procedure

1. In MATLAB, examine the MATLAB code that you want to package. For this example, create a MATLAB class named `MyMATLABClass.m` using the following code:

```
classdef MyMatlabClass < handle

    properties (Access = private)
        x % input variable
        y % input variable
        z % result variable
    end

    methods
        function this = MyMatlabClass()
            this.x = []; this.y = [];
        end

        function setInput(this, input)
            input = input(:);
            if isnumeric(input) && numel(input) == 2
                this.x = input(1);
                this.y = input(2);
            end
        end

        function result = getResult(this)
            result = this.z;
        end

        function status = compute(this)
            try
                this.z = (this.x.^2 + this.y.^2)^0.5;
                status = true;
            catch
```

```
                status = false;
            end
        end
    end

end
```

2 Create four MATLAB wrapper functions for the class: `CreateMyMATLABClass.m`, `SetInput.m`, `Compute.m`, and `GetResult.m`.

CreateMyMATLABClass.m.

```
function instance = CreateMyMATLABClass()
    instance = MyMATLABClass();
end
```

3 Build the Java package with the **Library Compiler** app or `compiler.build.javaPackage` using the following information:

Field	Value
Library Name	`MyMATLABClass1`
Class Name	`Class1`

Files to Compile	`CreateMyMATLABClass.m`
	`SetInput.m`
	`Compute.m`
	`GetResult.m`

For example, if you are using `compiler.build.javaPackage`, type:

```
buildResults = compiler.build.javaPackage(["CreateMyMATLABClass.m", ...
"SetInput.m","Compute.m","GetResult.m"], ...
'PackageName','MyMATLABClass1', ...
'ClassName','Class1');
```

For more details, see the instructions in “Generate Java Package and Build Java Application”.

Note You do not need to manually add the `MyMATLABClass.m` file to the package, as the compiler automatically includes it during dependency analysis.

---

4 Navigate to the folder that contains the generated `MyMATLABClass1.jar` package. If you used the Library Compiler, the package is in the `for_testing` folder.

5 Write source code for an application that accesses the MATLAB functions. The code for this example is provided below.

`javadriver.java`

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

class javadriver
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("--- USE: Constructors ---");
            /* Instantiate a new Java object */
```

```
            /* This should only be done once per application instance */
            MyMATLABClass1.Class1 obj = new MyMATLABClass1.Class1();

            Object[] a = obj.CreateMyMATLABClass(1);
            obj.SetInput(a[0],new double[]{1,2});
            Object[] b = obj.Compute(1,a[0]);
            System.out.println( (MWLogicalArray) b[0]);
            Object[] c = obj.GetResult(1, a[0]);
            System.out.println((MWNumericArray)c[0]);
            System.out.println("--- Done. ---");
        }
        catch (Exception e)
        {
        System.out.println("Exception: " + e.toString());
        }
        finally
        {
        }
    }
}
```

6 Open a command prompt window and navigate to the folder that contains `javadriver.java` and `MyMATLABClass.jar`.

7 Compile the `javadriver.java` application using `javac`.

- On Windows, type:

  ```
  javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\MyMATLABClass.jar javadriver.java
  ```

- On UNIX, type:

  ```
  javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./MyMATLABClass.jar javadriver.java
  ```

  Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Linux, the path may be `/usr/local/MATLAB/R2021b`.

8 Run the `javadriver` application.

  On Windows, type:

  ```
  java -classpath .;"matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\MyMATLABClass.jar javadriver
  ```

  On Linux, type:

  ```
  java -classpath .:"matlabroot/toolbox/javabuilder/jar/javabuilder.jar":.\MyMATLABClass.jar javadriver
  ```

  Note If you are running the application on the Mac 64-bit platform, you must add the `-d64` flag in the Java command.

The `javadriver` program displays the following output:

```
--- USE: Constructors ---
1
2.2361
--- Done. ---
```

See Also

`libraryCompiler` | `compiler.build.javaPackage`

Related Examples

- “Pass Java Objects to MATLAB” on page 5-22

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

Render MATLAB Image Data in Java

This section contains code snippets intended to demonstrate specific functionality related to working with figure and image data.

Working with Images

Get Encoded Image Bytes from Image in 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);
    }
}
```

Get Buffered Image in 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();
    }
}
```

Create Buffered Images from 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.

  See `renderArrayData` in the Java API documentation for information on input parameters, return values, exceptions thrown, and examples.

Creating Scalable Web Applications Using RMI

Remote Method Invocation for Client-Server Applications

You can expand your application's throughput capacity by taking advantage of Remote Method Invocation (RMI), the Java native remote procedure call (RPC) mechanism. The way MATLAB Compiler SDK implements RMI technology to automatically generate interface code that enables 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.

See Also

Related Examples

- “Run Client and Server Using RMI” on page 7-3
- “Represent Native Java Cell and Struct Arrays” on page 7-7

Run Client and Server Using RMI

This example shows how to implement RMI to run two separate processes that initialize MATLAB struct arrays. The client and the server run on the same machine.

To implement RMI with a client on one machine and a server on another, use the procedure in this example and:

1. Change how the server is bound to the system registry.
2. Redefine how the client accesses the server.

RMI Prerequisites

To run this example, your environment must meet the following prerequisites:

- Install MATLAB Compiler SDK on the development machine.
- Install a supported version of the Java Development Kit (JDK) on the development machine. For more information, see “Configure Your Java Environment for Generating Packages” on page 1-3.
- Install MATLAB Runtime on the web server. For details, see “Install and Configure MATLAB Runtime”.
- Ensure that your web server is capable of running accepted Java frameworks like J2EE.
- Install the `javabuilder.jar` library (*`matlabroot`*`/toolbox/javabuilder/jar/javabuilder.jar`) into your web server’s common library folder.

  If your implementation uses separate client machines, they also need `javabuilder.jar`, since it contains the `com.mathworks.extern.java` package.

Note You do not need MATLAB Runtime installed on the client side. Return values from MATLAB Runtime can be automatically converted using the `boolean marshalOutputs` in the `RemoteProxy` class. For details, see the Javadoc API documentation in *`matlabroot`*`/help/toolbox/javabuilder/MWArrayAPI`.

Files

MATLAB Function Location	*`matlabroot`*`\toolbox\javabuilder\Examples\RMIExamples\DataTypes\DataTypesDemoComp`
Java Code Location	*`matlabroot`*`\toolbox\javabuilder\Examples\RMIExamples\DataTypes\DataTypesDemoJavaApp`

Procedure

1. Copy the `DataTypes` folder from MATLAB to your work folder:

   ```
   copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','RMIExamples','DataTypes'))
   ```

   At the MATLAB command prompt, navigate to the new `DataTypes\DataTypesDemoComp` subfolder in your work folder.
2. Examine the MATLAB functions `createEmptyStruct.m` and `updateField.m`.

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');
```

3 Generate the Java package using `compiler.build.javaPackage` by issuing the following command at the MATLAB command prompt:

```
compiler.build.javaPackage({'createEmptyStruct.m','updateField.m'}, ...
    'PackageName','dataTypesComp', ...
    'ClassName','dataTypesClass', ...
    'Verbose','on');
```

For more details, see the instructions in “Generate Java Package and Build Java Application”.

4 At your system command prompt, navigate to the `DataTypes\DataTypesDemoJavaApp` folder.

Compile the server Java code by issuing one of the following `javac` commands at your system command prompt.

- On Windows, type:

```
javac -classpath
 "matlabroot\toolbox\javabuilder\jar\javabuilder.jar;path\to\dataTypesComp.jar"
 DataTypesServer.java
```

- On UNIX, type:

```
javac -classpath
 "matlabroot/toolbox/javabuilder/jar/javabuilder.jar:path/to/dataTypesComp.jar"
 DataTypesServer.java
```

Note Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder.

5 Compile the client Java code by issuing one of the following `javac` commands at your system command prompt.

- On Windows, type:

```
javac -classpath
 "matlabroot\toolbox\javabuilder\jar\javabuilder.jar;path\to\dataTypesComp.jar
 DataTypesClient.java
```

- On UNIX, type:

```
javac -classpath
 "matlabroot/toolbox/javabuilder/jar/javabuilder.jar:path/to/dataTypesComp.jar"
 DataTypesClient.java
```

Run Client and Server

Run the client and server as follows:

1. Open two command windows—one for the server and one for the client.
2. In each window, navigate to the folder that contains `DataTypesServer.java` or `DataTypesClient.java`, respectively.
3. Run the server by issuing one of the following `java` commands in a single line at the system command prompt.

   - On Windows, type:

   ```
   java -classpath
    .;"path\to\dataTypesComp.jar;matlabroot\toolbox\javabuilder\jar\javabuilder.jar"
    -Djava.rmi.server.codebase="file:///matlabroot\toolbox\javabuilder\jar\javabuilder.jar
    file:///path\to\dataTypesComp.jar" DataTypesServer
   ```

   - On UNIX, type:

   ```
   java -classpath
    .:"path/to/dataTypesComp.jar;matlabroot/toolbox/javabuilder/jar/javabuilder.jar"
    -Djava.rmi.server.codebase="file:///matlabroot/toolbox/javabuilder/jar/javabuilder.jar
    file:///path/to/dataTypesComp.jar" DataTypesServer
   ```

4. In the second command window, run the client by issuing one of the following `java` commands in a single line.

   - On Windows, type:

   ```
   java -classpath
    .;"path/to/\dataTypesComp.jar;matlabroot\toolbox\javabuilder\jar\javabuilder.jar"
    DataTypesClient
   ```

   - On UNIX, type:

   ```
   java -classpath
    .:"path/to/dataTypesComp.jar;matlabroot/toolbox/javabuilder/jar/javabuilder.jar"
    -Djava.rmi.server.codebase="file:///matlabroot/toolbox/javabuilder/jar/javabuilder.jar
    file:///path/to/dataTypesComp.jar" DataTypesClient
   ```

If the commands are 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]

##################################
```

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'

##################################
```

Note For more examples of RMI implementation, see the files in *matlabroot*/toolbox/javabuilder/Examples/RMIExamples.

See Also

Related Examples

- “Remote Method Invocation for Client-Server Applications” on page 7-2
- “Represent Native Java Cell and Struct Arrays” on page 7-7

Represent Native Java Cell and Struct Arrays

Java has 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 multidimensional `Object` array, which can be difficult to process.

When you use MATLAB Compiler SDK packages with RMI, 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.

You can use Java native representations of MATLAB struct and cell arrays if both of these conditions are true:

- You have MATLAB functions on a server with MATLAB struct or cell data types as inputs or outputs
- You do not want to install MATLAB Runtime on your client machines

The classes in the `com.mathworks.extern.java` package (in `javabuilder.jar`) do not need MATLAB Runtime. The names of the classes in this package are the same as those in `com.mathworks.toolbox.javabuilder` — allowing you 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 MATLAB Runtime.

Since the `MWArray` class hierarchy can be used only with MATLAB Runtime, if the client machine does not have MATLAB Runtime available, the server returns the output of `toArray()` for struct or cell arrays as instances of `com.mathworks.extern.java.MWStructArray` or `com.mathworks.extern.java.MWCellArray`, respectively.

Prerequisites

To run this example, your environment must meet the following prerequisites:

- Install MATLAB Compiler SDK on the development machine.
- Install a supported version of the Java Development Kit (JDK) on the development machine. For more information, see “Configure Your Java Environment for Generating Packages” on page 1-3.
- Install MATLAB Runtime on the web server. For details, see “Install and Configure MATLAB Runtime”.
- Ensure that your web server is capable of running accepted Java frameworks like J2EE.
- Install the `javabuilder.jar` library (*`matlabroot`*`/toolbox/javabuilder/jar/javabuilder.jar`) into your web server’s common library folder.

  If your implementation has separate client machines, they also need `javabuilder.jar`, since it contains the `com.mathworks.extern.java` package.

Note You do not need MATLAB Runtime installed on the client side. Return values from the MATLAB Runtime can be automatically converted using the `boolean marshalOutputs` in the `RemoteProxy` class. For details, see the Java API documentation in *`matlabroot`*`/help/toolbox/javabuilder/MWArrayAPI`.

Procedure

<listitem>
Copy the `NativeCellStruct` folder from MATLAB to your work folder:

```
copyfile(fullfile(matlabroot,'toolbox','javabuilder','Examples','RMIExamples','NativeCellStruct'))
```

At the MATLAB command prompt, navigate to the new `NativeCellStruct\NativeCellStructDemoComp` subfolder in your work folder.
</listitem>
<listitem>
Examine the MATLAB functions `createEmptyStruct.m` and `updateField.m`.

createEmptyStruct.m

```
function PartialStruct = createEmptyStruct(field_names)

fprintf('EVENT 1: Initializing the structure on server and sending it to client:\n');

PartialStruct = struct(field_names{1},' ',field_names{2},[]);

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 server:\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 client:\n\n']);
st.(field_name) = 'MathWorks';
FinalStruct = st;
disp(FinalStruct);
fprintf('\n###################################\n');
```

</listitem>
<listitem>
Generate the Java package using `compiler.build.javaPackage` by issuing the following command at the MATLAB command prompt:

```
compiler.build.javaPackage({'createEmptyStruct.m','updateField.m'}, ...
    'PackageName','nativeCellStructComp', ...
    'ClassName','nativeCellStructClass', ...
    'Verbose','on');
```

For more details, see the instructions in “Generate Java Package and Build Java Application”.
</listitem>
<listitem>
At your system command prompt, navigate to the `NativeCellStruct\NativeCellStructDemoJavaApp` folder.

Compile the server Java code by issuing one of the following `javac` commands at your system command prompt.

- On Windows, type:

  ```
  javac -classpath
   "matlabroot\toolbox\javabuilder\jar\javabuilder.jar;path\to\nativeCellStructComp.jar"
   NativeCellStructServer.java
  ```

- On UNIX, type:

```
javac -classpath
 "matlabroot/toolbox/javabuilder/jar/javabuilder.jar:path/to/nativeCellStructComp.jar"
 NativeCellStructServer.java
```

Note Replace *matlabroot* with the path to your MATLAB or MATLAB Runtime installation folder.

</listitem>
<listitem>
Compile the client Java code by issuing one of the following `javac` commands at your system command prompt.

- On Windows, type:

```
javac -classpath
 "matlabroot\toolbox\javabuilder\jar\javabuilder.jar;path\to\dataTypesComp.jar
 NativeCellStructClient.java
```

- On UNIX, type:

```
javac -classpath
 "matlabroot/toolbox/javabuilder/jar/javabuilder.jar:path/to/dataTypesComp.jar"
 NativeCellStructClient.java
```

</listitem>
<listitem>
Prepare to run the server and client applications by opening two command windows—one for the client and one for the server.
</listitem>
<listitem>
Run the server by issuing one of the following `java` commands in a single line at the system command prompt.

- On Windows, type:

```
java -classpath
 .;"path\to\dataTypesComp.jar;matlabroot\toolbox\javabuilder\jar\javabuilder.jar"
 -Djava.rmi.server.codebase="file:///matlabroot\toolbox\javabuilder\jar\javabuilder.jar
 file:///path\to\dataTypesComp.jar"
 NativeCellStructServer
```

- On UNIX, type:

```
java -classpath
 .:"path/to/dataTypesComp.jar;matlabroot/toolbox/javabuilder/jar/javabuilder.jar"
 -Djava.rmi.server.codebase="file:///matlabroot/toolbox/javabuilder/jar/javabuilder.jar
 file:///path/to/dataTypesComp.jar"
 NativeCellStructServer
```

</listitem>
<listitem>
In the second command window, run the client by issuing one of the following `java` commands in a single line.

- On Windows, type:

```
java -classpath
 .;"path/to/\dataTypesComp.jar;matlabroot\toolbox\javabuilder\jar\javabuilder.jar"
 NativeCellStructClient
```

- On UNIX, type:

```
java -classpath
 .:"path/to/dataTypesComp.jar;matlabroot/toolbox/javabuilder/jar/javabuilder.jar"
 -Djava.rmi.server.codebase="file:///matlabroot/toolbox/javabuilder/jar/javabuilder.jar
 file:///path/to/dataTypesComp.jar"
 NativeCellStructClient
```

</listitem>

If the commands are 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]


            ###################################
```

The following output appears in the command window running the client:

```
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

################################
```

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` or `DYLD_LIBRARY_PATH` variable is not set. For more information, see “Set MATLAB Runtime Path for Deployment”.

`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 throwing 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 and Limitations of 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:

- You must have MATLAB Compiler SDK installed.
- Your Java environment must be properly configured. For details, see “Configure Your Java Environment for Generating Packages” on page 1-3.
- Your end users must have MATLAB or MATLAB Runtime installed to run compiled MATLAB code.

Limitations of MATLAB Compiler SDK Java Target

Consider the following limitations when creating Java packages using MATLAB Compiler SDK:

- Special characters in MATLAB comments can cause compilation to fail. Remove special characters or replace them with XML characters. For example, "<" can be replaced with "<".
- JAR files created by MATLAB Compiler SDK cannot be loaded back into MATLAB with the MATLAB Java External Interface.

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.

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

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

Note When you call 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 by 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!
  ```

  The correct usage is:

  ```
  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 most other cases, 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`
	`)`

The following table shows details about the arguments for these samples of standard signatures.

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

Consider 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, consider 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 below to show overloaded implementations of the `myprimes` method in the Java code.

```
/* 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)
   }
```

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.

For details about the interfaces, see “Programming Interfaces Generated by MATLAB Compiler SDK” on page 9-8.

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.

10

Functions

compiler.build.javaPackage

Create Java package for deployment outside MATLAB

Syntax

```
compiler.build.javaPackage(Files)
compiler.build.javaPackage(Files,Name,Value)
compiler.build.javaPackage(ClassMap)
compiler.build.javaPackage(ClassMap,Name,Value)
compiler.build.javaPackage(opts)
results = compiler.build.javaPackage( ___ )
```

Description

`compiler.build.javaPackage(Files)` creates a Java package using the MATLAB functions specified by `Files`. Before creating Java packages, see Configure Your Java Environment on page 1-3.

`compiler.build.javaPackage(Files,Name,Value)` creates a Java package with additional options specified using one or more name-value arguments. Options include the class name, output directory, and additional files to include.

`compiler.build.javaPackage(ClassMap)` creates a Java package with a class mapping specified using a `container.Map` object `ClassMap`.

`compiler.build.javaPackage(ClassMap,Name,Value)` creates a Java package using `ClassMap` and additional options specified using one or more name-value arguments. Options include the package name, output directory, and additional files to include.

`compiler.build.javaPackage(opts)` creates a Java package with options specified using a `compiler.build.JavaPackageOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.javaPackage( ___ )` returns build information as a `compiler.build.Results` object using any of the input argument combinations in previous syntaxes. The build information consists of the build type, paths to the compiled files, and build options.

Examples

Create Java Package Using File Input

Create a Java package using a function file that generates a magic square.

In MATLAB, locate the MATLAB function that you want to deploy as a Java package. For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Build a Java package using the `compiler.build.javaPackage` command.

```
compiler.build.javaPackage(appFile);
```

This syntax generates the following within a folder named `magicsquarejavaPackage` in your current working directory:

- `classes` — Folder that contains the Java class files and the deployable archive file.
- `doc` — Folder that contains HTML documentation for all classes in the package.
- `examples` — Folder that contains Java source code files.
- `GettingStarted.html` — File that contains information on integrating your package.
- `includedSupportPackages.txt` — Text file that lists all support files included in the package.
- `magicsquare.jar` — Java archive file.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see Functions Not Supported For Compilation.
- `readme.txt` — Readme file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Customize Java Package

Create a Java package and customize it using name-value arguments.

For this example, use the files `flames.m` and `flames.mat` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','flames.m');
MATFile = fullfile(matlabroot,'extern','examples','compiler','flames.mat');
```

Build a Java package using the `compiler.build.javaPackage` command. Use name-value arguments to specify the package name, add a MAT-file, and enable verbose output.

```
compiler.build.javaPackage(appFile,'PackageName','JavaFlames',...
    'AdditionalFiles',MATFile,...
    'Verbose','on');
```

Create Java Package Using Class Map Input

Create a Java package using a class map and multiple MATLAB functions.

Create a `containers.Map` object whose keys are class names and whose values are the locations of function files.

```
cmap = containers.Map;
cmap('Class1') = {'exampleFcn1.m','exampleFcn2.m'};
cmap('Class2') = {'exampleFcn3.m','exampleFcn4.m'};
```

Build a Java package using the `compiler.build.javaPackage` command.

```
compiler.build.javaPackage(cmap);
```

You can also specify options using name-value arguments when you build the Java package.

```
compiler.build.javaPackage(cmap,...
    'PackageName','ExamplePackage',...
    'Verbose','on');
```

Customize Multiple Components Using Options Object

Customize multiple Java packages using a `compiler.build.JavaPackageOptions` object on a Windows system to specify a common output directory, use debug symbols, and enable verbose output.

For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Create a `JavaPackageOptions` object using `appFile` and additional options specified using name-value arguments.

```
opts = compiler.build.JavaPackageOptions(appFile,...
    'OutputDir','D:\Documents\MATLAB\work\JavaPackageBatch',...
    'DebugBuild','on',...
    'Verbose','on')
```

```
opts =

  JavaPackageOptions with properties:

              ClassMap: [1×1 containers.Map]
           DebugBuild: on
          PackageName: 'example.magicsquare'
  SampleGenerationFiles: {}
        AdditionalFiles: {}
    AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
                Verbose: on
             OutputDir: 'D:\Documents\MATLAB\work\JavaPackageBatch'

   Class Map Information
       magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler\magicsquare.m
```

Build the Java package using the `JavaPackageOptions` object.

```
compiler.build.javaPackage(opts);
```

To compile using the function file `hello.m` with the same options, use dot notation to modify the `ClassMap` of the existing `JavaPackageOptions` object before running the build function again.

```
remove(opts.ClassMap, keys(opts.ClassMap));
opts.ClassMap('helloClass') = fullfile(matlabroot,'extern','examples','compiler','hello.m');
compiler.build.javaPackage(opts);
```

By modifying the `ClassMap` argument and recompiling, you can compile multiple components using the same options object.

Get Build Information from Java Package

Create a Java package and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
results = compiler.build.javaPackage('magicsquare.m')
```

```
results =

  Results with properties:

              BuildType: 'javaPackage'
                  Files: {3×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.JavaPackageOptions]
```

The `Files` property contains the paths to the following:

- `doc` folder
- `magicsquare.jar`
- `GettingStarted.html`

Input Arguments

`Files` — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m","myfunc2.m"]`

Data Types: `char` | `string` | `cell`

`ClassMap` — Class map

`containers.Map` object

Class map, specified as a `containers.Map` object. Map keys are class names and each value is the set of files mapped to the corresponding class. Files must have a `.m` extension.

Example: `cmap`

`opts` — Java package build options

`compiler.build.JavaPackageOptions` object

Java package build options, specified as a `compiler.build.JavaPackageOptions` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Verbose','on'`

`AdditionalFiles` — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files to include in the Java package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myvars.mat","data.txt"]`

Data Types: `char` | `string` | `cell`

`AutoDetectDataFiles` — Flag to automatically include data files
`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the Java package.
- If you set this property to `'off'`, then you must add data files to the package using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles','off'`

Data Types: `logical`

`ClassName` — Name of Java class
character vector | string scalar

Name of the Java class, specified as a character vector or a string scalar. You cannot specify this option if you use a `ClassMap` input. Class names must meet Java class name requirements.

The default value is the name of the first file listed in the `Files` argument appended with `Class`.

Example: `'ClassName','magicsquareClass'`

Data Types: `char` | `string`

`DebugBuild` — Flag to enable debug symbols
`'off'` (default) | on/off logical value

Flag to enable debug symbols, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the compiled package contains debug symbols.
- If you set this property to `'off'`, then the compiled package does not contain debug symbols.

Example: `'DebugBuild','on'`

Data Types: `logical`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the package name appended with `javaPackage`.

Example: `'OutputDir','D:\Documents\MATLAB\work\mymagicjavaPackage'`

Data Types: `char` | `string`

`PackageName` — Name of Java package
character vector | string scalar

Name of the Java package, specified as a character vector or a string scalar. Specify `'PackageName'` as a namespace, which is a period-separated list, such as `companyname.groupname.component`. The name of the generated package is set to the last entry of the period-separated list. The name must begin with a letter and contain only alphabetic characters and periods.

Example: `'PackageName','mathworks.javapackage.mymagic'`

Data Types: `char` | `string`

`SampleGenerationFiles` — MATLAB sample files
character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample Java driver files for functions included within the package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `'SampleGenerationFiles',["sample1.m","sample2.m"]`

Data Types: `char` | `string` | `cell`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char` | `string` | `cell`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control build verbosity, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to `'off'`, then the command window does not display progress information.

Example: `'Verbose','on'`

Data Types: `logical`

Output Arguments

`results` — Build results

`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The `Results` object contains:

- Build type, which is `'javaPackage'`
- Paths to the compiled files
- A list of included support packages
- Build options, specified as a `JavaPackageOptions` object

See Also

`compiler.build.JavaPackageOptions`

Introduced in R2021a

compiler.build.JavaPackageOptions

Options for building Java packages

Syntax

```
opts = compiler.build.JavaPackageOptions(Files)
opts = compiler.build.JavaPackageOptions(Files,Name,Value)
opts = compiler.build.JavaPackageOptions(ClassMap)
opts = compiler.build.JavaPackageOptions(ClassMap,Name,Value)
```

Description

`opts = compiler.build.JavaPackageOptions(Files)` creates a `JavaPackageOptions` object using MATLAB functions specified by `Files`. Use the `JavaPackageOptions` object as an input to the `compiler.build.javaPackage` function.

`opts = compiler.build.JavaPackageOptions(Files,Name,Value)` creates a `JavaPackageOptions` object with options specified using one or more name-value arguments. Options include the package name, output directory, and additional files to include.

`opts = compiler.build.JavaPackageOptions(ClassMap)` creates a `JavaPackageOptions` object with a class mapping specified using a `container.Map` object `ClassMap`.

`opts = compiler.build.JavaPackageOptions(ClassMap,Name,Value)` creates a `JavaPackageOptions` object with a class mapping specified using `ClassMap` and options specified using one or more name-value arguments.

Examples

Create Java Package Options Object Using File

Create a `JavaPackageOptions` object using file input.

For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.JavaPackageOptions(appFile)
```

```
opts =

  JavaPackageOptions with properties:

                ClassMap: [1×1 containers.Map]
             DebugBuild: off
            PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
```

```
                Verbose: off
              OutputDir: '.\magicsquarejavaPackage'

   Class Map Information
         magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler'}
```

You can modify the property values of an existing `JavaPackageOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =

  JavaPackageOptions with properties:

               ClassMap: [1×1 containers.Map]
             DebugBuild: off
            PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
                Verbose: on
              OutputDir: '.\magicsquarejavaPackage'

   Class Map Information
         magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler'}
```

Use the `JavaPackageOptions` object as an input to the `compiler.build.javaPackage` function to build a Java package.

```
buildResults = compiler.build.javaPackage(opts);
```

Customize Java Package Options Object

Create a `JavaPackageOptions` object and customize it using name-value arguments.

For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`. Use name-value arguments to specify the output directory and disable automatic detection of data files.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.JavaPackageOptions(appFile,...
    'OutputDir','D:\Documents\MATLAB\work\MagicJavaPackage',...
    'AutoDetectDataFiles','off')
```

```
opts =

  JavaPackageOptions with properties:

               ClassMap: [1×1 containers.Map]
             DebugBuild: off
            PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
```

```
                Verbose: off
              OutputDir: 'D:\Documents\MATLAB\work\MagicJavaPackage'

   Class Map Information
         magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler'}
```

You can modify the property values of an existing `JavaPackageOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =

  JavaPackageOptions with properties:

               ClassMap: [1×1 containers.Map]
             DebugBuild: off
            PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
                Verbose: on
              OutputDir: 'D:\Documents\MATLAB\work\MagicJavaPackage'

   Class Map Information
         magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler'}
```

Use the `JavaPackageOptions` object as an input to the `compiler.build.javaPackage` function to build a Java package.

```
buildResults = compiler.build.javaPackage(opts);
```

Create Java Package Options Object Using Class Map

Create a `JavaPackageOptions` object using a class map.

Create a `containers.Map` object whose keys are class names and whose values are MATLAB function files.

```
cmap = containers.Map;
cmap('Class1') = {'exampleFcn1.m','exampleFcn2.m'};
cmap('Class2') = {'exampleFcn3.m','exampleFcn4.m'};
```

Create the `JavaPackageOptions` object using the class map `cmap`.

```
opts = compiler.build.JavaPackageOptions(cmap)
```

```
opts =

  JavaPackageOptions with properties:

               ClassMap: [1×1 containers.Map]
             DebugBuild: off
            PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
```

```
                Verbose: off
              OutputDir: '.\magicsquarejavaPackage'

  Class Map Information
                 Class1: {2×1 cell}
                 Class2: {2×1 cell}
```

You can also create a `JavaPackageOptions` object using name-value arguments or modify an existing object using dot notation. For this example, specify an output directory, enable verbose output, and disable automatic detection of data files.

```
opts = compiler.build.JavaPackageOptions(cmap,...
    'OutputDir','D:\Documents\MATLAB\work\MagicJavaPackage',...
    'Verbose','On');
opts.AutoDetectDataFiles = 'off';

opts =

  JavaPackageOptions with properties:

                ClassMap: [1×1 containers.Map]
              DebugBuild: off
             PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: off
          SupportPackages: {'autodetect'}
                  Verbose: on
                OutputDir: 'D:\Documents\MATLAB\work\MagicJavaPackage'

  Class Map Information
                 Class1: {2×1 cell}
                 Class2: {2×1 cell}
```

Use the `JavaPackageOptions` object as an input to the `compiler.build.javaPackage` function to build a Java package.

```
buildResults = compiler.build.javaPackage(opts);
```

Input Arguments

`Files` — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m","myfunc2.m"]`

Data Types: `char` | `string` | `cell`

`ClassMap` — Class map

`containers.Map` object

Class map, specified as a `containers.Map` object. Map keys are class names and each value is the set of files mapped to the corresponding class. Files must have a `.m` extension.

Example: `cmap`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Verbose','on'`

`AdditionalFiles` — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files to include in the Java package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myvars.mat","data.txt"]`

Data Types: `char` | `string` | `cell`

`AutoDetectDataFiles` — Flag to automatically include data files

`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the Java package.
- If you set this property to `'off'`, then you must add data files to the package using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles','off'`

Data Types: `logical`

`ClassName` — Name of Java class

character vector | string scalar

Name of the Java class, specified as a character vector or a string scalar. You cannot specify this option if you use a `ClassMap` input. Class names must meet Java class name requirements.

The default value is the name of the first file listed in the `Files` argument appended with `Class`.

Example: `'ClassName','magicsquareClass'`

Data Types: `char` | `string`

`DebugBuild` — Flag to enable debug symbols

`'off'` (default) | on/off logical value

Flag to enable debug symbols, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the compiled package contains debug symbols.
- If you set this property to `'off'`, then the compiled package does not contain debug symbols.

Example: `'DebugBuild','on'`

Data Types: `logical`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the package name appended with `javaPackage`.

Example: `'OutputDir','D:\Documents\MATLAB\work\mymagicjavaPackage'`

Data Types: `char` | `string`

`PackageName` — Name of Java package
character vector | string scalar

Name of the Java package, specified as a character vector or a string scalar. Specify `'PackageName'` as a namespace, which is a period-separated list, such as `companyname.groupname.component`. The name of the generated package is set to the last entry of the period-separated list. The name must begin with a letter and contain only alphabetic characters and periods.

Example: `'PackageName','mathworks.javapackage.mymagic'`

Data Types: `char` | `string`

`SampleGenerationFiles` — MATLAB sample files
character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample Java driver files for functions included within the package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `'SampleGenerationFiles',["sample1.m","sample2.m"]`

Data Types: `char` | `string` | `cell`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char` | `string` | `cell`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control build verbosity, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to `'off'`, then the command window does not display progress information.

Example: `'Verbose','on'`

Data Types: `logical`

Output Arguments

`opts` — Java package build options
`JavaPackageOptions` object

Java package build options, returned as a `JavaPackageOptions` object.

See Also

`compiler.build.javaPackage`

Introduced in R2021a

mcrinstaller

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

Syntax

```
[installer_path, major, minor, platform] = mcrinstaller
```

Description

`[installer_path, major, minor, platform] = mcrinstaller` displays information about available MATLAB Runtime installers.

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

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 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
```

Output Arguments

`installer_path` — Full path to the installer
character vector

The `installer_path` is the full path to the installer for the current platform.

`major` — Major version number
positive integer scalar

The `major` is the major version number of the installer.

`minor` — Minor version number
positive integer scalar

The `minor` is the minor version number of the installer.

`platform` — Name of the current platform
character vector

The `platform` is the name of the current platform (returned by `COMPUTER(arch)`).

See Also

`mcrversion` | `compiler.runtime.download`

Topics
“Install and Configure MATLAB Runtime”

mcrversion

Return MATLAB Runtime version number that matches MATLAB version

Syntax

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

Description

`[major,minor] = mcrversion` returns the MATLAB Runtime version number matching the version of MATLAB from where the command is executed. 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`.

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.

Examples

Return the MATLAB Runtime Version

Return the MATLAB Runtime Version Number Matching the Version of MATLAB.

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

```
major =
     9
minor =
     9
```

Output Arguments

`major` — Major version number
positive integer scalar

Major version number returned as a positive integer scalar.

Data Types: `double`

`minor` — Minor version number
positive integer scalar

Minor version number returned as a positive integer scalar.

Data Types: `double`

See Also

`compiler.runtime.download` | `mcrinstaller`

Topics

“Install and Configure MATLAB Runtime”

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

“Block Console Display When Creating Figures in Java” on page 2-40

Introduced before R2006a