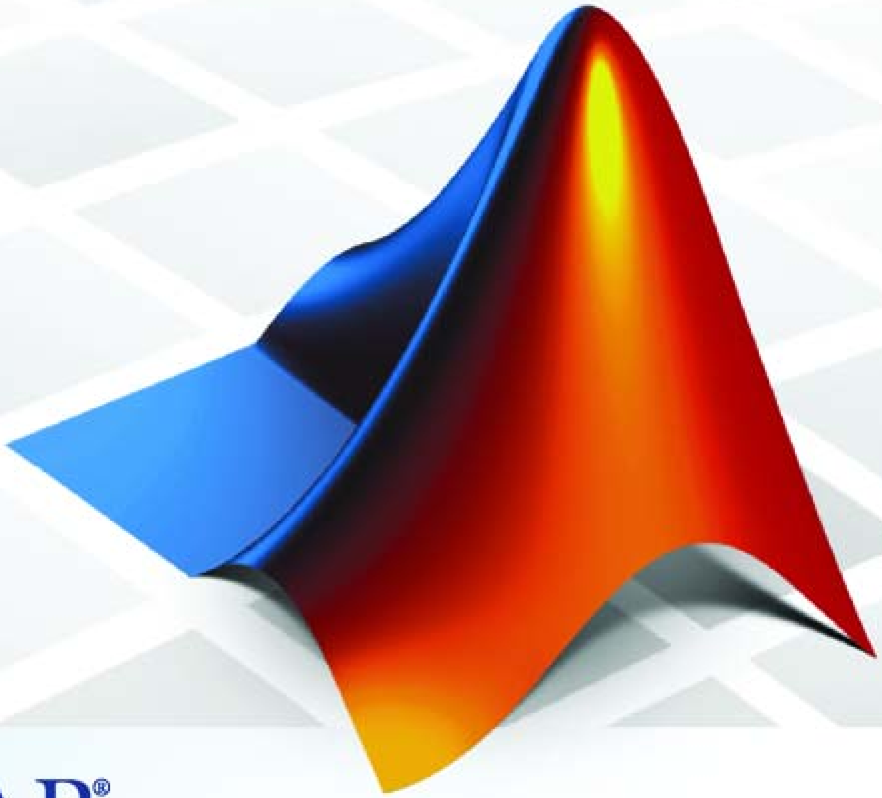
MATLAB® Builder for Java™ 2



MATLAB®

The MathWorks™

Accelerating the pace of engineering and science

How to Contact The MathWorks

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Builder for Java User's Guide

Revision History

September 2006	Online only	New for Version 1.0 (Release 2006b)
March 2007	Online only	Revised for Version 1.1 (Release 2007a)
September 2007	Online only	Revised for Version 2.0 (Release 2007b)

Contents

1 Getting Started

2 Concepts

4 Using Classes and Methods

5 Sample Java Applications

7 Reference Information for Java

8 Functions — Alphabetical List

A Examples

Getting Started

What Is MATLAB Builder for Java?

MATLAB® Builder for Java (also called Java Builder) is an extension to MATLAB® Compiler. Use Java Builder to wrap MATLAB® functions into one or more Java classes that make up a Java component, or package. Each MATLAB function is encapsulated as a method of a Java class and can be invoked from within a Java application.

When you package and distribute the application to your users, supporting files generated by Java Builder are included as well as the MATLAB Component Runtime (MCR), which is provided by MATLAB Compiler. Your users do not have to purchase and install MATLAB.

Before You Begin

In this section...

What You Need To Know

The following technical knowledge is required to use MATLAB Builder for Java:

- Java Builder requires MATLAB. This documentation assumes that you know how to work with MATLAB cell arrays and structures.
- It is helpful to be familiar with the Java programming language, MS-DOS or UNIX command syntax, and Object Oriented programming concepts, but it is not necessary.
- MATLAB Builder for Java does not support MATLAB object data types (for example, Time Series objects).

Configuring Your Environment

Configure your environment to work with the supplied example files and set up your Java environment.

Setting Up Your Java Environment

1 Ensure your Java Runtime Environment and JDK are compatible with Sun Microsystems, Inc's JDK Version 1.6.0. The MathWorks recommends using the Java Runtime Environment (JRE) shipped with MATLAB (`*matlabroot*\sys\java\jre\*architecture*\jre.cfg`), where `*architecture*` is the name of the directory that is associated with the platform on which you run MATLAB, for example, `Win32` or `Win64`.

Download JDK Version 1.6 from Sun Microsystems, Inc. if you do not yet have it installed.

2 Set the `JAVA_HOME` variable to the location of your installed JDK (Java Developer's Kit).

   At the command prompt, enter:

   ```
   set JAVA_HOME=JDK_pathname
   ```

   where *`JDK_pathname`* is the path to your installed JDK. The JDK should be compatible with Sun JDK version 1.6.0.

   Alternately, you can use the `setenv` command from MATLAB. To do this, start MATLAB and issue the following at the MATLAB Command Prompt:

   ```
   setenv('JAVA_HOME','JDK_pathname')
   ```

   Note Optimally, `JAVA_HOME` should be defined as an environment variable to ensure maximum versatility and usability.

3 At the MATLAB command prompt, enter `getenv JAVA_HOME`. The answer returned by MATLAB should match the `JAVA_HOME` location you previously set.

Preparing To Use the Example Files

1 Copy the Java Builder examples to a work directory, including the M application `makesqr.m` and the Java demo application `getmagic.java`.

   a. Navigate to *`matlabroot`*`\toolbox\javabuilder\Examples\MagicSquareExample`. To determine what path *`matlabroot`* is set to, enter `matlabroot` at the MATLAB command line.

   b. Create a local work directory named `javabuilder_examples`. In this example, this directory is on the local Windows `D:` drive.

c. Create a subdirectory under `javabuilder_examples` named `magicsquare`.

d. Copy the contents of the `MagicSquareExample` directory to `D:\javabuilder_examples\magicsquare`.

2 Using a system command prompt, navigate to `\javabuilder_examples\magicsquare` by switching to the `D:` drive and entering:

```
cd \javabuilder_examples\magicsquare
```

Starting MATLAB Builder for Java

MATLAB Builder for Java is accessed through the Deployment Tool interface.

To open the Deployment Tool, start MATLAB and type `deploytool` at the command prompt in the MATLAB Command Window.

Overview of MATLAB Compiler

Use MATLAB® Compiler to convert MATLAB® programs to applications and libraries that you can distribute to end users who do not have MATLAB installed. You can compile M-files, MEX-files, or other MATLAB code. MATLAB Builder for Java supports all the features of MATLAB, including objects, private functions, and methods. Using MATLAB Builder for Java you can generate the following:

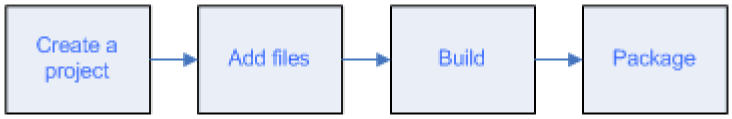
- Standalone C and C++ applications on UNIX, Windows, and Macintosh platforms
- C and C++ shared libraries (dynamically linked libraries, or DLLs, on Microsoft Windows)

Use the `mcc` command to invoke MATLAB Builder for Java. Alternatively, you can use the graphical user interface for MATLAB Builder for Java by issuing the following command at the MATLAB prompt:

```
deploytool
```

What Is the Deployment Tool?

The Deployment Tool is the GUI to MATLAB Compiler. Use the Deployment Tool to perform the tasks in the following illustration.



Essential Steps to Deploying a Java Builder Component

In this section...

Create a Deployable Java Builder Component

A Java Builder component is a self-contained Java application (`.jar` file) that can be deployed to other computers. Note that as of R2007b, the CTF file is now embedded in the JAR file for convenient deployment. For more information, see "How Does MATLAB Builder for Java Use JAR Files?" on page 2-4

To create your component, do the following:

1. Run the Build function of `deploytool` to encapsulate your M-code into a Java class.

2. Compile your Java program with the `javac` command.

3. Test your program by executing it with the `java` command.

Package Your Java Component

Packaging your Java component involves including the following key elements, in addition to your Java component, which are essential for deployment on other machines:

- The `.jar` file (which includes the CTF archive) contains all Java files needed to deploy the application.
- The MCR Installer (if the **Include MCR** option was selected when the component was built) is a self-contained runtime executable capable of running an instance of MATLAB on a computer that does not have MATLAB installed.

- Documentation generated by the Sun™ Microsystems Javadoc tool is included that details how the Java classes and methods referenced in your application can be further customized and includes detailed examples of implementation.

Distribute Your Java Component

Finally, make the component available to end users through one of the following mechanisms:

- Distribute the component privately using external storage media or e-mail.
- Make the component available over a local area network.
- Use the Web to remotely distribute the component.

Customize Your Java Component

Finally, customize your component before you distribute it to your end users. Some of the things you will want to do include:

- Importing Java classes and MATLAB libraries into existing code
- Creating an instance of, or instantiating, a Java class
- Calling class methods from Java

Deploying a Component

In this section...

Magic Square Example

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

The Magic Square example shows you how to create a Java component (`magicsquare`), which contains the `magic` class, a `.jar` file, and other files needed to deploy your application. The class encapsulates a MATLAB function, `makesqr`, which computes a magic square. It represents the magic square as a two-dimensional array.

The client Java application, `getmagic.java` converts the array returned by `makesqr` to a native array and displays it on the screen.

When you run the `getmagic` application from the command line, you can pass the dimension for the magic square as a command-line argument.

About the Examples The examples for MATLAB Builder for Java are in *`matlabroot`*`\toolbox\javabuilder\Examples`. In most examples, Windows syntax is featured (backslashes, instead of forward slashes).

The examples in this section utilize the `deploytool` command. For information about how to perform these examples using the `mcc` command, see "Using the Command-Line Interface" on page 7-17.

Create a Deployable Java Component

1 Create a new deployment project.

  a. Start the Deployment Tool if you have not already done so. See"Starting MATLAB Builder for Java" on page 1-5 for details.

  b. Click in the Deployment Tool toolbar. Alternatively you can select **File > New Deployment Project** from the MATLAB menu bar.

  c. In the navigation pane (the left pane), select **MATLAB Builder for Java** as the product you want to use to create the deployment project.

  d. From the component list (the right pane), select **Java Package** as the kind of component you want to create.

  e. Click **Browse** to select the location for your project. In this case, use `D:\javabuilder_examples\magicsquare`.

  f. Enter `magicsquare` as the project name and click **OK**. By default, the project name is also the component name.

2 Customize the project settings as needed.

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

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

  c. In the **Current Directory** pane of MATLAB, navigate to `D:\javabuilder_examples\magic_square\MagicDemoComp`.

  d. Add the `makesqr.m` file in the `MagicDemoComp` directory to the project by dragging this file from the **Current Directory** pane in the MATLAB desktop to the renamed `magic` folder in the **Deployment Tool** pane.

  e. In the **Deployment Tool** pane, ensure that the **Generate Verbose Output** option is selected.

  f. Save the project by clicking in the Deployment Tool toolbar.

3 Build the project, creating your initial Java component.

Build the project by clicking in the Deployment Tool toolbar.

The build process begins, and a log of the build is created. Files are generated in two directories, `src` and `distrib`, within the project directory. The paths to these directories are defined in the Deployment Project Settings dialog. A copy of the build log is placed in the `src` directory.

4. After you have built your Java class, examine it by navigating to the directory `D:\javabuilder_examples\magicsquare\magicsquare`, and inspect the contents of the newly created `src` and `distrib` directories.

   - The `src` directory contains the generated Java source.
   - The `distrib` directory contains a Java archive (`.jar`) file that will be distributed with your deployed application.

5. At the system command prompt, enter one of the following commands:

   Caution When entering these commands, ensure single spaces are inserted at the end of each line below.

   - On **Windows**:

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

   - On **UNIX**:

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

6. When you run `getmagic`, you pass an input argument representing the dimension for the magic square. In this example, the value for the dimension is 5. The program converts the number passed on the command line to a scalar double value, creates an instance of class `magic`, and calls

the `makesqr` method on that object. The method computes the square using the MATLAB `magic` function.

Run `getmagic` by doing the following:

a Copy `getmagic.class` to your work directory.

b Enter one of the following commands at the system command prompt:

- On **Windows**:

```
matlabroot\sys\java\jre\architecture\jre_directory\bin\java
 -classpath
.;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;.\magicsquare\distrib\magicsquare.jar
getmagic 5
```

- On **UNIX**:

```
matlabroot/sys/java/jre/architecture/jre_directory/bin/java
 -classpath
.:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:./magicsquare/distrib/magicsquare.jar
getmagic 5
```

Note *`jre_directory`* refers to the directory described in "Setting Up Your Java Environment" on page 1-3. To test directly against the MCR, substitute *`mcrroot`* for *`matlabroot`*, where *`mcrroot`* is the location where the MCR is installed on your system. An example of an MCR root location is: `D:\Applications\MATLAB\MATLAB Component Runtime\`*`MCR_version_number`*. Remember to double-quote all parts of the `java` command path arguments that contain spaces.

7 Verify the program output. If the program ran successfully, a magic square of order five will print, as follows:

```
Magic square of order 5
```

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

Package Your Java Application

When you package the application for your users, you must include supporting files generated by Java Builder as well as the MATLAB Component Runtime (MCR), which is provided by MATLAB Compiler. After you have built and packaged a few test Java components with the Magic Square example, follow this procedure to distribute your own applications to the rest of your enterprise computing environment.

For more details on these steps, see "Developing Your Application" on page 1-14.

Note You must repeat these steps for each development machine where you want to use the components.

Step 1 is optional if you are developing your application on the same machine where you created the Java component.

---

1. If the component is not already installed on the machine where you want to develop your application, unpack and install the component as follows:
   a. Copy the package that you created.
   b. If the package is a self-extracting executable, paste the package in a directory on the development machine, and run it. If the package is a `.zip` file, unzip and extract the contents to the development machine.
2. If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 7-2.
3. Import the MATLAB libraries and the component classes into your code with the Java `import` function. For example:

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

4. Instantiate a Java Builder class using the Java `new` operator to create an instance of each class you want to use in the application.

5. Call the class methods, mapping the names of your M-functions to Java methods.

6. Handle data conversion as needed.

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

   - To manually convert to one of the standard MATLAB data types, use MWArray classes in the package `com.mathworks.toolbox.javabuilder`.
   - If you pass them as Java data types, they are automatically converted.

7. Build and test the Java application as you would any application.

Developing Your Application

After you create and distribute the initial application, you will want to continue to enhance it. Use the following Java “basics”:

Importing Classes

You must import the MATLAB libraries and your own Java classes into your code. Use the Java `import` function to do this.

For the `magicsquare` example, the following statements perform the necessary actions:

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

Creating an Instance of the Class

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

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

Calling Class Methods from Java

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

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

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

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

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

Next Steps

Understanding concepts needed to use MATLAB Builder for Java	"What Is a Project?" on page 2-2
Writing Java applications that can access Java methods that encapsulate M-code	"Import Classes " on page 3-3
Sample applications that access methods developed in MATLAB	"Plot Example" on page 5-2
Reference information about automatic data conversion rules	"Data Conversion Rules" on page 7-7

Concepts

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

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

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

What Is a Project?

In this section...

Overview

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

Classes and Methods

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

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

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

Note When you add files to a project, you do not have to add any M-files for functions that are called by the functions that you add. When Java Builder builds a component, it automatically includes any M functions called by the functions that you explicitly specify for the component. See "Spectral Analysis Example" on page 5-8 for a sample application that illustrates this feature.

Naming Conventions

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

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

How Does MATLAB Builder for Java Use JAR Files?

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

For information on how to produce a separate CTF archive and JAR file (the default behavior prior to R2007b), see “Using MCR Component Cache and MWComponentOptions” on page 3-47 and learn how to use the `MWCtfExtractLocation.EXTRACT_TO_COMPONENT_DIR` value with the `ExtractLocation` option of `MWComponentOptions`.

How Does MATLAB Builder for Java Handle Data?

In this section...

Java Builder API

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

Understanding the API Data Conversion Classes

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

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

Note This discussion provides conceptual information about the classes.

For usage information, see Chapter 4, "Using Classes and Methods".

For reference information, see `com.mathworks.toolbox.javabuilder`.

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

Overview of Classes and Methods in the Data Conversion Class Hierarchy

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

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

- Constructors and finalizers to instantiate and dispose of MATLAB arrays
- `get` and `set` methods to read and write the array data
- Methods to identify properties of the array
- Comparison methods to test the equality or order of the array
- Conversion methods to convert to other data types

Advantage of Using Data Conversion Classes

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

Automatic Conversion to MATLAB Types

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

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

In contrast, if your code uses a native Java primitive or array as an input parameter, Java Builder converts it to an instance of the appropriate `MWArray` class before it is passed to the method. Java Builder can convert any Java string, numeric type, or any multidimensional array of these types to an appropriate `MWArray` type, using its data conversion rules. See "Data Conversion Rules" on page 7-7 for a list of all the data types that are supported along with their equivalent types in MATLAB.

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

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

Understanding Function Signatures Generated by Java Builder

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

Note In addition to handling optional function arguments, the overloaded Java methods that wrap MATLAB functions handle data conversion. See "Automatic Conversion to MATLAB Types" on page 2-7 for more details.

Understanding MATLAB Function Signatures

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

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

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

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

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

Overloaded Methods in Java That Encapsulate M-Code

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

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

These overloaded methods are called the standard interface (encapsulating input arguments) and the `mlx` interface (encapsulating return values). See "Programming Interfaces Generated by Java Builder" on page 7-11 for details.

Returning Data from MATLAB to Java

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

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

What Happens in the Build Process?

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

To create a component, Java Builder does the following:

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

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

2. Compiles the Java code produced in step 1.
3. Generates `/distrib` and `/src` subdirectories.
4. Invokes the Jar utility to package the Java class files it has created into a Java archive file (*`mycomponent`*`.jar`).

What Happens in the Package Process?

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

- The Java Builder component
- The MCR Installer (if the **Install MCR** option was selected when the component was built)
- Documentation generated by Sun Microsystems Inc.'s Javadoc tool

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

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

The reason for this limitation is that although the `.jar` file itself might be platform independent, the `.jar` file is dependent on the `.ctf` file, which is not platform independent.

How Does Component Deployment Work?

There are two kinds of deployment:

- Installing components and setting up support for them on a development machine so that they can be accessed by a developer who seeks to use them in writing a Java application.
- Deploying support for the components when they are accessed at run time on an end-user machine.

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

Programming

To access a Java component built and packaged by MATLAB Builder for Java, you must first unpack and install components so you can use them on a particular machine.

Then you perform the following programming tasks:

Ensuring Multi-Platform Portability (p. 3-45)	Learn how to ensure platform independence if your CTF archive contains MEX files
Using MCR Component Cache and MWComponentOptions (p. 3-47)	Save network storage space by learning how to store components locally, or control how the CTF archive is managed and stored.
Learning About Java Classes and Methods by Exploring the Javadoc (p. 3-50)	How to search for information on Java classes and methods used with Java Builder by searching the Javadoc

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

For examples of these tasks, see Chapter 5, “Sample Java Applications”.

For information about deploying your application after you complete these tasks, see “How Does Component Deployment Work?” on page 2-12.

Import Classes

To use a component generated by MATLAB Builder for Java, you must do the following:

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

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

2. Import the component classes created by Java Builder, for example:

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

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

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

The reason for this limitation is that although the `.jar` file itself might be platform-independent, the `.jar` file is dependent on the `.ctf` file, which is intrinsically platform dependent. It is possible to make your `.ctf` file platform independent in certain circumstances; see "Ensuring Multi-Platform Portability" on page 3-45 for more details.

Creating an Instance of the Class

In this section...

What is an Instance?

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

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

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

Code Fragment: Instantiating a Java Class

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

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

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

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

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

```
      {
         super();
      }

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

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

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

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

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

The following statement instantiates the class `myclass`:

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

The following statement calls the class method `myprimes`:

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

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

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

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

myprimes Function

The code for `myprimes` is as follows:

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

%   Copyright 2001-2007 The MathWorks, Inc.

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

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

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

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

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

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

Passing Arguments to and from Java

In this section...

The Format

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

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

Manual Conversion of Data Types

To manually convert to one of the standard MATLAB data types, use the `MWArray` data conversion classes provided by Java Builder. For class reference information, see the `com.mathworks.toolbox.javabuilder` package. For extensive usage information, see Chapter 4, "Using Classes and Methods".

Code Fragment: Using MWNumericArray

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

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

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

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

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

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

Automatic Conversion to a MATLAB Type

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

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

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

For reference information about data conversion (tables showing each Java type along with its converted MATLAB type, and each MATLAB type with its converted Java type), see "Data Conversion Rules" on page 7-7.

Code Fragment: Automatic Data Conversion

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

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

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

Here is another example:

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

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

Code Fragment: Passing a Java Double Object

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

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

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

Code Fragment: Passing an MWArray

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

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

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

Code Fragment: Calling MWArray Methods

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

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

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

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

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

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

Changing the Default by Specifying the Type

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

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

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

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

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

Specifying Optional Arguments

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

```
function y = mysum(varargin)
```

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

%   Copyright 2001-2007 The MathWorks, Inc.

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

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

Code Fragment: Passing Variable Numbers of Inputs

Java Builder generates a Java interface to this function as follows:

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

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

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

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

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

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

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

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

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

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

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

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

   try
   {
      x = new Object[vals.length];
      for (int i = 0; i < vals.length; i++)
         x[i] = new MWNumericArray(vals[i], MWClassID.DOUBLE);

      cls = new myclass();
      y = cls.mysum(1, x);
      return ((MWNumericArray)y[0]).getDouble(1);
   }
   finally
   {
      MWArray.disposeArray(x);
      MWArray.disposeArray(y);
      if (cls != null)
         cls.dispose();
   }
}
```

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

Code Fragment: Passing a Variable Number of Outputs

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

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

%   Copyright 2001-2007 The MathWorks, Inc.

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

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

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

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

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

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

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

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

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

Handling Return Values

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

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

- Use reflection support in the Java language to query any object for its type.
- Use several methods provided by the `MWArray` class to query information about the underlying MATLAB array.
- Coercing to a specific type using the to*`Type`*Array methods.

Code Fragment: Using Java Reflection

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

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

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

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

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

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

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

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

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

      else if (a instanceof long[][])
      {
         long[][] x = (long[][])a;
         /* (do something with x...) */
      }

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

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

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

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

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

This example uses the `toArray` method (see "Methods to Copy, Convert, and Compare MWArrays" on page 4-59) to return a Java primitive array representing the underlying MATLAB array. The `toArray` method works just like `getData` in the previous examples, except that the returned array has the same dimensionality as the underlying MATLAB array.

Code Fragment: Using MWArray Query

The next example uses the `MWArray` `classID` method (see "Methods to Return Information About an MWArray" on page 4-50) to determine the type of the underlying MATLAB array. It also checks the dimensionality by calling `numberOfDimensions`. If any unexpected information is returned, an exception is thrown. It then checks the `MWClassID` and processes the array accordingly.

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

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

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

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

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

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

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

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

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

      else if (clsid == MWClassID.INT64 ||
              clsid == MWClassID.UINT64)
      {
         long[][] x = (long[][])((MWArray)y[0]).toArray();

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

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

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

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

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

Code Fragment: Using to*Type*Array Methods

The next example demonstrates how you can coerce or force data to a specified numeric type by invoking any of the to*Type*Array methods (see "Additional MWArray Methods" on page 4-4 in Chapter 4, "Using Classes and Methods" for more information about these methods) . These methods return an array of Java types matching the primitive type specified in the name of the called method. The data is coerced or forced to the primitive type specified in the method name. Note that when using these methods, data will be truncated when needed to allow conformance to the specified data type.

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

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

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

Passing Java Objects by Reference

In this section...
"MATLAB Array" on page 3-22
"Wrappering and Passing Java Objects to M-Functions with MWJavaObjectRef" on page 3-22

MATLAB Array

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

Wrappering and Passing Java Objects to M-Functions with MWJavaObjectRef

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

Code Fragment: Passing a Java Object Into a Java Builder Component

To pass an object into a Java Builder component, simply do the following:

1. Use `MWJavaObjectRef` to wrap your object.
2. Pass your object to an M-function.

For example:

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

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

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

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

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

displayObj.m.

```
function className = displayObj(h)

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


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

Code Fragment: Clone an Object Inside a Builder Component

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

1. Add to the original hash.
2. Clone the object.
3. Optionally, continue to add items to each copy.

   For example:

```
            origRef = new MWJavaObjectRef(hash);
            System.out.println("hash:");
            System.out.println(hash.toString());

            result = theComponent.addToHash(1, origRef);

            outputRef = (MWJavaObjectRef)result[0];

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

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

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

addToHash.m.

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

%   Copyright 2001-2007 The MathWorks, Inc.
%   $Revision: 1.1.4.50 $  $Date: 2007/07/27 18:43:13 $

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

% Add an item
```

```
h.put('From MATLAB',12);
% Clone the Hashtable and add items to both resulting objects
h2 = h.clone();
h.put('Orig',20);
h2.put('Clone',21);
```

Code Fragment: Passing a Date Into a Component and Getting a Date From a Component

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

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

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

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

   For example:

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

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

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

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

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

nextWeek.m.

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

%   Copyright 2001-2007 The MathWorks, Inc.
%   $Revision: 1.1.4.50 $  $Date: 2007/07/27 18:43:13 $

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

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

Returning Java Objects Using unwrapJavaObjectRefs

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

This method recursively connects a single `MWJavaObjectRef` or a multi-dimentional array of `MWJavaObjectRef` objects to a reference or array of references.

The following code snippets show two examples of calling `unwrapJavaObjectRefs`:

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

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

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

Produces the following output:

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

Code Snippet: Returning an Array of References with unwrapJavaObjectRefs.

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

Produces the following output:

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

An Optimization Example Using MWJavaObjectRef

For a full example of how to utilize `MWJavaObjectRef` to create a reference to a Java object and pass it to a component, see the“Optimization Example” on page 5-42.

Handling Errors

In this section...

Error Overview

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

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

Handling Checked Exceptions

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

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

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

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

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

- Catch and handle the `MWException`.
- Allow the calling program to catch it.

The following two sections provide examples of each.

Code Fragment: Handling an Exception in the Called Function

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

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

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

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

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

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

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

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

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

Code Fragment: Handling an Exception in the Calling Function

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

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

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

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

Handling Unchecked Exceptions

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

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

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

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

Code Fragment: Catching General Exceptions

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

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

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

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

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

Code Fragment: Catching Multiple Exception Types

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

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

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

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

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

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

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

Managing Native Resources

In this section...

What are Native Resources?

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

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

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

Using Garbage Collection Provided by the JVM

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

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

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

Using the dispose Method

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

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

Code Fragment: Using dispose

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

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

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

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

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

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

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

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

Code Fragment: Using dispose in a finally Clause.

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

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

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

Overriding the Object.Finalize Method

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

Handling Data Conversion Between Java and MATLAB

In this section...

Overview

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

For example, consider the following Java statement:

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

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

Calling MWArray Methods

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

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

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

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

Specifying the Type

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

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

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

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

Creating Buffered Images From a MATLAB Array

Use the `renderArrayData` method to:

- Create a buffered image from data in a given MATLAB array.
- Verify the array is of three dimensions (height, width, and color component).
- Verify color component order is red, green, and blue.

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

For a complete example of `renderArrayData`'s implementation, see "Buffered Image Creation Example" on page 5-37.

Setting Java Properties

In this section...

How to Set Java System Properties

Set Java system properties in one of two ways:

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

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

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

Ensuring a Consistent GUI Appearance

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

Code Fragment: Setting DisableSetLookAndFeel

The following are examples of how to set `mathworks.DisableSetLookAndFeel` using the techniques in "How to Set Java System Properties" on page 3-40:

- In the `Java` statement:

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

- In the Java code:

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

Blocking Execution of a Console Application that Creates Figures

In this section...

waitForFigures Method

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

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

- There are one or more figures open that were created by a Java component created by Java Builder.
- 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.

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

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

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

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

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

   ```
   drawplot.m

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

3. Use Java Builder to create a Java component with the following properties:

Package name	`examples`
Class name	`Plotter`

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

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

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

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

5 Compile the application with the `javac` command. For an example, see "Create a Deployable Java Builder Component" on page 1-7.

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

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

Ensuring Multi-Platform Portability

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

However, if your CTF archive 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 Java Builder application.

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

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

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

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

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

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

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

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

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

Using MCR Component Cache and MWComponentOptions

In this section...

MWComponentOptions

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

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

Select Options

Choose from the following `CtfSource` or `ExtractLocation` instantiation options to customize how Java Builder manages CTF content with `MWComponentOptions`.

- `CtfSource` — This option specifies where the CTF file may be found for an extracted component. It defines a binary data stream comprised of the bits of the CTF file. The following values are objects of some type extending `MWCtfSource`:
  - `MWCtfSource.NONE` — Indicates that no CTF file is to be extracted. This implies that the extracted CTF data is already accessible somewhere on your file system. This is a public, static, final instance of `MWCtfSource`.
  - `MWCtfFileSource` — Indicates that the CTF data resides within a particular file location that you specify. This class takes a `java.io.File` object in its constructor.
  - `MWCtfDirectorySource` — Indicate a directory to be scanned when instantiating the component: if a file with a `.ctf` suffix is found in the

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

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

- `ExtractLocation` — This option specifies where the extracted CTF content is to be located. Since the MCR requires all CTF content be located somewhere on your file system, use the desired `ExtractLocation` option, along with the component type information, to define a unique location. A value for this option is an instance of the class `MWCtfExtractLocation`. An instance of this class can be created by passing a `java.io.File` or `java.lang.String` into the constructor to specify the file system location to be used or one of these predefined, static final instances may be used:
  - `MWCtfExtractLocation.EXTRACT_TO_CACHE` — use to indicate that the CTF content is to be placed in the MCR component cache. This is the default setting for releases R2007a and forward (see "How Does MATLAB Builder for Java Use JAR Files?" on page 2-4).
  - `MWCtfExtractLocation.EXTRACT_TO_COMPONENT_DIR` — Use when you want to locate the JAR or `.class` files from which the component has been loaded. If the location is found (e.g.: it is on the file system), then the CTF data is extracted into the same directory. This option most closely matches the behavior of R2007a and previous releases.

Set Options

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

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

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

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

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

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

```
MWComponentOptions options = new MWComponentOptions();

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

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

Learning About Java Classes and Methods by Exploring the Javadoc

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

Javadoc contains, among other information:

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

Using Classes and Methods

The following topics explain how to use the data conversion classes in the `com.mathworks.toolbox.javabuilder.MWArray` package.

Guidelines for Working with MWArray Classes

Overview of the MWArray API

The MWArray Java API is a class hierarchy that represents the major MATLAB array types. The root class is `MWArray`, which has the following subclasses:

- `MWNumericArray`
- `MWLogicalArray`
- `MWCharArray`
- `MWCellArray`
- `MWStructArray`

These subclasses provide constructors and factory methods for creating new MATLAB arrays from standard Java types and objects. You can use these MATLAB arrays as arguments in method calls.

Note To improve performance, `MWArrays` are designed so that they cannot be resized or reshaped once they are created.

Understanding the MWArray Base Class

`MWArray` stores a reference to a native MATLAB array and provides a set of methods for accessing the array's properties and data. `MWArray` also provides methods for converting the MATLAB array to standard Java types from the outputs of a Java class method call.

Accessing Elements of the Arrays

You cannot access the underlying MATLAB array's data buffers directly. Instead use `set` and `get` methods to retrieve or modify an element of the array. The `set` and `get` methods support simple indexing through a single subscript (value at offset) or you can supply an array of `int` representing the indices of the requested value. In the case of structure arrays, indexing by field name is also supported.

Method Overrides Implemented by MWArray

To ensure integration with Java programs, `MWArray` provides overrides for `java.lang.Object` methods and implements the required Java interfaces as needed. The following table provides more information about the overrides.

Overrides

Method in MWArray Base Class	Override Description
`equals`	Overrides `Object.equals` to provide a logical equality test for two `MWArrays`. Internally, this method does a byte-wise comparison of the native buffer. Therefore, two `MWArray` instances are logically equal when they are of the same MATLAB type and have identical size, shape, and content.
`hashCode`	Overrides `Object.hashCode` to allow `MWArray` to function properly with hash-based collections.
`toString`	Overrides `Object.toString` so that `MWArray` objects will print properly. This method formats a new `java.lang.String` from the underlying MATLAB array so that calls to `System.out.println` with an `MWArray` as an argument will produce the same output as displaying the array in MATLAB.
`finalize`	Overrides `Object.finalize` so that the underlying MATLAB array is destroyed when the garbage collector reclaims the containing `MWArray` object. This method has protected access and is not user callable.

Java Interfaces Implemented by MWArray

`MWArray` implements the standard Java interfaces shown in the following table.

Java Interfaces Implemented by MWArray

Interface	Method in MWArray Base Class	Description of Method
`Cloneable`	`clone` (public method)	Produces a new `MWArray` object that contains a deep copy of the underlying MATLAB array.
`Comparable`	`compareTo` (public method)	Allows comparisons of `MWArrays` for order. Internally, this method does a byte-wise comparison of the native buffer. Therefore, `MWArray` has a natural ordering that is based on a combination of the array's MATLAB type, size, and shape.

`Serializable`	`writeObject`
	`readObject`
	(private methods)
		Provides serialization support as required by the `Serializable` interface.

Additional MWArray Methods

`MWArray` also implements several base class methods that are common to all `MWArray` subclasses. These methods are shown in the following table.

Method	Usage
`MWArray()`	Constructs an empty array.
`classID()`	Returns the MATLAB type of the array.

Method	Usage
`columnIndex()`	Returns the column index (second dimension) of each element in the array. Call this method to get an array of column indices for the nonzero elements of a sparse array.
`dispose()`	Frees the native resources associated with the underlying MATLAB array.
`disposeArray(Object)`	Calls `dispose` on all `MWArray` instances contained in the input.
`get(int)`	Returns the elements at the specified one-based offset.
`get(int[])`	Returns the elements at the specified one-based index array.
`getData()`	Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array as an array of Java types. The elements in the returned array are arranged in column-wise order. The different kinds of arrays are returned as follows: • If the underlying MATLAB array is complex, the real part is returned. • If the underlying array is sparse, an array containing the nonzero elements is returned. • If the underlying array is a cell or struct array, `toArray` is recursively called on each element.
`getDimensions()`	Returns an array of dimensions for the array.
`isEmpty()`	Tests if the array is empty.
`isSparse()`	Tests if the array is sparse.

Method	Usage
`maximumNonZeros()`	Returns the current allocated capacity of nonzero elements for a sparse array.
`numberOfDimensions()`	Returns the number of dimensions in the array.
`numberOfElements()`	Returns the number of elements in the array.
`numberOfNonZeros()`	Returns the current number of nonzero elements for a sparse array.
`rowIndex()`	Returns the row index (first dimension) of each element in the array. Call this method to get an array of row indices for the nonzero elements of a sparse array.
`set(int, Object)`	Replaces the element at the one-based index with the supplied value.
`set(int[], Object)`	Replaces the element at the one-based index array with the supplied value.
`setdata(Object)`	Sets the elements of the underlying MATLAB array to the given values. This method should only be called on primitive `MWArray` types (not, for example, on `MWCellArray`, `MWStructArray`, or `MWJavaObjectRef`).

Method	Usage
`sharedCopy()`	Creates a new `MWArray` instance that represents a shared copy of the underlying MATLAB array. A shared copy points to the same underlying MATLAB array as the original. Changing the data in a shared copy also changes the original array.
`toArray()`	Returns an array containing a copy of the data in the underlying MATLAB array as an array of Java types. The returned array has the same dimensionality as the underlying MATLAB array. The different kinds of arrays are returned as follows: • If the underlying MATLAB array is complex, the real part is returned. • If the underlying array is sparse, a full representation of the array is returned. • If the underlying array is a cell or struct array, `toArray` is recursively called on each element. • If the underlying array is empty (including uninitialized elements of a struct or cell array), `toArray` returns an array of `double[0][]`. See also "Methods to Convert Array Data to a Specific Type" on page 4-103.

Note MCOS objects, Java objects, and function handles passed from M to Java via a Java Builder component and accessed using an `MWArray` are unsupported and will cause a run-time exception.

Constructing Numeric Arrays

The `MWNumericArray` class provides a Java interface to a numeric MATLAB array. An instance of this class can store a reference to a MATLAB array of type `double`, `single`, `int8`, `uint8`, `int16`, `int32`, `uint32`, `int64`, and `uint64`. `MWNumericArrays` can be real or complex, dense or sparse (sparse is supported for `double` type only).

Overview of Constructors and Data Types

The following table lists `MWNumericArray` class constructors.

Constructor	Usage
`MWNumericArray()`	Empty `double` array
`MWNumericArray (MWClassID)`	Empty array of type specified by `MWClassID`
`MWNumericArray(`*`type`*`, MWClassID)`	Real array of type specified by `MWClassID`
`MWNumericArray(`*`type`*`)`	Real array with type determined from default conversion rules
`MWNumericArray(`*`type`*`, `*`type`*`, MWClassID)`	Complex array of type specified by `MWClassID`
`MWNumericArray(`*`type`*`, type)`	Complex array with type determined from default conversion rules

Supported Data Types. In the previous table, *`type`* represents supported Java types. `MWNumericArray` supports the following Java primitive types:

- `double`
- `float`
- `byte`
- `short`
- `int`
- `long`
- `boolean`

The following object types are also supported:

- Subclasses of `java.lang.Number`
- Subclasses of `java.lang.String`
- Subclasses of `java.lang.Boolean`

In addition to supporting scalar values of the types listed, general N-dimensional arrays of each type are also supported.

Constructing Different Types of Numeric Arrays

Here are some examples showing how to construct different types of numeric arrays with the various forms of the `MWNumericArray` constructor.

Constructing Complex Arrays

The following four statements all construct a complex scalar `int32` array with a value of `1+2i`:

```
MWNumericArray a1 = new MWNumericArray(1, 2);
MWNumericArray a2 = new MWNumericArray(1.0, 2.0,
   MWClassID.INT32);
MWNumericArray a3 = new MWNumericArray(new Double(1.0),
   New Integer(2),  MWClassID.INT32);
MWNumericArray a4 = new MWNumericArray("1.0", "2.0",
   MWClassID.INT32);
```

Constructing Matrices

The next group of statements constructs a 2-by-2 double matrix with the following values:

```
 [1  2
  3  4]

double[][] x1 = {{1.0, 2.0}, {3.0, 4.0}};
int[][] x2 = {{1, 2}, {3, 4}};
Double[][] x3 = {{new Double(1.0), new Double(2.0)},
   {new Double(3.0), new Double(4.0)}};
String[][] x4 = {{"1.0", "2.0'}, {"3.0", "4.0"}};

MWNumericArray a1 = new MWNumericArray(x1, MWClassID.DOUBLE);
MWNumericArray a2 = new MWNumericArray(x2, MWClassID.DOUBLE);
MWNumericArray a3 = new MWNumericArray(x3, MWClassID.DOUBLE);
MWNumericArray a4 = new MWNumericArray(x4, MWClassID.DOUBLE);
```

Constructing N-Dimensional Arrays

The `MWNumericArray` constructors also support multidimensional arrays of all supported types. For example, you can construct a 2-by-3-by-2 `double` array with the following two statements:

```
Double[][][] x1 = {
   {{ 1.0,  2.0,  3.0},
    { 4.0,  5.0,  6.0}},
   {{ 7.0,  8.0,  9.0},
    {10.0, 11.0, 12.0}}
   };

MWNumericArray a1 = new MWNumericArray(x1);
```

Constructing Jagged Arrays

The previous examples constructed rectangular Java arrays and used these arrays to initialize MATLAB arrays. Multidimensional arrays in Java are implemented as arrays of arrays, which means that it is possible to construct a Java array in which each row can have a different number of columns. Such arrays are commonly referred to as *jagged* arrays.

`MWNumericArray` constructors support jagged arrays by constructing a rectangular array and padding with zeros any missing elements. The resulting MATLAB array will have a column count equal to the largest column count in any row of the input array. For example, the following two statements construct a 5-by-5 `double` matrix from a 5-by-5 Java `double` array in which the number of columns in the *i*th row equals `i`:

```
double[][] pascalsTriangle = {
             {1.0},
           {1.0, 1.0},
         {1.0, 2.0, 1.0},
       {1.0, 3.0, 3.0, 1.0},
     {1.0, 4.0, 6.0, 4.0, 1.0}
   };

MWNumericArray a1 = new MWNumericArray(pascalsTriangle);
```

The resulting MATLAB array has the following structure:

```
[1 0 0 0 0
 1 1 0 0 0
 1 2 1 0 0
 1 3 3 1 0
 1 4 6 4 1]
```

Passing Arguments to Constructors as `MWClassID`. In some cases, the constructor converts the input to the specified type passed as an `MWClassID` value. When this value is omitted, the inputs are converted according to default conversion rules.

For example, each of the following statements creates a real scalar `double` array with a value of 1.0:

```
MWNumericArray a1 = new MWNumericArray(1.0);
MWNumericArray a2 = new MWNumericArray(1, MWClassID.DOUBLE);
MWNumericArray a3 = new MWNumericArray(new Double(1.0),
   MWClassID.DOUBLE);
MWNumericArray a4 = new MWNumericArray("1.0", MWClassID.DOUBLE);
```

In general, it is most efficient to supply an argument that causes the desired array to be created using the default conversion rules.

Some types require coercion to produce the correct MATLAB array. If you supply an unsupported type to an `MWNumericArray` constructor, an exception is thrown and the array is not created.

The following example constructs a real 1-by-3 `double` array with the values `[1 2 3]`:

```
double[] x1 = {1.0, 2.0, 3.0};
int[] x2 = {1, 2, 3};
Double[] x3 = {new Double(1.0), new Double(2.0),
   new Double(3.0)};
String[] x4 = {"1.0", "2.0", "3.0"};

MWNumericArray a1 = new MWNumericArray(x1, MWClassID.DOUBLE);
MWNumericArray a2 = new MWNumericArray(x2, MWClassID.DOUBLE);
MWNumericArray a3 = new MWNumericArray(x3, MWClassID.DOUBLE);
MWNumericArray a4 = new MWNumericArray(x4, MWClassID.DOUBLE);
```

Using Static Factory Methods to Construct MWNumericArrays

An alternative method for constructing numeric arrays is to use the static factory methods of the `MWNumericArray` class. The following table lists such methods.

Factory Method	Usage
`newInstance(int[], MWClassID, MWComplexity)`	Numeric array of specified type and complexity. Values are initialized to 0.

Factory Method	Usage
`newInstance(int[], Object, MWClassID)`	Real numeric array of specified type. Values are initialized with supplied arrays.
`newInstance(int[], Object, Object, MWClassID)`	Complex numeric array of specified type. Values are initialized with supplied arrays.
`newSparse(int[], int[], Object, int, int, int, MWClassID)`	Real sparse numeric matrix of specified type, dimensions, and maximum nonzeros. Values are initialized with supplied row, column, and data arrays.
`newSparse(int[], int[], Object, int, int, MWClassID)`	Real sparse numeric matrix of specified type and dimensions. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros are computed from input data.
`newSparse(int[], int[], Object, MWClassID)`	Real sparse numeric matrix of specified type. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros and dimensions are computed from input data.
`newSparse(int[], int[], Object, Object, int, int, int, MWClassID)`	Complex sparse numeric matrix of specified type, dimensions, and maximum nonzeros. Values are initialized with supplied row, column, and data arrays.
`newSparse(int[], int[], Object, Object, int, int, MWClassID)`	Complex sparse numeric matrix of specified type and dimensions. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros are computed from input data.

Factory Method	Usage
`newSparse(int[], int[], Object, Object, MWClassID)`	Complex sparse numeric matrix of specified type. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros and dimensions are computed from input data.
`newSparse(int, int, int, MWClassID, MWComplexity)`	Sparse numeric matrix with specified type, complexity, dimensions, and maximum nonzeros. Values are initialized to 0.
`newSparse(Object, MWClassID)`	Real sparse numeric matrix of specified type. Values are initialized from the supplied full matrix.
`newSparse(Object, Object, MWClassID)`	Complex sparse numeric matrix of specified type. Values are initialized from the supplied full matrix.

Data Arrangement in the Array. Each of the static factory methods for `MWNumericArray` returns a new `MWNumericArray` instance constructed from the input information. The methods can be used to construct and initialize an array with supplied data, or to construct an array of a specified size and initialize all values to zero. The main difference is that (with exception of the last two `newSparse` methods) data is supplied to the factory methods in one-dimensional arrays with the data arranged in column-wise order.

The following example rewrites the previous one-dimensional array constructor example:

```
double[] x1 = {1.0, 2.0, 3.0};
int[] x2 = {1, 2, 3};
Double[] x3 = {new Double(1.0),
               new Double(2.0),
               new Double(3.0)};
String[] x4 = {"1.0", "2.0", "3.0"};

int[] dims = {1, 3};
MWNumericArray a1 =
   MWNumericArray.newInstance(dims, x1, MWClassID.DOUBLE);
MWNumericArray a2 =
   MWNumericArray.newInstance(dims, x2, MWClassID.DOUBLE);
MWNumericArray a3 =
   MWNumericArray.newInstance(dims, x3, MWClassID.DOUBLE);
MWNumericArray a4 =
   MWNumericArray.newInstance(dims, x4, MWClassID.DOUBLE);
```

Similarly, the 2-by-2 matrix example can be rewritten as follows:

```
   double[] x1 = {1.0, 3.0, 2.0, 4.0};
   int[] x2 = {1, 3, 2, 4};
   Double[] x3 = {new Double(1.0),
                  new Double(3.0),
                  new Double(2.0),
                  new Double(4.0)};
   String[] x4 = {"1.0", "3.0"", "2.0", "4.0"};

   int[] dims = {2, 2};
   MWNumericArray a1 =
      MWNumericArray.newInstance(dims, x1, MWClassID.DOUBLE);
   MWNumericArray a2 =
      MWNumericArray.newInstance(dims, x2, MWClassID.DOUBLE);
   MWNumericArray a3 =
      MWNumericArray.newInstance(dims, x3, MWClassID.DOUBLE);
   MWNumericArray a4 =
      MWNumericArray.newInstance(dims, x4, MWClassID.DOUBLE);
```

Note the order of the data in the input buffers. The matrix elements are entered in column-wise order, which is the internal storage order used by MATLAB.

Constructing Sparse Arrays

An efficient way to construct sparse matrices is to use the `newSparse` constructor methods. The examples shown here create a 4-by-4 sparse matrix with the following values:

```
   x = [ 2 -1  0  0
        -1  2 -1  0
         0 -1  2 -1
```

```
      0  0 -1  2 ]
```

Calling newSparse. The call to newSparse passes three arrays: an array of matrix data (x), an array containing the row indices of x (rowindex), and an array of column indices of x (colindex). The number of rows (4) and columns (4) are also passed, as well as the type (MWClassID.DOUBLE):

```
double[] x = { 2.0, -1.0, -1.0,  2.0, -1.0,
              -1.0,  2.0, -1.0, -1.0,  2.0 };
int[] rowindex = {1, 2, 1, 2, 3, 2, 3, 4, 3, 4};
int[] colindex = {1, 1, 2, 2, 2, 3, 3, 3, 4, 4};

MWNumericArray a =
   MWNumericArray.newSparse(rowindex, colindex, x, 4, 4,
      MWClassID.DOUBLE);
```

Constructing the Array Without Setting Rows and Columns. You could have passed just the row and column arrays and let the newSparse method determine the number of rows and columns from the maximum values of rowindex and colindex as follows:

```
MWNumericArray  a = MWNumericArray.newSparse(rowindex, colindex,
   x, MWClassID.DOUBLE);
```

Constructing the Array from a Full Matrix. You can also construct a sparse array from a full matrix using newSparse. The next example rewrites the previous example using a full matrix:

```
double[][] x = {{ 2.0, -1.0,  0.0,  0.0},
                {-1.0,  2.0, -1.0,  0.0},
                { 0.0  -1.0,  2.0, -1.0},
                { 0.0,  0.0, -1.0,  2.0 }};

MWNumericArray a = MWNumericArray.newSparse(x,
MWClassID.DOUBLE);
```

Note Numeric sparse matrices are supported only for type `double`. Attempting to construct a sparse numeric matrix with any other type results in an exception being thrown.

Accessing MWNumericArray Elements

The `MWNumericArray` class provides methods for accessing and modifying array data in the form of `get` and `set` methods. The following table lists the `get` and `set` methods.

Method	Usage
`get`*`type`*`(int)`	Returns the real part of the element at the one-based index. Return value is of the type specified (e.g., `getDouble` returns a `double`).
`get`*`type`*`(int[])`	Returns the real part of the element at the one-based index array. Return value is of the type specified (e.g., `getDouble` returns a `double`).
`getImag`*`type`*`(int)`	Returns the imaginary part of the element at the one-based index. Return value is of the type specified (e.g., `getImagDouble` returns a `double`).
`getImag`*`type`*`(int[])`	Returns the imaginary part of the element at the one-based index array. Return value is of the type specified (e.g., `getDouble` returns a `double`).
`set(int, `*`type`*`)`	Replaces the real part of the element at the one-based index with the supplied value.
`set(int[], `*`type`*`)`	Replaces the real part of the element at the one-based index array with the supplied value.

Method	Usage
`setImag(int, type)`	Replaces the imaginary part of the element at the one-based index with the supplied value.
`setImag(int[], type)`	Replaces the imaginary part of the element at the one-based index array with the supplied value.

In these method calls, `type` represents one of the following supported Java types of `MWNumericArray`:

- `double`
- `float`
- `byte`
- `short`
- `int`
- `long`
- `Boolean`
- Subclass of `java.lang.Number`
- Subclass of `java.lang.String`
- Subclass of `java.lang.Boolean`

The `get` and `set` methods access a single element at a specified index. An index is passed to these accessor methods in the form of a single offset or as an array of indices.

Note All indexing is one-based, which is the MATLAB convention, as opposed to zero-based, which is the Java convention.

Examples of Using set. The following examples construct the 2-by-2 matrix of the previous example using the set method. The first example uses a single index:

```
int[] dims = {2, 2};
MWNumericArray a =
   MWNumericArray.newInstance(dims, MWClassID.DOUBLE,
      MWComplexity.REAL);
int index = 0;
double[] values = {1.0, 3.0, 2.0, 4.0};

for (int index = 1; index <= 4; index++)
   a.set(index, values[index-1]);
```

Here is the same example, but this time using an index array:

```
int[] dims = {2, 2};
MWNumericArray a =
   MWNumericArray.newInstance(dims, MWClassID.DOUBLE,
      MWComplexity.REAL);
int[] index = new int[2];
int k = 0;

for (index[0] = 1; index[0] <= 2; index[0]++)
{
   for (index[1] = 1; index[1] <= 2; index[1]++)
      a.set(index, ++k);
}
```

The sparse array example can likewise be rewritten using set as follows:

```
MWNumericArray a =
   MWNumericArray.newSparse(4, 4, 10, MWClassID.DOUBLE,
      MWComplexity.REAL);
int[] index = {1, 1};

for (index[0] = 1; index[0] <= 4; index[0]++)
{
   for (index[1] = 1; index[1] <= 4; 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);
      }
   }
```

The example allocates the 4-by-4 sparse matrix with a capacity of 10 nonzero elements. Initially, the array has no nonzero elements. The `for` loops set the array's values using an index array.

Sparse arrays allocate storage only for the nonzero elements that are assigned. This example preallocates the array with a capacity of 10 elements because it is known in advance that this many nonzeros are needed. If you set additional zero elements to nonzero values, the allocated capacity is automatically increased to accommodate the new values.

Examples of Using get. The `get` methods work like the `set` methods. The `get` methods support indexing through one-based offset or index array. The next example displays the elements of an N-dimensional array where all indices are equal:

```
public void printDiagonals(MWNumericArray a)
{
   int[] dims = a.getDimensions();
   int n = dims[0];

   for (int i = 1; i < dims.length; i++)
   {
      if (dims[i] < n)
         n = dims[i];
   }

   for (int i = 1; i <= n; i++)
   {
      for (int j = 0; j < dims.length; j++)
         dims[j] = n;
      System.out.print("[");

      for (int j = 0; j < dims.length; j++)
         System.out.print(i + (j!=dims.length-1?",":""));
      System.out.print("] = " + a.getDouble(dims));

      if (a.complexity() == MWComplexity.COMPLEX)
         System.out.print(" + "+a.getImagDouble(dims)+"i");
      System.out.print("\n");
   }
}
```

The next example sums the real parts of all the elements in a numeric array and returns the result as a `double` value:

```
   public double sumElements(MWNumericArray a)
   {
      double sum = 0.0;
      int n = a.numberOfElements();

      for (int i = 1; i <= n; i++)
         sum = sum + a.getDouble(i);

      return sum;
   }
```

This example multiplies a Java `double[][]` with an `MWNumericArray` and returns the result as a Java `double[][]`:

```
   public double[][] matrixMult(double[][] a, MWNumericArray b)
   {
      int[] dims = b.getDimensions();
      double[][] result = new double[a.length][dims[1]];
      int[] index = new int[2];

      for (int i = 0; i < result.length; i++)
      {
         double[] row = a[i];
         if (row.length != dims[0])
            throw new IllegalArgumentException("Incompatible dims");

         for (index[1] = 1; index[1] <= result[0].length; index[1]++)
         {
            double sum = 0.0;
            for (index[0] = 1; index[0] <= dims[0]; index[0]++)
               sum += row[index[0]-1]*b.getDouble(index);
            result[i][index[0]] = sum;
         }
      }
      return result;
   }
```

Working with Logical Arrays

The `MWLogicalArray` class provides a Java interface to a MATLAB logical array. `MWLogicalArrays` can be dense or sparse.

Constructing an MWLogicalArray

The `MWLogicalArray` class provides a set of constructors and factory methods for creating logical arrays. The following table lists the supplied constructors.

Constructor	Usage
`MWLogicalArray()`	Empty `logical` array
`MWLogicalArray(`*`type`*`)`	`Logical` array with values initialized with supplied data

Here, *`type`* represents supported Java types. `MWLogicalArray` supports the following Java primitive types: `double`, `float`, `byte`, `short`, `int`, `long`, and `boolean`. The following object types are also supported: subclasses of `java.lang.Number`, `java.lang.String`, and `java.lang.Boolean`. In addition to supporting scalar values of the types listed, general N-dimensional arrays of each type are also supported.

When numeric types are used, the values in the logical array are set to `true` if the input value is nonzero, and `false` otherwise. The following examples create a scalar logical array with its value initialized to `true`:

```
MWLogicalArray a1 = new MWLogicalArray(true);
MWLogicalArray a2 = new MWLogicalArray(1);
MWLogicalArray a3 = new MWLogicalArray("true");
MWLogicalArray a4 = new MWLogicalArray(new Boolean(true));
```

These examples construct a scalar logical array initialized to `false`:

```
MWLogicalArray a1 = new MWLogicalArray(false);
MWLogicalArray a2 = new MWLogicalArray(0);
MWLogicalArray a3 = new MWLogicalArray("false");
MWLogicalArray a4 = new MWLogicalArray(new Boolean(false));
```

As with `MWNumericArray`, `MWLogicalArrays` can be constructed with multidimensional Java arrays. Here are some examples:

```
boolean[][] x1 = {{true, false}, {false, true}};
int[][] x2 = {{1, 0}, {0, 1}};

Boolean[][] x3 = {{new Boolean(true), new Boolean(false)},
                  {new Boolean(false), new Boolean(true)}};
String[][] x4 = {{"true", "false"},
                 {"false", "true"}};
boolean[][][] x5 = {
                    {{ true,  false, true},
                     { false, true,  false}},
                    {{ true,  false, true},
                     { false, true,  false}}
                   };

MWLogicalArray a1 = new MWLogicalArray(x1);
MWLogicalArray a2 = new MWLogicalArray(x2);
MWLogicalArray a3 = new MWLogicalArray(x3);
MWLogicalArray a4 = new MWLogicalArray(x4);
MWLogicalArray a5 = new MWLogicalArray(x5);
```

Using Static Factory Methods to Create MWLogicalArrays

The following table lists the static factory methods of `MWLogicalArray`.

Factory Method	Usage
`newInstance(int[])`	New `logical` array. Values are initialized to `false`.

Factory Method	Usage
`newInstance(int[], Object)`	New `logical` array. Values are initialized with supported data.
`newSparse(int[], int[], Object, int, int, int)`	Sparse `logical` matrix of specified dimensions and maximum nonzeros. Values are initialized with supplied row, column, and data arrays.
`newSparse(int[], int[], Object, int, int)`	Sparse `logical` matrix of specified dimensions. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros are computed from input data.
`newSparse(int[], int[], Object)`	Sparse `logical` matrix. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros and dimensions are computed from input data.
`newSparse(Object)`	Sparse `logical` matrix. Values are initialized from supplied full matrix.

These methods all return a new `MWLogicalArray` instance constructed from the input information. You can use these methods to construct and initialize an array with supplied data, or to construct an array of a specified size and initialize all values to `false`. The main difference is that, exception for the last `newSparse` method, data is supplied to the factory methods in one-dimensional arrays with the data arranged in column-wise order.

The following examples rewrite the two-dimensional array constructor examples using `newInstance`:

```
boolean[] x1 = {true, false, false, true};
int[] x2 = {1, 0, 0, 1};
Boolean[] x3 = {new Boolean(true), new Boolean(false),
                      new Boolean(false), new Boolean(true)};
String[] x4 = {"true", "false', "false", "true"};

int[] dims = {2, 2};
```

```
MWLogicalArray a1 = MWLogicalArray.newInstance(dims, x1);
MWLogicalArray a2 = MWLogicalArray.newInstance(dims, x2);
MWLogicalArray a3 = MWLogicalArray.newInstance(dims, x3);
MWLogicalArray a4 = MWLogicalArray.newInstance(dims, x4);
```

Accessing MWLogicalArray Elements

The `MWLogicalArray` class provides methods for accessing and modifying array data in the form of `get` and `set` methods. The following table lists the `get` and `set` methods.

Method	Usage
`get(int)`	Returns the element at the one-based index as type `java.lang.Boolean` (inherited from `MWArray`).
`get(int[])`	Returns the element at the one-based index array as type `java.lang.Boolean` (inherited from `MWArray`).
`getBoolean(int)`	Returns the element at the one-based index as type `boolean`.
`getBoolean(int[])`	Returns the element at the one-based index array as type `boolean`.
`set(int, Object)`	Replaces the element at the one-based index with the supplied value (inherited from `MWArray`).
`set(int[], Object)`	Replaces the element at the one-based index array with the supplied value (inherited from `MWArray`).
`set(int, boolean)`	Replaces the element at the one-based index with the supplied `boolean` value.
`set(int[], boolean)`	Replaces element at the one-based index array with the supplied `boolean` value.

The `get` methods return a `java.lang.Boolean` representing the value at the specified index. The `getBoolean` methods do the same thing, except they return a primitive `boolean` value. The `set` methods replace the value at the specified index with the supplied value. These methods collectively support the same types as the `MWLogicalArray` constructors: `boolean`, double, float, byte, short, int, long, java.lang.Boolean, subclasses of java.lang.Number, and java.lang.String.

Examples of Using set and get Methods. This example constructs a random sparse logical matrix with a specified fraction of nonzero elements:

```
MWLogicalArray getRandomSparse(int m, int n, double fillFactor)
{
   if (m < 0 || n < 0)
      throw new IllegalArgumentException(
         "Dimensions must be positive");

   if (fillFactor < 0.0 || fillFactor > 1.0)
      throw new IllegalArgumentException(
         "Fill factor must be between 0.0 and 1.0");

   int nsize = (int)(m*n*fillFactor);
   MWLogicalArray a = newSparse(m, n, nsize);
   if (nsize == 0)
      return a;

   while (a.numberOfNonZeros() < nsize)
   {
      int k = (int)(m*n*java.lang.Math.random());
      a.set((k != 0 ? k : 1), true);
   }
   return a;
}
```

This example toggles all elements of a logical array from true/false to false/true:

```
void toggleArray(MWLogicalArray a)
{
   for (int k = 1; k <= a.numberOfElements(); k++)
      a.set(k, !getBoolean(k));
}
```

Working with Character Arrays

The MWCharArray class provides a Java interface to a MATLAB char array.

Constructing an MWCharArray

The `MWCharArray` class provides a set of constructors and factory methods for creating logical arrays. The following table lists the supplied constructors.

Constructor	Usage
`MWCharArray()`	Empty `char` array
`MWCharArray(`*`type`*`)`	`char` array with values initialized with supplied data

Here, *`type`* represents supported Java types. `MWCharArray` supports the following Java types: `char`, `java.lang.Character`, and `java.lang.String`. In addition to supporting scalar values of the types listed, general N-dimensional arrays of each type are also supported. The following examples create scalar `char` arrays:

```
MWCharArray a1 = new MWCharArray('a');
MWCharArray a2 = new MWCharArray(new Character('a'));
```

Constructing Strings. You can use the `MWCharArray` class to create character strings, as shown in these examples:

```
char[] x1 = {'A', ' ', 'S', 't', 'r', 'i', 'n', 'g'};
String x2 = "A String";
Character[] x3 = {
   new Character('A'),
   new Character(' '),
   new Character('S'),
   new Character('t'),
   new Character('r'),
   new Character('i'),
   new Character('n'),
   new Character('g')};

MWCharArray a1 = new MWCharArray(x1);
MWCharArray a2 = new MWCharArray(x2);
MWCharArray a3 = new MWCharArray(x3);
```

Constructing an N-Dimensional Character Array. You can create a multidimensional char array using a multidimensional array of either char or java.lang.Character, or by using an array of java.lang.String, as shown in these examples:

```
char[][] x1 = {{'A', ' ', 'S', 't', 'r', 'i', 'n', 'g'}
               {'A', 'n', 'o', 't', 'h', 'e', 'r', ' ',
                'S', 't', 'r', 'i', 'n', 'g'}};
String[] x2 = {"A String",
               "Another String"};
Character[][] x3 = {
   {new Character('A'),
    new Character(' '),
    new Character('S'),
    new Character('t'),
    new Character('r'),
    new Character('i'),
    new Character('n'),
    new Character('g')},

   {new Character('A'),
    new Character('n'),
    new Character('o'),
    new Character('t'),
    new Character('h'),
    new Character('e'),
    new Character('r'),
    new Character(' '),
    new Character('S'),
    new Character('t'),
    new Character('r'),
    new Character('i'),
    new Character('n'),
    new Character('g')}
   };

MWCharArray a1 = new MWCharArray(x1);
MWCharArray a2 = new MWCharArray(x2);
MWCharArray a3 = new MWCharArray(x3);
```

The a1, a2, and a3 arrays constructed all contain a 2-by-14 MATLAB char array. The column count of the array is equal to the largest string length in the input array. Rows with fewer characters than the maximum are Null padded. Arrays with larger numbers of dimensions are handled similarly. This behavior parallels the way that MWNumericArray and MWLogicalArray handle jagged arrays.

Using Static Factory Methods for Constructing MWCharArrays

The following table lists the factory methods of MWCharArray.

Factory Method	Usage
newInstance(int[])	New char array. Values are initialized to Null.
newInstance(int[] Object)	New char array. Values are initialized with supported data.

These methods all return a new MWCharArray instance constructed from the input information. You can use these methods to construct and initialize an array with supplied data, or to construct an array of a specified size and initialize all values to zero. The main difference is that data is supplied to the factory methods in one-dimensional arrays with the data arranged in column-wise order. The input data array must be either a one-dimensional array of char, a one-dimensional array of java.lang.Character, or a single java.lang.String.

Rewriting Strings Using the newInstance Method. The following examples rewrite the character string examples using newInstance:

```
char[] x1 = {'A', ' ', 'S', 't', 'r', 'i', 'n', 'g'};
String x2 = "A String";
Character[] x3 =
{
   new Character('A'),
   new Character(' '),
   new Character('S'),
   new Character('t'),
   new Character('r'),
   new Character('i'),
   new Character('n'),
    new Character('g')
};

int[] dims = {1, 8};
MWCharArray a1 = MWCharArray.newInstance(dims, x1);
MWCharArray a2 = MWCharArray.newInstance(dims, x2);
MWCharArray a3 = MWCharArray.newInstance(dims, x3);
```

Constructing a Two-Dimensional Character Array. This example constructs the two-dimensional char array of the previous example:

```
char[] x1 = ('A', 'A', ' ', 'n', 'S', 'o', 't', 't', 'r', 'h',
                 'i', 'e', 'n', 'r', 'g', ' ', '\0', 'S', '\0', 't',
                 '\0', 'r', '\0', 'i', '\0', 'n', '\0', 'g'};

int[] dims = {2, 14};
     MWCharArray a1 = MWCharArray.newInstance(dims, x1);
```

Note that the array of characters supplied to initialize the array is arranged in column-wise order, and the end of the shorter string is padded with Null characters ('\0'). Higher dimensional character arrays can be constructed using the same procedure.

Accessing MWCharArray Elements

The MWCharArray class provides methods for accessing and modifying array data in the form of get and set methods. The following table lists the get and set methods.

Method	Usage
get(int)	Returns the element at the one-based index as type java.lang.Character (inherited from MWArray).
get(int[])	Returns the element at the one-based index array as type java.lang.Character (inherited from MWArray).
getChar(int)	Returns the element at the one-based index as type char.
getChar(int[])	Returns the element at the one-based index array as type char.

Method	Usage
`set(int, Object)`	Replaces the element at the one-based index with the supplied value (inherited from `MWArray`).
`set(int[], Object)`	Replaces the element at the one-based index array with the supplied value (inherited from `MWArray`).
`set(int, char)`	Replaces the element at the one-based index with the supplied `char` value.
`set(int[], char)`	Replaces element at the one-based index array with the supplied `char` value.

The `get` methods return a `java.lang.Character` representing the character at the specified index. The `getChar` methods do the same thing, except they return a primitive `char` value. The `set` methods replace the character at the specified index with the supplied value. These methods collectively support types `char`, `java.lang.Character`, and `java.lang.String` (use a `String` of length `1` to pass a character to replace).

Replacing Character Occurrences Using MWCharArray Methods. This example replaces every occurrence of a given character in an `MWCharArray` with a specified new character:

```
void replaceWithChar(MWCharArray a, char ch, char newch)
{
   if (a == null)
      return;

   for (int k = 1; k <= a.numberOfElements(); k++)
   {
      if (a.getChar(k) == ch)
         a.setChar(k, newch);
   }
}
```

Working with Cell Arrays

The `MWCellArray` class provides a Java interface to a MATLAB cell array.

Using MWCellArray Constructors

The `MWCellArray` class provides the following constructors:

Constructor	Usage
`MWCellArray()`	Empty cell array.
`MWCellArray(int[])`	New cell array with specified dimensions. All cells are initialized to empty.
`MWCellArray(gint, int)`	New cell matrix with specified number of rows and columns.

Constructing a cell array is a two-step process. First, allocate the array using one of the constructors in the previous table, then assign values to each cell using one of the `set` methods.

Constructing an MWCellArray. For simple arrays, passing a Java array directly is the most convenient approach. When you want to assign a more complicated type to a cell (i.e., a complex array or another cell array), you must create a temporary `MWArray` for the input value. You should dispose of any temporary arrays after assigning them to a cell.

This example creates and initializes a 2-by-2 cell array:

```
String x11 = "A String";
double[][] x12 = {{1.0, 2.0},
                  {3.0, 4.0}};
int[][] x21 = {{1, 2},
               {3, 4}};
boolean[][] x22 = {{true, false},
                   {false, true}};

int[] index = {1, 1};
a.set(index, x11);
index[1] = 2;
a.set(index, x12);
index[0] = 2;
a.set(index, x22);
index[1] = 1;
a.set(index, x21);
```

Here, each cell is initialized with a Java array, and default conversion rules are used to create the MATLAB array for each cell.

Constructing an MWCellArray Containing Complex Arrays. The next example creates a helper function that constructs a cell array containing a list of complex `double` arrays. The real and imaginary parts of each cell are passed in the `re` and `im` arrays, respectively. The new cell array has dimensions 1-by-`N`, where `N` is the length of the input arrays, which must be the same.

```
MWCellArray createNumericCell(Object[] re, Object[] im)
   throws MWException
{
   if (re == null || im == null)
      throw new MWException("Invalid input");
   if (re.length != im.length)
      throw new MWException(
         "Input arrays must be the same length");

   MWCellArray a = null;
   MWNumericArray x = null;

   try
   {
      a = new MWCellArray(1, re.length);
      for (int k = 1; k <= re.length; k++)
      {
         x = new MWNumericArray(re[k-1], im[k-1],
            MWClassID.DOUBLE);
         a.set(k, x);
         x.dispose();
         x = null;
      }
      return a;
   }

   catch (Exception e)
   {
      if (a != null)
         a.dispose();
```

```
            if (x != null)
               x.dispose();
            throw new MWException(e.getMessage());
         }
      }
```

This method creates a new `MWCellArray` of the necessary size. Next, the code loops over the number of elements in the supplied arrays. For each loop iteration, a temporary `MWNumericArray`, `x`, is created for the current cell and initialized with the contents of `re[k-1]` and `im[k-1]` (the loop variable, `k`, is one-based). A shared copy of the temporary numeric array is then assigned to the cell at `k` using the `set` method.

The native resources associated with `x` are freed when you call `dispose`. If an exception is thrown during the construction phase, the partially constructed cell array and the temporary numeric array are disposed of, if necessary. The exception is then rethrown as an `MWException`. If everything goes well, the `MWCellArray` is returned.

Accessing MWCellArray Elements

The `MWCellArray` class provides methods for accessing and modifying array data in the form of `get` and `set` methods. The following table lists the `get` and `set` methods.

Method	Usage
`get(int)`	Returns the element at the one-based index as a Java array (inherited from `MWArray`).
`get(int)`	Returns the element at the one-based index array as a Java array (inherited from `MWArray`).
`getCell(int)`	Returns the element at the one-based index as an `MWArray` instance.
`getCell(int[])`	Returns the element at the one-based index array as an `MWArray` instance.

Method	Usage
`set(int, Object)`	Replaces the element at the one-based index with the supplied value (inherited from `MWArray`).
`set(int[], Object)`	Replaces the element at the one-based index array with the supplied value (inherited from `MWArray`).

The `set` methods replace the cell at the specified index with the supplied value. The cell value can be passed as any supported Java type or as an `MWArray` instance. When the cell value is passed as a Java type, the value is converted to a MATLAB array using default conversion rules. When the value is passed as an `MWArray`, the cell is assigned a shared copy of the underlying MATLAB array.

Using getCell. The `getCell` methods return an `MWArray` instance of the proper subclass type representing a shared copy of the underlying cell. The array returned by `getCell` should be disposed of when it is no longer needed. This is the most efficient way of accessing a cell, because an `MWArray` object is created to encapsulate a shared copy of the underlying array. This process is significantly more efficient than converting the entire array to a Java array each time you access the cell. The next example prints information about a cell array to standard output:

```
void printCellInfo(MWCellArray a)
{
   if (a == null)
      return;
   MWArray c = null;
   int n = a.numberOfElements();
   System.out.println("Number of elements: " + n);

   try
   {
      for (int k = 1; k <= n; k++)
      {
         c = a.getCell(k);
         System.out.println("cell: " + k + " type: " +
            a.classID());
         c.dispose();
         c = null;
      }
   }

   finally
   {
      if (c != null)
         c.dispose();
   }
}
```

This method loops through the array and prints the type of each cell. The temporary array returned by `getCell` is disposed of after it is used. The `finally` clause ensures that the array is disposed of before exiting, in the case of an exception. `MWCellArray` also overrides the `MWArray.get` methods. In this case, a Java array is returned that represents a conversion of the underlying cell, as would be returned by `toArray`.

Using get. You can think of `get` as being implemented as follows:

```
   Object get(int index)
   {
      MWArray a = null;
      try
      {
         a = this.getCell(index);
         return a.toArray();
      }

      finally
      {
         if (a != null)
            a.dispose();
      }
   }
```

Using `get`, you can retrieve the cells from the first `MWCellArray` example as Java arrays.

```
   int[] index = {1, 1};
   String x11 = (String)a.get(index);
```

```
index[1] = 2;
double[][] x12 = (double[][])a.get(index);
index[0] = 2;
boolean[][] x22 = (boolean[][])a.get(index);
index[1] = 1;
int[][] x21 = (int[][])a.get(index);
```

As with `set`, default conversion rules are applied (same rules as apply to `toArray`). In this example, the default rules are fine. In the case where complex arrays, other cell arrays, or struct arrays are stored in the cell array, it is recommended to use `getCell` to return an `MWArray` instance.

toArray and getData Methods

In addition to `get` and `getCell`, the `toArray` and `getData` methods are implemented on `MWCellArray` to return a conversion of the entire cell array. These methods provide a bulk conversion to an array of Java arrays, although the output can be difficult to dissect in some cases (particularly in the case of nested cell arrays).

The `getData` method returns a one-dimensional array of type `Object`. Each element of the return cell array is converted by calling `toArray` on the corresponding cell.

The `toArray` method returns the same array, except that it has the same dimensionality as the underlying cell array.

Working with Struct Arrays

The `MWStructArray` class manages a native MATLAB struct array.

The following code excerpts show how to perform various data manipulations with the MWStructArray class. for a complete end-to-end example of an application that utilizes many of the methods presented in this section, see the "Phonebook Example" on page 5-28.

Constructing an MWStructArray

The following demonstrates how to set up constructors for the MWStructArray class. The `dispose` method calls are optional.

```
try
        {
            int rows = 2;
            int cols = 3;
            int[] dims = {rows, cols};
            String[] fieldNames = {"name", "phone"};

            MWStructArray a1 = new MWStructArray();
            MWStructArray a2 = new MWStructArray(dims, fieldNames);
            MWStructArray a3 = new MWStructArray(rows, cols, fieldNames);

            a1.dispose();
            a2.dispose();
            a3.dispose();
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
        finally
        {
            /* Free native resources */
//             a1.dispose();
//             a2.dispose();
//             a3.dispose();
        }
```

Getting Information About a Structure

Use the following code snippet as a model for how to extract information about an existing structure. The `dispose` method calls are optional.

```
try
        {
            int rows = 2;
            int cols = 3;
            String[] fieldNames = {"name", "phone"};

            MWStructArray arr = new MWStructArray(rows, cols, fieldNames);
```

```
                System.out.println("Number of Elements: " + arr.numberOfElements());
                System.out.println("Number of Fields: " + arr.numberOfFields());
                java.lang.String[] fieldNames2 = arr.fieldNames();

                System.out.print("Field names: ");
                for (int j = 0; j < fieldNames2.length; j++)
                {
                 System.out.print(fieldNames2[j] + " ");
                }
                System.out.println("");

                arr.dispose();


            }
            catch (Exception e)
            {
                System.out.println("Exception: " + e.toString());
            }
            finally
            {
                /* Free native resources */
//                 arr.dispose();
            }
```

From this code, the following output is produced:

```
Number of Elements: 6
Number of Fields: 2
Field names: name phone
```

Modifying Elements in an MWStructArray

Use the following code snippet as an example of how to modify elements in an MWStructArray:

```
try
        {

            /* Java Hashtable containing some name-number pairs: */
            Map friendsList = new TreeMap();
```

```
friendsList.put("Jordan Robert", new Integer(3386));
friendsList.put("Mary Smith", new Integer(3912));
friendsList.put("Stacy Flora", new Integer(3238));
friendsList.put("Harry Alpert", new Integer(3077));

/* Create an MWStructArray to hold the same data */
int numberOfFriends = friendsList.size();
int numberOfColumns = 1;
String[] fieldNames = {"name", "phone"};
MWStructArray friends = new MWStructArray(numberOfFriends,
                        numberOfColumns,
                        fieldNames);

/* Populate MWStructArray with data from a Java set */
Set names = friendsList.keySet();
Iterator itr = names.iterator();
Integer index = 0;
while (itr.hasNext()) {
    String key = (String)itr.next();
    index++;
    friends.set("name", index, new MWCharArray(key));
    friends.set("phone", index, (Integer)friendsList.get(key));
}

System.out.println("- MWStructArray: friends -");
dispStruct(friends);

 /* Populate another MWStructArray with data from a Java array */
MWStructArray friends2 = new MWStructArray(numberOfFriends,
                        numberOfColumns,
                        fieldNames);



/* Populate MWStructArray with data from a Java set */
//String[] namesArr = (String[])(names.toArray());
String[] namesArr = names.toArray(new String[0]);
/* Java uses 0-based indices, but MWStructArrays are 1-based */
for (int j = 0; j < namesArr.length; j++)
{
```

```
            friends2.set("name", j+1, new MWCharArray(namesArr[j]));
            friends2.set("phone", j+1, (Integer)friendsList.get(namesArr[j]));
        }

        System.out.println("- MWStructArray: friends2 -");
        dispStruct(friends2);

        friends.dispose();
        friends2.dispose();
    }
    catch (Exception e)
    {
        System.out.println("Exception: " + e.toString());
    }
    finally
    {
        /* Free native resources */
//          friends.dispose();
    }
```

This code produces the following output:

```
- MWStructArray: friends -
Number of Elements: 4
Number of Fields: 2
Standard MATLAB view:
4x1 struct array with fields:
    name
    phone
Walking structure:
Element 1
   name: Harry Alpert
   phone: 3077
Element 2
   name: Jordan Robert
   phone: 3386
Element 3
   name: Mary Smith
   phone: 3912
```

```
Element 4
   name: Stacy Flora
   phone: 3238
- MWStructArray: friends2 -
Number of Elements: 4
Number of Fields: 2
Standard MATLAB view:
4x1 struct array with fields:
    name
    phone
Walking structure:
Element 1
   name: Harry Alpert
   phone: 3077
Element 2
   name: Jordan Robert
   phone: 3386
Element 3
   name: Mary Smith
   phone: 3912
Element 4
   name: Stacy Flora
   phone: 3238
```

Copying Elements from an MWStructArray

Use either of the following methods to copy elements from a struct array:

sharedCopy. The following code demonstrates how to do a copy by reference (a shared copy) of an element in a struct array. The `dispose` method calls are optional.

```
try
    {

        String[] fieldnames = {"f"};
        MWStructArray A = new MWStructArray(1,1, fieldnames);
        A.set("f",1,new MWCharArray("one"));
        MWStructArray S = (MWStructArray)A.sharedCopy();
        System.out.println("Original struct has field data \"" +  A.getField("f",1).toString() + "\"
            System.out.println("Shared copy struct has field data \"" +  S.getField("f",1).toString() +

            MWCharArray newVal = new MWCharArray("two");
            S.set("f",1, newVal);
            System.out.println("After changing shared copy:");
            System.out.println("Original struct has field data \"" +  A.getField("f",1).toString() + "\"
            System.out.println("Shared copy struct has field data \"" +  S.getField("f",1).toString() +

            A.dispose();
            S.dispose();
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
        finally
        {
            /* Free native resources */
//             A.dispose();
//             S.dispose();
        }
```

This code produces the following output:

```
Original struct has field data "one"
Shared copy struct has field data "one"
After changing shared copy:
Original struct has field data "two"
Shared copy struct has field data "two"
```

clone. The following code demonstrates how to do a copy by value (deep copy) of an element in a struct array. The `dispose` method calls are optional.

```
 try
        {
            /* USE: clone */
            System.out.println("--- USE: clone ---");

            String[] fieldnames = {"f"};
            MWStructArray A = new MWStructArray(1,1, fieldnames);
            A.set("f",1,new MWCharArray("one"));
            MWStructArray C = (MWStructArray)A.clone();
            System.out.println("Original struct has field data \"" +  A.getField("f",1).toString() + "\"
            System.out.println("Cloned struct has field data \"" +  C.getField("f",1).toString() + "\"")

            MWCharArray newVal = new MWCharArray("two");
            C.set("f",1, newVal);
            System.out.println("After changing clone:");
            System.out.println("Original struct has field data \"" +  A.getField("f",1).toString() + "\"
            System.out.println("Cloned struct has field data \"" +  C.getField("f",1).toString() + "\"")

            A.dispose();
            C.dispose();
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
        finally
        {
         /* Free native resources */
//       A.dispose();
//       C.dispose();
        }
```

This code produces the following output:

```
Original struct has field data "one"
Cloned struct has field data "one"
After changing clone:
Original struct has field data "one"
Cloned struct has field data "two"
```

Creating Nested Structures in an MWStructArray

This code snippet illustrates how to create parent and child nested structures within an MWStructArray, how to display the structures, and how to modify the child structure.

```
    try
        {
         /* Create child structure */
```

```
                String[] childFieldNames = {"a", "b"};
                MWStructArray child = new MWStructArray(1,1, childFieldNames);
                child.set("a",1,new MWNumericArray(new Integer(12)));
                child.set("b",1,new MWNumericArray(new Integer(14)));

                /* Create parent structure */
                String[] parentFieldNames = {"c", "substruct"};
                MWStructArray parent = new MWStructArray(1,1, parentFieldNames);
                parent.set("c",1,new MWNumericArray(new Integer(19)));
                parent.set("substruct",1,child);

                /* Display nested structures */
                System.out.println("- Parent struct: -");
                 dispStruct(parent);

                 /* Modify the child */
                 MWStructArray toModify = (MWStructArray)parent.getField("substruct",1);
                 toModify.set("a", 1, new MWNumericArray(new Integer(25)));

                 /* Print */
                System.out.println("- After modifying child: -");
                 dispStruct(parent);

                parent.dispose();
                child.dispose();
               }
               catch (Exception e)
               {
                System.out.println("Exception: " + e.toString());
               }
               finally
               {
                /* Free native resources */
//              A.dispose();
//              C.dispose();
               }
               System.out.println("--- Done. ---");
           }
```

This code produces the following output:

```
- Parent struct: -
Number of Elements: 1
Number of Fields: 2
Standard MATLAB view:
            c: 19
    substruct: [1x1 struct]
Walking structure:
Element 1
   c: 19
   substruct: nested structure:
+++ Begin of "substruct" nested structure
Number of Elements: 1
Number of Fields: 2
Standard MATLAB view:
    a: 12
    b: 14
Walking structure:
Element 1
   a: 12
   b: 14
+++ End of "substruct" nested structure
- After modifying child: -
Number of Elements: 1
Number of Fields: 2
Standard MATLAB view:
            c: 19
    substruct: [1x1 struct]
Walking structure:
Element 1
   c: 19
   substruct: nested structure:
+++ Begin of "substruct" nested structure
Number of Elements: 1
Number of Fields: 2
Standard MATLAB view:
    a: 25
    b: 14
Walking structure:
Element 1
   a: 25
   b: 14
+++ End of "substruct" nested structure
--- Done. ---
```

Accessing Elements in an MWStructArray

Use the following example code as a model for accessing struct array elements:

```
public static void dispStruct(MWStructArray arr) {
   System.out.println("Number of Elements: " + arr.numberOfElements());
   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());
           }
```

Using Class Methods

Using MWArray

This section covers the following topics on `MWArray`:

- "Constructing an MWArray" on page 4-48
- "Methods to Create and Destroy an MWArray" on page 4-48
- "Methods to Return Information About an MWArray" on page 4-50
- "Methods to Get and Set Data in the MWArray" on page 4-54
- "Methods to Copy, Convert, and Compare MWArrays" on page 4-59
- "Methods to Use on Sparse MWArrays" on page 4-64

Constructing an MWArray

Use this constructor to create an empty two-dimensional `MWArray` object:

```
MWArray()
```

The type given to this object is `MWClassID.UNKNOWN`.

Example. Construct an empty `MWArray` object:

```
MWArray A = new MWArray();
```

Methods to Create and Destroy an MWArray

Use these methods to destroy an object of class `MWArray` or any of its child classes.

Method	Description
"dispose" on page 4-49	Frees the native MATLAB array contained by this array.
"disposeArray" on page 4-49	Frees all native MATLAB arrays contained in the input object.

dispose. This method destroys the native MATLAB array contained by the array object and frees the memory occupied by the array.

The prototype for the `dispose` method is as follows:

```
public void dispose()
```

Input Parameters

None

Example — Constructing an MWArray Object

Construct and then destroy an `MWArray` object:

```
MWArray A = new MWArray();

A.dispose();
```

disposeArray. This method destroys any native MATLAB arrays contained in the input object and frees the memory occupied by them. This is a static method of the class and thus does not need to be invoked in reference to an instance of the class.

The prototype for the `disposeArray` method is as follows:

```
public static void disposeArray(Object arr)
```

Input Parameters

`arr`

Object to be disposed of

If the input object represents a single `MWArray` instance, then that instance is freed when you call its `dispose()` method.

If the input object represents an array of `MWArray` instances, each object in the array is disposed of.

If the input object represents an array of `Object` or a multidimensional array, the array is recursively processed to free each `MWArray` contained in the array.

Example — Constructing an MWNumericArray Object

Construct and then destroy an array of numeric objects:

```
MWArray[] MArr = new MWArray[10];
for (int i = 0; i < 10; i++)
   MArr[i] = new MWNumericArray();

MWArray.disposeArray(MArr);
```

Methods to Return Information About an MWArray

Use these methods to return information about an object of class `MWArray` or any of its child classes.

Method	**Description**
"classID" on page 4-51	Returns the MATLAB type of the array.
"getDimensions" on page 4-51	Returns the size of each dimension of the array.
"isEmpty" on page 4-52	Tests if the array has no elements.

Method	Description
"numberOfDimensions" on page 4-53	Returns the number of dimensions of the array.
"numberOfElements" on page 4-53	Returns the total number of elements in the array.

The examples in the following sections use a 3-by-6 `MWNumericArray` object A, as constructed by this Java code:

```
int[][] Adata = {{ 1,  2,  3,  4,  5,  6},
                 { 7,  8,  9, 10, 11, 12},
                 {13, 14, 15, 16, 17, 18}};

MWNumericArray A = new MWNumericArray(Adata, MWClassID.INT32);
```

classID. This method returns the MATLAB type of the `MWArray` object. The return type is a field defined by the `MWClassID` class.

The prototype for the `classID` method is as follows:

```
public MWClassID classID()
```

Input Parameters

None

Example — Getting the Class ID of an MWArray

Return the class ID for an `MWNumericArray` object created previously:

```
System.out.println("Class of A is " + A.classID());
```

When run, the example displays this output:

```
Class of A is int32
```

getDimensions. This method returns a one-dimensional `int` array containing the size of each dimension of the `MWArray` object.

The prototype for the `getDimensions` method is as follows:

```
public int[] getDimensions()
```

Input Parameters

None

Example — Getting Array Dimensions of an MWArray

```
int[] dimA = A.getDimensions();

System.out.println("Dimensions of A are " +
                   dimA[0] + " x " + dimA[1]);
```

When run, the example displays this output:

```
Dimensions of A are 3 x 6
```

isEmpty. This method returns `true` if the array object contains no elements, and `false` otherwise.

The prototype for the `isEmpty` method is as follows:

```
public boolean isEmpty()
```

Input Parameters

None

Example — Testing for an Empty MWArray

Display a message if array object `A` is an empty array. Otherwise, display the contents of `A`:

```
if (A.isEmpty())
   System.out.println("Matrix A is empty");
else
```

```
    System.out.println("A = " + A.toString());
```

When run, the example displays the contents of A:

```
A =    1     2     3     4     5     6
       7     8     9    10    11    12
      13    14    15    16    17    18
```

numberOfDimensions. This method returns the number of dimensions of the array object.

The prototype for the numberOfDimensions method is as follows:

```
public int numberOfDimensions()
```

Input Parameters

None

Example — Getting the Number of Dimensions of an MWArray

Display the number of dimensions for array object A:

```
System.out.println("Matrix A has " + A.numberOfDimensions() +
                   " dimensions");
```

When run, the example displays this output:

```
Matrix A has 2 dimensions
```

numberOfElements. This method returns the total number of elements in the array object.

The prototype for the numberOfElements method is as follows:

```
public int numberOfElements()
```

Input Parameters

None

Example — Getting the Number of MWArray Elements

Display the number of elements in array object A:

```
System.out.println("Matrix A has " + A.numberOfElements() +
                   " elements");
```

When run, the example displays this output:

```
Matrix A has 18 elements
```

Methods to Get and Set Data in the MWArray

Use these methods to get and set values in an object of class `MWArray` or any of its child classes.

Method	Description
"get" on page 4-54	Returns the element at the specified one-based offset or index array as an `Object`.
"getData" on page 4-56	Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array. The array is in column-wise order.
"set" on page 4-57	Replaces the element at the specified one-based offset, or index array, in this array with the specified element.
"toArray" on page 4-58	Returns an array containing a copy of the data in the underlying MATLAB array. The returned array has the same dimensionality as the MATLAB array.

get. This method returns the element located at the specified one-based offset or index array in the array object. The element is returned as an `Object`.

To get the element at a specific index, use one of the following:

```
public Object get(int index)
public Object get(int[] index)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`index`

Index of the requested element in the `MWArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the *i*th dimension.

Exceptions

The `get` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Getting an MWArray Value with get

```
int[] cdims = {1, 3};
         MWArray C = new MWArray(cdims);
         Integer val = new Integer(15);
         int[] index2  = {1, 3};
         C.set(index2, val);
         Object x = C.get(index2);
         if (x instanceof int[][])
         {
          int[][] y = (int[][])x;
             System.out.println("B: Cell data C(1,3) is " + y[0][0]);
         }
```

When run, the example displays this output:

```
B: Cell data C(1,3) is 15
```

getData. This method returns all elements of the `MWArray` object. Elements are returned in a one-dimensional array, in column-wise order. Elements are returned as type `Object`.

The prototype for the `getData` method is as follows:

```
public Object getData()
```

Input Parameters

None

The elements of the returned array are converted according to default conversion rules. If the underlying MATLAB array is a complex numeric type, `getData` returns the real part.

Example — Getting an MWArray Value with getData

Get the data from `MWArray` object A, casting the type from `Object` to `int`:

```
System.out.println("Data read from matrix A is:");
```

```
int[] x = (int[]) A.getData();
for (int i = 0; i < x.length; i++)
   System.out.print(" " + x[i]);

System.out.println();
```

When run, the example displays this output:

```
Data read from matrix A is:
 1 7 13 2 8 14 3 9 15 4 10 16 5 11 17 6 12 18
```

set. This method replaces the element at a specified index in the `MWArray` object with the input element.

To set the element at a specific index, use one of the following:

```
public void set(int index, Object element)
public void set(int[] index, Object element)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`element`

New element to replace at `index`

`index`

Index of the requested element in the `MWArray`.

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

Exceptions

The `set` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Setting an MWArray Value

Modify the data in element `(2, 4)` of `MWArray` object `A`:

```
int[] index = {2, 4};
A.set(index, 555);

Object d_out = A.get(index);
System.out.println("Data read from A(2,4) is " +
                    d_out.toString());
```

When run, the example displays this output:

```
Data read from A(2,4) is 555
```

toArray. This method creates an array with the same dimensionality as the MATLAB array.

The prototype for the `toArray` method is as follows:

```
public Object[] toArray()
```

The elements of the returned array are converted according to default conversion rules. If the underlying MATLAB array is a complex numeric type, `toArray` returns the real part.

Input Parameters

None

Example — Getting an MWArray with toArray

Create and display a copy of `MWArray` object `A`:

```
int[][] x = (int[][]) A.toArray();
int[] dimA = A.getDimensions();

System.out.println("Matrix A is:");
for (int i = 0; i < dimA[0]; i++)
   {
   for (int j = 0; j < dimA[1]; j++)
      System.out.print(" " + x[i][j]);
   System.out.println();
   }
```

When run, the example displays this output:

```
Matrix A is:
 1 2 3 4 5 6
 7 8 9 10 11 12
 13 14 15 16 17 18
```

Methods to Copy, Convert, and Compare MWArrays

Use these methods to copy, convert, and compare objects of class `MWArray` or any of its child classes.

Method	Description
"clone" on page 4-60	Creates and returns a deep copy of this array.
"compareTo" on page 4-61	Compares this array with the specified array for order.

Method	Description
"equals" on page 4-62	Indicates whether some other array is equal to this one.
"hashCode" on page 4-62	Returns a hash code value for the array.
"sharedCopy" on page 4-63	Creates and returns a shared copy of this array.
"toString" on page 4-64	Returns a string representation of the array.

clone. This method creates and returns a deep copy of the `MWArray` object. Because `clone` allocates a new array, any changes made to this new array are not reflected in the original.

The prototype for the `clone` method is as follows:

```
public Object clone()
```

Input Parameters

None

Exceptions

The `clone` method throws the following exception:

`java.lang.CloneNotSupportedException`

The object's class does not implement the `Cloneable` interface.

Example — Cloning an MWArray Object

Create a clone of `MWArray` object `A`:

```
Object C = A.clone();
```

```
System.out.println("Clone of matrix A is:");
System.out.println(C.toString());
```

When run, the example displays this output:

```
Clone of matrix A is:
   1     2     3     4     5     6
   7     8     9    10    11    12
  13    14    15    16    17    18
```

compareTo. This method compares the `MWArray` object with the input object. It returns a negative integer, zero, or a positive integer if the `MWArray` object is less than, equal to, or greater than the specified object, respectively.

The prototype for the `compareTo` method is as follows:

```
public int compareTo(Object obj)
```

See the `compareTo` method in interface `java.lang.Comparable` for a full description of the return value.

Input Parameters

`obj`

Array to compare this `MWArray` object to

Example — Comparing MWArrays with compareTo

Create a shared copy of the `MWArray` object and then compare it to the original object. A return value of zero indicates that the two objects are equal:

```
Object S = A.sharedCopy();

if (A.compareTo(S) == 0)
   System.out.println("Matrix S is equal to matrix A");
```

When run, the example displays this output:

```
Matrix S is equal to matrix A
```

equals. This method indicates the `MWArray` object is equal to the input object. The `equals` method of the `MWArray` class overrides the `equals` method of class `Object`.

The prototype for the `equals` method is as follows:

```
public boolean equals(Object object)
```

Input Parameters

`object`

Array to compare this `MWArray` object to

Example — Comparing MWArrays with equals

Create a shared copy of the `MWArray` object and then compare it to the original object. A return value of `true` indicates that the two objects are equal:

```
Object S = A.sharedCopy();

if (A.equals(S))
   System.out.println("Matrix S is equal to matrix A");
```

When run, the example displays this output:

```
Matrix S is equal to matrix A
```

hashCode. This method returns a hash code value for the `MWArray` object. The `hashCode` method of the `MWArray` class overrides the `hashCode` method of class `Object`.

The prototype for the `hashCode` method is as follows:

```
public int hashCode()
```

Input Parameters

None

Example — Getting an MWArray Hash Code

Obtain the hash code for `MWArray` object `A`:

```
System.out.println("Hash code for matrix A is " + A.hashCode());
```

When run, the example displays this output:

```
Hash code for matrix A is 456687478
```

sharedCopy. This method creates and returns a shared copy of the array. The shared copy points to the underlying original MATLAB array. Any changes made to the copy are reflected in the original.

The prototype for the `sharedCopy` method is as follows:

```
public Object sharedCopy()
```

Input Parameters

None

Example — Making a Shared Copy of an MWArray

Create a shared copy of `MWArray` object `A`:

```
Object S = A.sharedCopy();

System.out.println("Shared copy of matrix A is:");
System.out.println(S.toString());
```

When run, the example displays this output:

```
Shared copy of matrix A is:
   1     2     3     4     5     6
```

```
      7       8       9      10      11      12
     13      14      15      16      17      18
```

toString. This method returns a string representation of the array. The `toString` method of the `MWArray` class overrides the `toString` method of class `Object`.

The prototype for the `toString` method is as follows:

```
public java.lang.String toString()
```

Input Parameters

None

Example — Converting an MWArray to a String

Display the contents of `MWArray` object `A`:

```
System.out.println("A = " + A.toString());
```

When run, the example displays the contents of `A`:

```
A =    1       2       3       4       5       6
       7       8       9      10      11      12
      13      14      15      16      17      18
```

Methods to Use on Sparse MWArrays

Use these methods to return information on sparse arrays of type `MWArray` or any of its child classes.

Method	Description
"isSparse" on page 4-65	Tests whether the array is sparse.

Method	Description
“columnIndex” on page 4-66	Returns an array containing the column index of each nonzero element in the underlying MATLAB array.
“rowIndex” on page 4-67	Returns an array containing the row index of each nonzero element in the underlying MATLAB array.
“maximumNonZeros” on page 4-67	Returns the allocated capacity of a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements()`.
“numberOfNonZeros” on page 4-68	Returns the number of nonzero elements in a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements()`.

The examples that follow use the sparse `MWArray` object constructed below using the “newSparse” on page 4-77 method of `MWNumericArray`:

```
double[] Adata = { 0, 10, 0, 0, 40, 50, 60,  0,  0, 90};

int[] ri = {1, 1, 1, 1, 1, 2, 2, 2, 2, 2};
int[] ci = {1, 2, 3, 4, 5, 1, 2, 3, 4, 5};

MWNumericArray A = MWNumericArray.newSparse(ri, ci,
                     Adata, MWClassID.DOUBLE);

System.out.println(A.toString());
```

Here are the contents of the sparse `MWArray`:

```
(2,1)          50
(1,2)          10
(2,2)          60
(1,5)          40
(2,5)          90
```

isSparse. This method returns `true` if the `MWArray` object is sparse, and `false` otherwise.

The prototype for the `isSparse` method is as follows:

```
public boolean isSparse()
```

Input Parameters

None

Example — Testing an MWArray for Sparseness

Test the `MWArray` object `A` created previously for sparseness:

```
if (A.isSparse())
   System.out.println("Matrix A is sparse");
```

When run, the example displays this output:

```
Matrix A is sparse
```

columnIndex. This method returns an array containing the column index of each element in the underlying MATLAB array.

The prototype for the `columnIndex` method is as follows:

```
public int[] columnIndex()
```

Input Parameters

None

Example — Getting the Column Indices of a Sparse MWArray

Get the column indices of the elements of the sparse array:

```
System.out.print("Column indices are:  ");
int[] colidx = A.columnIndex();
for (int i = 0; i < 5; i++)
   System.out.print(colidx[i] + " ");
System.out.println();
```

When run, the example displays this output:

```
Column indices are:  1 2 2 5 5
```

rowIndex. This method returns an array containing the row index of each element in the underlying MATLAB array.

The prototype for the `rowIndex` method is as follows:

```
public int[] rowIndex()
```

Input Parameters

None

Example — Getting the Row Indices of a Sparse MWArray

Get the row indices of the elements of the sparse array:

```
System.out.print("Row indices are:  ");
int[] rowidx = A.rowIndex();
for (int i = 0; i < 5; i++)
   System.out.print(rowidx[i] + " ");
System.out.println();
```

When run, the example displays this output:

```
Row indices are:  2 1 2 1 2
```

maximumNonZeros. This method returns the allocated capacity of a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements`.

The prototype for the `maximumNonZeros` method is as follows:

```
public int maximumNonZeros()
```

Input Parameters

None

Example — Getting the Maximum Number of Nonzeros in an MWArray

Display the maximum number of nonzeros for this array:

```
System.out.println("Maximum number of nonzeros for matrix A is "
                    + A.maximumNonZeros());
```

When run, the example displays this output:

```
Maximum number of nonzeros for matrix A is 10
```

numberOfNonZeros. This method returns the number of nonzero elements in a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements`.

The prototype for the `numberOfNonZeros` method is as follows:

```
public int numberOfNonZeros()
```

Input Parameters.

None

Example — Getting the Number of Nonzeros in an MWArray

Display the number of nonzero values in this array:

```
System.out.println("The number of nonzeros for matrix A is " +
                    A.numberOfNonZeros());
```

When run, the example displays this output:

```
The number of nonzeros for matrix A is 5
```

Using MWNumericArray

This section covers the following topics:

- "Constructing an MWNumericArray" on page 4-69
- "Methods to Create and Destroy an MWNumericArray" on page 4-74
- "Methods to Get and Set the Real Parts of an MWNumericArray" on page 4-86
- "Methods to Get and Set the Imaginary Parts of an MWNumericArray" on page 4-90
- "Methods to Copy, Convert, and Compare MWNumericArrays" on page 4-98
- "Methods to Use on Sparse MWNumericArrays" on page 4-101
- "Methods to Return Special Constant Values" on page 4-101

Constructing an MWNumericArray

Use the tables in this section to construct an `MWNumericArray` from a particular Java data type. See the examples at the end of this section for more help.

- "Constructing an Empty Scalar" on page 4-69
- "Constructing a Real or Complex Numeric Scalar" on page 4-71
- "Constructing a Real or Complex Numeric Array" on page 4-72

In addition to using the `MWNumericArray` constructor, you can also use "newSparse" on page 4-77 to construct an `MWNumericArray`. These two methods provide better performance than the constructor.

Constructing an Empty Scalar. Use either of the following constructors to create an empty scalar `MWNumericArray`:

To construct an empty scalar of type `MWClassID.DOUBLE`, use the following:

```
MWNumericArray()
```

To construct an empty scalar of type `classid`, use the following:

```
MWNumericArray(MWClassID classid)
```

Example — Constructing an Empty Numeric Array Object

Create an empty scalar of type `int64`:

```
MWNumericArray A = new MWNumericArray(MWClassID.INT64);
System.out.println("A = " + A);
```

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

```
A = []
```

Constructing a Real or Complex Numeric Scalar. Use this constructor syntax to create a real scalar `MWNumericArray` from a primitive Java type:

```
MWNumericArray(javatype realValue)
```

Or use this syntax to create a complex scalar `MWNumericArray` from a primitive Java type:

```
MWNumericArray(javatype realValue, javatype imagValue)
```

The class ID for the returned `MWNumericArray` is shown in the following table:

javatype Input	Class ID of MWNumericArray
`double`	`MWClassID.DOUBLE`
`float`	`MWClassID.SINGLE`
`long`	`MWClassID.INT64`
`int`	`MWClassID.INT32`
`short`	`MWClassID.INT16`
`byte`	`MWClassID.INT8`

Exceptions

The `MWNumericArray` constructor throws the following exception:

`ArrayStoreException`

A nonnumeric array type was specified.

Example — Constructing an Integer Array Object

Construct a scalar numeric array of type `MWClassID.INT16`:

```
double AReal = 24;

MWNumericArray A = new MWNumericArray(AReal, MWClassID.INT16);
System.out.println("Array A of type " + A.classID() + " = \n" + A);
```

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

```
Array A of type int16 =
    24
```

Example — Constructing a Complex Array Object

Construct a numeric scalar having real and imaginary parts:

```
double AReal = 24;
double AImag = 5;

MWNumericArray A = new MWNumericArray(AReal, AImag);
System.out.println("Array A of type " + A.classID() + " = \n" + A);
```

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

```
Array A of type double =
   24.0000 + 5.0000i
```

Constructing a Real or Complex Numeric Array. Use this constructor syntax to create a real nonscalar `MWNumericArray` from a primitive Java type:

```
MWNumericArray(javatype realValue, MWClassID classid)
```

Or use this syntax to create a complex nonscalar `MWNumericArray` from a primitive Java type:

```
MWNumericArray(javatype realValue, javatype imagValue,
                    MWClassID classid)
```

The type *javatype* can be any of the following:

- `double`
- `float`
- `long`
- `int`
- `short`
- `byte`
- `boolean`
- `Object`

Example — Constructing a Real Array of a Specific Type

Construct a 3-by-6 real array of type `MWClassID.SINGLE`:

```
double[][] AData = {{ 1,  2,  3,  4,  5,  6},
                    { 7,  8 , 9, 10, 11, 12},
                    {13, 14, 15, 16, 17, 18}};

MWNumericArray A = new MWNumericArray(AData, MWClassID.SINGLE);
System.out.println("Array A = \n" + A);
```

When run, the example displays this output:

```
A =    1     2     3     4     5     6
       7     8     9    10    11    12
      13    14    15    16    17    18
```

Example — Constructing a Complex Array of a Specific Type

Construct a 1-by-3 complex array of `MWClassID.DOUBLE`:

```
double[] AReal = {24.2, -7, 113};
double[] AImag = {5, 31, 27};
```

```
MWNumericArray A =
   new MWNumericArray(AReal, AImag, MWClassID.DOUBLE);

System.out.println("Array A of type " + A.classID() + " = \n" + A);
```

When run, the example displays this output:

```
Array A of type double =
  1.0e+002 *
    0.2420 + 0.0500i  -0.0700 + 0.3100i   1.1300 + 0.2700i
```

Methods to Create and Destroy an MWNumericArray

In addition to the `MWNumericArray` constructor, you can use the `newInstance` and `newSparse` methods to construct a numeric array. These two methods offer better performance than using the class constructor. To destroy the arrays, use either `dispose` or `disposeArray`, inherited from class `MWArray`.

Method	Description
"newInstance" on page 4-74	Constructs an array with the specified dimensions and complexity.
"newSparse" on page 4-77	Constructs a real sparse numeric matrix with the specified number of rows and columns and maximum nonzero elements, and initializes the array with the supplied data.
"dispose" on page 4-82	Frees the native MATLAB array contained by this array.
"disposeArray" on page 4-82	Frees all native MATLAB arrays contained in the input object.

newInstance. This method constructs a real or complex array, specifying the array dimensions, type, and complexity. This is a static method of the class and thus does not need to be invoked in reference to an instance of the class.

Note This method offers better performance than using the class constructor.

To construct an uninitialized real or complex numeric array, use the following:

```
newInstance(int[] dims, MWClassID classid, MWComplexity cmplx)
```

To construct and initialize a real numeric array, use

```
newInstance(int[] dims, Object rData, MWClassID classid)
```

To construct and initialize a complex numeric array, use

```
newInstance(int[] dims, Object rData, Object iData,
            MWClassID classid)
```

Input Parameters

`dims`

Array of nonnegative dimension sizes

`classId`

`MWClassID` representing the MATLAB type of the array

`rData`

Data to initialize the real part of the array. You must format the `rData` array in column-wise order.

`iData`

Data to initialize the imaginary part of the array. You must format the `iData` array in column-wise order.

Valid types for `realData` and `imagData` are as follows:

- `double[]`
- `float[]`
- `long[]`
- `int[]`

- `short[]`
- `byte[]`
- `boolean[]`
- One-dimensional arrays of any subclass of `java.lang.Number`
- One-dimensional arrays of `java.lang.Boolean`

Exceptions

The `newInstance` method throws the following exceptions:

`NegativeArraySizeException`

The specified `dims` parameter is negative.

`ArrayStoreException`

The array type is nonnumeric.

Example — Constructing a Numeric Array Object with newInstance

Construct a 3-by-6 real numeric array using the `newInstance` method. Note that data in the Java array must be stored in column-wise order so that it will be in the correct order in the final `MWNumericArray` object.

```
int[] dims = {3, 6};
double[] Adata = { 1,  7, 13,
                   2,  8, 14,
                   3,  9, 15,
                   4, 10, 16,
                   5, 11, 17,
                   6, 12, 18};

MWNumericArray A =
   MWNumericArray.newInstance(dims, Adata, MWClassID.DOUBLE);

System.out.println("A = " + A);
```

When run, the example displays this output:

```
A =    1      2      3      4      5      6
       7      8      9     10     11     12
      13     14     15     16     17     18
```

newSparse. This method constructs a real or complex sparse `MWNumericArray`, with the specified number of rows and columns and maximum nonzero elements, and initializes the array with the supplied data. This is a static method of the class and thus does not need to be invoked in reference to an instance of the class.

Constructing a Sparse Matrix with No Nonzero Elements

To construct a sparse matrix with no nonzero elements, use

```
newSparse(int rows, int cols, int nzmax, MWClassID classid,
          MWComplexity cmplx)
```

Constructing a Sparse Matrix of Real Numbers

To construct a real sparse array from an existing nonsparse two-dimensional array, use

```
newSparse(Object realData, MWClassID classid)
```

To construct and initialize a new real sparse array, use

```
newSparse(int[] rowindex, int[] colindex, Object realData,
          MWClassID classid)
```

To construct and initialize a new real sparse array, specifying its dimensions. use

```
newSparse(int[] rowindex, int[] colindex, Object realData,
          int rows, int cols, MWClassID classid)
```

To construct and initialize a new real sparse array, specifying its dimensions and maximum number of nonzeros, use

```
newSparse(int[] rowindex, int[] colindex, Object realData,
          int rows, int cols, int nzmax, MWClassID classid)
```

Constructing a Sparse Matrix of Complex Numbers

To construct a complex sparse array from an existing nonsparse two-dimensional array, use

```
newSparse(Object realData, Object imagData, MWClassID classid)
```

To construct and initialize a new complex sparse array, use

```
newSparse(int[] rowindex, int[] colindex, Object realData,
          Object imagData, MWClassID classid)
```

To construct and initialize a new complex sparse array, specifying its dimensions, use

```
newSparse(int[] rowindex, int[] colindex, Object realData,
          Object imagData, int rows, int cols, MWClassID classid)
```

To construct and initialize a new complex sparse array, specifying its dimensions and maximum number of nonzeros, use

```
newSparse(int[] rowindex, int[] colindex, Object realData,
          Object imagData, int rows, int cols, int nzmax,
          MWClassID classid)
```

Input Parameters

`realData` and `imagData`

Data to initialize the real and imaginary parts of the array. See information on valid data types below.

`rowIndex` and `colIndex`

Arrays of one-based row and column indices

Row and column index arrays are used to construct the sparse array such that the following holds true, with space allocated for `nzmax` nonzeros:

```
S(rowIndex(k), columnIndex(k)) = realData(k) + imagData(k)*i
```

If you assign multiple values to a single `rowIndex` and `colIndex` pair, then the element at that index is assigned the sum of these values.

`rows` and `cols`

Number of rows and columns in the matrix

`nzmax`

Maximum number of nonzero elements

`classID`

`MWClassID` representing the MATLAB type of the array. The only `classID` currently supported is `MWClassID.DOUBLE`.

Valid types for the `realData` and `imagData` parameters are as follows:

- `double[]`
- `float[]`
- `long[]`
- `int[]`
- `short[]`
- `byte[]`
- `boolean[]`
- One-dimensional arrays of any subclass of `java.lang.Number`
- One-dimensional arrays of `java.lang.Boolean`
- One-dimensional arrays of `java.lang.String`

Exceptions

The `newSparse` method throws the following exceptions:

`NegativeArraySizeException`

Row or column size is negative.

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

`ArrayStoreException`

Incompatible array type or invalid array data

Example — Constructing a Sparse Array Object with newSparse

Creating a sparse complex `MWNumericArray`:

Construct a two-dimensional complex sparse `MWNumericArray` from the real and imaginary `double` vectors:

```
double[][] rData = {{ 0,  0,  0, 16,  0},
                    {71, 63, 32,  0,  0}};

double[][] iData = {{ 0,  0,  0, 41,  0},
                    { 0,  0, 32,  0,  2}};

MWNumericArray A =
   MWNumericArray.newSparse(rData, iData, MWClassID.DOUBLE);

System.out.println("A = " + A.toString());
```

When run, the example displays this output:

```
A =    (2,1)     71.0000
       (2,2)     63.0000
       (2,3)     32.0000 +32.0000i
```

```
        (1,4)      16.0000 +41.0000i
        (2,5)            0 + 2.0000i
```

Example — Using newSparse with Row and Column Indices

Construct a sparse `MWNumericArray` from vector `Adata`:

```
double[] Adata = { 0, 10, 0, 0, 40, 50, 60,  0,  0, 90};

int[] ri = {1, 1, 1, 1, 1, 2, 2, 2, 2, 2};
int[] ci = {1, 2, 3, 4, 5, 1, 2, 3, 4, 5};

MWNumericArray A = MWNumericArray.newSparse(ri, ci,
                   Adata, MWClassID.DOUBLE);

System.out.println("A = " + A.toString());
```

When run, the example displays this output:

```
(2,1)          50
(1,2)          10
(2,2)          60
(1,5)          40
(2,5)          90
```

Example — Assigning Multiple Values to a Single Array Element

Create a sparse `MWNumericArray` using the `rowindex` and `colindex` arguments, specifying multiple values for the array element at index (`2`, `5`). The result is that this element stores the sum of the values from `Adata(1)`, `Adata(7)`, `Adata(8)`, and `Adata(9)`, which is equal to `250`.

```
double[] Adata = { 0, 10, 0, 0, 40, 50, 60, 70, 80, 90};

int[] ri = {1, 2, 1, 1, 1, 2, 2, 2, 2, 2};
int[] ci = {1, 5, 2, 3, 5, 1, 2, 5, 5, 5};

MWNumericArray A =
   MWNumericArray.newSparse(ri, ci, Adata, 4, 5,
                            MWClassID.DOUBLE);
   System.out.println("A = " + A.toString());
```

When run, the example displays this output:

```
      (2,1)       50
      (2,2)       60
      (1,5)       40
      (2,5)      250
```

dispose. `MWNumericArray` inherits this method from the `MWArray` class.

disposeArray. `MWNumericArray` inherits this method from the `MWArray` class.

Methods to Return Information About an MWNumericArray

Use these methods to return information about an object of class `MWNumericArray`.

Method	Description
"classID" on page 4-83	Returns the MATLAB type of this array.
"complexity" on page 4-83	Returns the complexity of this array.
"getDimensions" on page 4-83	Returns an array containing the size of each dimension of this array.
"isEmpty" on page 4-83	Tests whether the array has no elements.
"isFinite" on page 4-83	Tests for finiteness in a machine-independent manner.
"isInf" on page 4-84	Tests for infinity in a machine-independent manner.
"isNaN" on page 4-85	Tests for `NaN` (not a number) in a machine-independent manner.
"numberOfDimensions" on page 4-86	Returns the number of dimensions of this array.
"numberOfElements" on page 4-86	Returns the total number of elements in this array.

classID. `MWNumericArray` inherits this method from the `MWArray` class.

complexity. This method returns the complexity of the `MWNumericArray` object as either `MWComplexity.REAL` for a real array, or `MWComplexity.COMPLEX` for a complex array.

The prototype for the `complexity` method is

```
public MWComplexity complexity()
```

Input Parameters

None

Example — Testing for a Complex Array

Determine whether matrix `A` is real or complex:

```
double AReal = 24;
double AImag = 5;

MWNumericArray A = new MWNumericArray(AReal, AImag);
System.out.println("A is a " + A.complexity() + " matrix");
```

When run, the example displays this output:

```
A is a complex matrix
```

getDimensions. `MWNumericArray` inherits this method from the `MWArray` class.

isEmpty. `MWNumericArray` inherits this method from the `MWArray` class.

isFinite. This method tests for finiteness in a machine-independent manner. This is a static method of the class and does not need to be invoked in reference to an instance of the class.

The prototype for the `isFinite` method is as follows:

```
public static boolean isFinite(double value)
```

Input Parameters

`value`

`double` value to test for finiteness

Example — Testing for Finite Array Values

Test `x` for finiteness:

```
double x = 25;

if (MWNumericArray.isFinite(x))
   System.out.println("The input value is finite");
```

When run, the example displays this output:

```
The input value is finite
```

isInf. This method tests for infinity in a machine-independent manner. This is a static method of the class and does not need to be invoked in reference to an instance of the class.

The prototype for the `isInf` method is as follows:

```
public static boolean isInf(double value)
```

Input Parameters

`value`

`double` value to test for infinity

Example — Testing for Infinite Array Values

Test x for infinity:

```
double x = 1.0 / 0.0;

if (MWNumericArray.isInf(x))
   System.out.println("The input value is infinite");
```

When run, the example displays this output:

```
The input value is infinite
```

isNaN. This method tests for NaN (Not a Number) in a machine-independent manner. This is a static method of the class and does not need to be invoked in reference to an instance of the class.

The prototype for the isNaN method is

```
public static boolean isNaN(double value)
```

Input Parameters

value

double value to test for NaN

Example — Testing for NaN Array Values

Test x for NaN:

```
double x = 0.0 / 0.0;

if (MWNumericArray.isNaN(x))
   System.out.println("The input value is not a number.");
```

When run, the example displays this output:

```
The input value is not a number.
```

numberOfDimensions. `MWNumericArray` inherits this method from the `MWArray` class.

numberOfElements. `MWNumericArray` inherits this method from the `MWArray` class.

Methods to Get and Set the Real Parts of an MWNumericArray

Use these methods to get and set real values in an object of class `MWNumericArray`.

Method	Description
"get" on page 4-87	Returns the element at the specified offset as an `Object`.
"getData" on page 4-87	Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.
"getDouble" on page 4-87	Returns the real part at the specified offset as a `double` value.
"getFloat" on page 4-88	Returns the real part at the specified offset as a `float` value.
"getLong" on page 4-88	Returns the real part at the specified offset as a `long` value.
"getInt" on page 4-88	Returns the real part at the specified offset as an `int` value.
"getShort" on page 4-88	Returns the real part at the specified offset as a `short` value.
"getByte" on page 4-89	Returns the real part at the specified offset as a `byte` value.
"set" on page 4-90	Replaces the real part at the specified offset with the specified value.
"toArray" on page 4-90	Returns an array containing a copy of the data in the underlying MATLAB array. The returned array has the same dimensionality as the MATLAB array.

The following syntax applies to all the above methods except `getData` and `toArray`.

Calling Syntax. To get the element at a specific index, use one of the following:

```
public type getType(int index)
public type getType(int[] index)
```

To set the element at a specific index, use one of the following:

```
public void set(int index, type element)
public void set(int[] index, type element)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWNumericArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the ith dimension.

Exceptions. The `MWNumericArray` constructor throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

get. `MWNumericArray` inherits this method from the `MWArray` class.

getData. `MWNumericArray` inherits this method from the `MWArray` class.

getDouble. This method returns the real part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `double`.

Use either of the following prototypes for the `getDouble` method, where `index` can be of type `int` or `int[]`:

```
public double getDouble(int index)
public double getDouble(int[] index)
```

getFloat. This method returns the real part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `float`.

Use either of the following prototypes for the `getFloat` method, where `index` can be of type `int` or `int[]`:

```
public float getFloat(int index)
public float getFloat(int[] index)
```

getLong. This method returns the real part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `long`.

Use either of the following prototypes for the `getLong` method, where `index` can be of type `int` or `int[]`:

```
public long getLong(int index)
public long getLong(int[] index)
```

getInt. This method returns the real part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `int`.

Use either of the following prototypes for the `getInt` method, where `index` can be of type `int` or `int[]`:

```
public int getInt(int index)
public int getInt(int[] index)
```

getShort. This method returns the real part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `short`.

Use either of the following prototypes for the getShort method, where index can be of type int or int[]:

```
public short getShort(int index)
public short getShort(int[] index)
```

getByte. This method returns the real part of the MWNumericArray element located at the specified one-based index or index array. The return value is given type byte.

Use either of the following prototypes for the getByte method, where index can be of type int or int[]:

```
public byte getByte(int index)
public byte getByte(int[] index)
```

Example — Getting a Short Value from a Numeric Array

The following examples use this array:

```
short[][] Adata = {{ 1,  2,  3,  4,  5,  6},
                   { 7,  8 , 9, 10, 11, 12},
                   {13, 14, 15, 16, 17, 18}};

MWNumericArray A = new MWNumericArray(Adata, MWClassID.INT16);
int[] index = {2, 4};
System.out.println("A(2,4) = " + A.getShort(index));
```

When run, the example displays this output:

```
A(2,4) = 10
```

Example — Using get and set on a Numeric Array

Given the same MWNumericArray used in the previous example, get and then modify the value of element (2, 3):

```
int[] idx = {2, 3};

System.out.println("A(2, 3) is " + A.get(idx).toString());
System.out.println("");
System.out.println("Setting A(2, 3) to a new value ...");
A.set(idx, 555);
System.out.println("");

System.out.println("A(2, 3) is now " + A.get(idx).toString());
```

When run, the example displays this output:

```
A(2, 3) is 9.0

Setting A(2, 3) to a new value ...

A(2, 3) is now 555.0
```

set. `MWNumericArray` inherits the following methods from the `MWArray` class.

```
set(int index, type element)
set(int[] index, type element)
```

`MWNumericArray` also overloads `set` for primitive `byte`, `short`, `int`, `long`, `float`, and `double` types.

toArray. `MWNumericArray` inherits this method from the `MWArray` class.

Methods to Get and Set the Imaginary Parts of an MWNumericArray

Use these methods to get and set imaginary values in an object of class `MWNumericArray`.

Method	Description
"getImag" on page 4-92	Returns the imaginary part at the specified index array in this array.
"getImagData" on page 4-93	Returns a one-dimensional array containing a copy of the imaginary data in the underlying MATLAB array.
"getImagDouble" on page 4-94	Returns the imaginary part at the specified offset as a `double` value.

Method	Description
"getImagFloat" on page 4-95	Returns the imaginary part at the specified offset as a `float` value.
"getImagLong" on page 4-95	Returns the imaginary part at the specified offset as a `long` value.
"getImagInt" on page 4-95	Returns the imaginary part at the specified offset as an `int` value.
"getImagShort" on page 4-95	Returns the imaginary part at the specified offset as a `short` value.
"getImagByte" on page 4-96	Returns the imaginary part at the specified offset as a `byte` value.
"setImag" on page 4-96	Replaces the imaginary part at the specified index array in this array with the specified double value.
"toImagArray" on page 4-97	Returns an array containing a copy of the imaginary data in the underlying MATLAB array. The returned array has the same dimensionality as the MATLAB array.

The following syntax applies to all the above methods except `getImagData`.

Calling Syntax. To get the element at a specific index, use one of the following:

```
public type getImagType(int index)
public type getImagType(int[] index)
```

To set the element at a specific index, use one of the following:

```
public void setImag(int index, type element)
public void setImag(int[] index, type element)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

In the case where index is of type int, the valid range for index is 1 <= index <= N, where N is the total number of elements in the array.

In the case where index is of type int[], each element of the index vector is an index along one dimension of the MWNumericArray object. The valid range for any index is 1 <= index[i] <= N[i], where N[i] is the size of the ith dimension.

Exceptions. These methods throw the following exception:

IndexOutOfBoundsException

The specified index parameter is invalid.

getImag. This method returns the imaginary part of the MWNumericArray element located at the specified one-based index or index array. The type of the return value is Object.

Use either of the following prototypes for the getImag method, where index can be of type int or int[]:

```
public Object getImag(int index)
public Object getImag(int[] index)
```

Example — Getting the Real and Imaginary Parts of an Array

Start by creating a two-dimensional array of complex values:

```
double[][] Rdata = {{ 2,  3,  4},
                    { 8 , 9, 10},
                    {14, 15, 16}};

double[][] Idata = {{ 6,  5, 14},
                    { 7 , 1, 23},
                    { 1,  1,  9}};

MWNumericArray A = new MWNumericArray(Rdata, Idata,
                        MWClassID.DOUBLE);

System.out.println("Complex matrix A =");
System.out.println(A.toString());
```

Here is the complex array that is displayed:

```
  2.0000 + 6.0000i   3.0000 + 5.0000i   4.0000 + 14.0000i
  8.0000 + 7.0000i   9.0000 + 1.0000i  10.0000 + 23.0000i
 14.0000 + 1.0000i  15.0000 + 1.0000i  16.0000 +  9.0000i
```

Now, use `get` and `getImag` to read the real and imaginary parts of the element at index (2, 3):

```
int[] index = {2, 3};
System.out.println("The real part of A(2,3) = " +
                    A.get(index));
System.out.println("The imaginary part of A(2,3) = " +
                    A.getImag(index));
```

When run, the example displays this output:

```
The real part of A(2,3) = 10.0
The imaginary part of A(2,3) = 23.0
```

getImagData. This method returns a one-dimensional `MWNumericArray` containing a copy of the imaginary data in the underlying MATLAB array.

The prototype for the `getImagData` method is as follows:

```
public Object getImagData()
```

`getImagData` returns the array of elements in column-wise order. The elements are converted according to default conversion rules.

Example — Getting Data from a Complex Array

Using the same array as in the example for “getImag” on page 4-92, get the entire contents of the complex array:

```
int[] index = {2, 3};
double[] x;
```

```
System.out.println("The real data in matrix A is:");
x = (double[]) A.getData();
for (int i = 0; i < x.length; i++)
   System.out.print(" " + x[i]);
System.out.println();

System.out.println();

System.out.println("The imaginary data in matrix A is:");
x = (double[]) A.getImagData();
for (int i = 0; i < x.length; i++)
   System.out.print(" " + x[i]);
System.out.println();
```

When run, the example displays this output:

```
The real data in matrix A is:
 2.0 8.0 14.0 3.0 9.0 15.0 4.0 10.0 16.0

The imaginary data in matrix A is:
 6.0 7.0 1.0 5.0 1.0 1.0 14.0 23.0 9.0
```

getImagDouble. This method returns the imaginary part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `double`.

Use either of the following prototypes for the `getImagDouble` method, where `index` can be of type `int` or `int[]`:

```
public double getImagDouble(int index)
public double getImagDouble(int[] index)
```

Example — Getting Complex Data of a Specific Type

Using the same array as in the example for "getImag" on page 4-92, get the real and imaginary parts of one element of the `MWNumericArray`:

```
int[] index = {2, 3};
System.out.println("The real part of A(2,3) = " +
                    A.getDouble(index));
```

```
System.out.println("The imaginary part of A(2,3) = " +
                    A.getImagDouble(index));
```

When run, the example displays this output:

```
The real part of A(2,3) = 10.0
The imaginary part of A(2,3) = 23.0
```

getImagFloat. This method returns the imaginary part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `float`.

Use either of the following prototypes for the `getImagFloat` method, where `index` can be of type `int` or `int[]`:

```
public float getImagFloat(int index)
public float getImagFloat(int[] index)
```

getImagLong. This method returns the imaginary part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `long`.

Use either of the following prototypes for the `getImagLong` method, where `index` can be of type `int` or `int[]`:

```
public long getImagLong(int index)
public long getImagLong(int[] index)
```

getImagInt. This method returns the imaginary part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `int`.

Use either of the following prototypes for the `getImagInt` method, where `index` can be of type `int` or `int[]`:

```
public int getImagInt(int index)
public int getImagInt(int[] index)
```

getImagShort. This method returns the imaginary part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `short`.

Use either of the following prototypes for the `getImagShort` method, where `index` can be of type `int` or `int[]`:

```
public short getImagShort(int index)
public short getImagShort(int[] index)
```

getImagByte. This method returns the imaginary part of the `MWNumericArray` element located at the specified one-based index or index array. The return value is given type `byte`.

Use either of the following prototypes for the `getImagByte` method, where `index` can be of type `int` or `int[]`:

```
public byte getImagByte(int index)
public byte getImagByte(int[] index)
```

setImag. This method replaces the imaginary part at the specified one-based index array in this array with the specified byte value.

Use either of the following prototypes for the `setImag` method, where `index` can be of type `int` or `int[]`:

```
public void setImag(int index, javatype element)
public void setImag(int[] index, javatype element)
```

The type *`javatype`* can be any of the following:

- `double`
- `float`
- `long`
- `int`
- `short`
- `byte`
- `Object`

Exceptions

These methods throw the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

toImagArray. This method returns an array containing a copy of the imaginary data in the underlying MATLAB array.

The prototype for the `toImagArray` method is

```
public Object toImagArray()
```

The array that is returned has the same dimensionality as the MATLAB array. The elements of this array are converted according to default conversion rules.

Input Parameters

None

Example — Getting Complex Data with toImagArray

Using the same array as in the example for “getImag” on page 4-92, get and display a copy of the imaginary part of that array:

```
double[][] x = (double[][]) A.toImagArray();
int[] dimA = A.getDimensions();

System.out.println("The imaginary part of matrix A is:");
for (int i = 0; i < dimA[0]; i++)
   {
   for (int j = 0; j < dimA[1]; j++)
      System.out.print(" " + x[i][j]);
   System.out.println();
   }
```

When run, the example displays this output:

```
The imaginary part of matrix A is:
 6.0 5.0 14.0
 7.0 1.0 23.0
 1.0 1.0 9.0
```

Methods to Copy, Convert, and Compare MWNumericArrays

Use these methods to copy, convert, and compare objects of class `MWNumericArray`.

Method	Description
"clone" on page 4-98	Creates and returns a deep copy of this array.
"compareTo" on page 4-99	Compares this array with the specified array for order.
"equals" on page 4-100	Indicates whether some other array is equal to this one.
"hashcode" on page 4-100	Returns a hash code value for the array.
"sharedCopy" on page 4-100	Creates and returns a shared copy of this array.
"toString" on page 4-100	Returns a string representation of the array.

clone. This method creates and returns a deep copy of this array. Because `clone` allocates a new array, any changes made to this new array are not reflected in the original.

The `clone` method of `MWNumericArray` overrides the `clone` method of class `MWArray`.

The prototype for the `clone` method is

```
public Object clone()
```

Input Parameters

None

Exceptions

The clone method throws the following exception:

`java.lang.CloneNotSupportedException`

The object's class does not implement the `Cloneable` interface.

Example — Cloning a Numeric Array Object

Create a 3-by-6 array of type `double`:

```
double[][] AData = {{ 1,  2,  3,  4,  5,  6},
                    { 7,  8 , 9, 10, 11, 12},
                    {13, 14, 15, 16, 17, 18}};

MWNumericArray A = new MWNumericArray(AData, MWClassID.DOUBLE);
```

Create a clone of the `MWNumericArray` object `A`:

```
Object C = A.clone();

System.out.println("Clone of matrix A is:");
System.out.println(C.toString());
```

When run, the example displays this output:

```
Clone of matrix A is:
   1     2     3     4     5     6
   7     8     9    10    11    12
  13    14    15    16    17    18
```

compareTo. `MWNumericArray` inherits this method from the `MWArray` class.

equals. `MWNumericArray` inherits this method from the `MWArray` class.

hashcode. `MWNumericArray` inherits this method from the `MWArray` class.

sharedCopy. This method creates and returns a shared copy of the `MWNumericArray` object. The shared copy points to the underlying original MATLAB array. Any changes made to the copy are reflected in the original.

The `sharedCopy` method of `MWNumericArray` overrides the `sharedCopy` method of class `MWArray`.

The prototype for the `sharedCopy` method is as follows:

```
public Object sharedCopy()
```

Input Parameters

None

Example — Making a Shared Copy of a Numeric Array Object

Create a shared copy of `MWArray` object `A`:

```
Object S = A.sharedCopy();

System.out.println("Shared copy of matrix A is:");
System.out.println(S.toString());
```

When run, the example displays this output:

```
Shared copy of matrix A is:
    1     2     3     4     5     6
    7     8     9    10    11    12
   13    14    15    16    17    18
```

toString. `MWNumericArray` inherits this method from the `MWArray` class.

Methods to Use on Sparse MWNumericArrays

Use these methods to return information on sparse arrays of type `MWNumericArray`. All are inherited from class `MWArray`.

Operations on sparse arrays of type `MWNumericArray` are currently supported only for the `double` type.

Method	Description
"newSparse" on page 4-77	Constructs a real sparse numeric matrix with the specified number of rows and columns and maximum nonzero elements, and initializes the array with the supplied data.
"isSparse" on page 4-65	Tests whether the array is sparse.
"columnIndex" on page 4-66	Returns an array containing the column index of each nonzero element in the underlying MATLAB array.
"rowIndex" on page 4-67	Returns an array containing the row index of each nonzero element in the underlying MATLAB array.
"maximumNonZeros" on page 4-67	Returns the allocated capacity of a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements()`.
"numberOfNonZeros" on page 4-68	Returns the number of nonzero elements in a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements()`.

`MWNumericArray` inherits all the above methods from the `MWArray` class.

Methods to Return Special Constant Values

Use these methods to return the values symbolized by `EPS`, `Inf`, and `NaN` in MATLAB.

Method	Description
"getEps" on page 4-102	Get the value represented by `EPS` (floating-point relative accuracy) in MATLAB.
"getInf" on page 4-102	Get the value represented by `INF` (infinity) in MATLAB.
"getNaN" on page 4-103	Get the value represented by `NaN` (Not a Number) in MATLAB.

getEps. This method returns the MATLAB concept of `EPS`, which stands for the floating-point relative accuracy.

The prototype for the `getEps` method is

```
public static double getEps()
```

Input Parameters

None

Exceptions

None

getInf. This method returns the MATLAB concept of `Inf`, which stands for infinity.

The prototype for the `getInf` method is

```
public static double getInf()
```

Input Parameters

None

Exceptions

None

getNaN. This method returns the MATLAB concept of `NaN`, which stands for "Not a Number".

The prototype for the `getNaN` method is

```
public static double getNaN()
```

Input Parameters

None

Exceptions

None

Methods to Convert Array Data to a Specific Type

Use these methods to return copies of MATLAB arrays of a specified primitive data type, such as `float` or `int`.

Method	Description

`toByteArray`
`toDoubleArray`
`toFloatArray`
`toIntArray`
`toLongArray`
`toShortArray`
`toImagArray`
`toImagByteArray`
`toImagDoubleArray`
`toImagFloatArray`
`toImagIntArray`
`toImagLongArray`
`toImagShortArray`
	These methods return an array of Java types matching the primitive type in the name of the method. The returned array is of the same dimension as the underlying MATLAB array.
	
	For example, if you call `toShortArray`, an array of type short is returned regardless of the data type in the underlying array. The data is converted from another primitive type, if necessary, and the array's original dimensions are preserved upon return.
	
	If conversion is performed, truncation or other loss of precision may occur. For example, if you call `toFloatArray` on an instance of `MWArray` containing data of the type double, the floating–point values are truncated from double–precision (double) to single–precision (float) numbers.
	
	These methods can also be useful in determining the types in an array when the dimensionality of a real or complex `MWArray` is known but the type is not.
	
	For more information on a specific method, enter the method name in the MATLAB Help browser to browse the Javadoc.
	
	For examples, see "Code Fragment: Using to*Type*Array Methods" on page 3-20.

Method	Description

`getByteData`
`getDoubleData`
`getFloatData`
`getIntData`
`getLongData`
`getShortData`
`getImagData`
`getImagByteData`
`getImagDoubleData`
`getImagFloatData`
`getImagIntData`
`getImagLongData`
`getImagShortData`
	These methods return a one– dimensional array of Java types matching the primitive type in the name of the method.
	
	For example, if you call `getShortData`, an array of type short is returned regardless of the data type in the underlying array. The data is converted from another primitive type, if necessary.
	
	If conversion is performed, truncation or other loss of precision may occur. For example, if you call `getShortData` on an instance of `MWArray` containing data of the type double, the floating–point values are truncated from double–precision (double) to single–precision (float) numbers.
	
	These methods can be helpful when you require your data in a one–dimensional format for performance reasons, for example.
	
	For more information on a specific method, enter the method name in the MATLAB Help browser **Search** field to browse the Javadoc.

Using MWLogicalArray

This section covers the following topics:

- "Constructing an MWLogicalArray" on page 4-106
- "Methods to Create and Destroy an MWLogicalArray" on page 4-107
- "Methods to Return Information About an MWLogicalArray" on page 4-112
- "Methods to Get and Set Data in an MWLogicalArray" on page 4-114
- "Methods to Copy, Convert, and Compare MWLogicalArrays" on page 4-118
- "Methods to Use on Sparse MWLogicalArrays" on page 4-121

Constructing an MWLogicalArray

You can construct two types of `MWLogicalArray` objects – an empty logical scalar or an initialized logical scalar or array.

Constructing an Empty Logical Scalar. To construct an empty scalar logical of type `MWClassID.LOGICAL`, use

```
MWLogicalArray()
```

Constructing an Initialized Logical Scalar or Array. Use this constructor syntax to create a `MWLogicalArray` scalar or array that represents the primitive Java type *`javatype`*:

```
MWLogicalArray(javatype array)
```

The value of `array` is set to `true` if the argument is nonzero, and `false` otherwise.

The type *`javatype`* can be any of the following:

- `double`
- `float`
- `long`
- `int`
- `short`
- `byte`
- `boolean`
- `Object`

Example — Constructing an Initialized Logical Array Object

```
boolean[][] Adata = {{true, false, false},
                     {false, true, false}};

MWLogicalArray A = new MWLogicalArray(Adata);
```

Methods to Create and Destroy an MWLogicalArray

In addition to the `MWLogicalArray` constructor, you can use the `newInstance` and `newSparse` methods to construct a logical array. These two methods offer better performance than using the class constructor. To destroy the arrays, use either `dispose` or `disposeArray`.

Method	Description
"newInstance" on page 4-107	Constructs a logical array with the specified dimensions.
"newSparse" on page 4-108	Constructs a sparse logical matrix from the supplied full matrix.
"dispose" on page 4-111	Frees the native MATLAB array contained by this array.
"disposeArray" on page 4-111	Frees all native MATLAB arrays contained in the input object.

newInstance. This method constructs a real or complex array, specifying the array dimensions, type, and complexity. This is a static method of the class and thus does not need to be invoked in reference to an instance of the class.

Note This method offers better performance than using the class constructor.

To construct a logical array with specified dimensions and all elements initialized to `false`, use the following:

```
public static MWLogicalArray newInstance(int[] dims)
```

To construct a logical array with specified dimensions and initialized to the supplied data, use the following:

```
public static MWLogicalArray newInstance(int[] dims,
    Object data)
```

Input Parameters

`dims`

Array of nonnegative dimension sizes

`data`

Data to initialize the array

Exceptions

The `newInstance` method throws the following exceptions:

`NegativeArraySizeException`

The specified `dims` parameter is negative.

`ArrayStoreException`

The specified data is nonnumeric or non-Boolean.

Example — Constructing a Logical Array Object with newInstance

Construct a 1-by-5 logical array using the `newInstance` method. Note that data in the Java array must be stored in a column-wise order so that it will be in row-wise order in the final `MWLogicalArray` object.

```
boolean[] Adata = { true, true, false, false, true};
int[] dims = {1, 5};

MWLogicalArray A = MWLogicalArray.newInstance(dims, Adata);
System.out.println("Array A: " + A.toString());
```

When run, the example displays this output:

```
Array A:      1     1     0     0     1
```

newSparse. This method constructs a sparse `MWLogicalArray` with the specified number of rows and columns and maximum nonzero elements, and initializes the array with the supplied data. This is a static method of the class and thus does not need to be invoked in reference to an instance of the class.

Supported Prototypes

Supported prototypes for `newSparse` are as follows. All input parameters shown here are described under Input Parameters on page 109. Any parameters not specified are given their default values.

To construct a sparse logical matrix with no nonzero elements, use the following:

```
public static MWLogicalArray newSparse(int rows, int cols,
    int nzmax)
```

To construct a sparse logical matrix from a supplied full matrix, use the following:

```
public static MWLogicalArray newSparse(Object data)
```

To specify what data is assigned to each element, use the following:

```
public static MWLogicalArray newSparse(int[] rowindex,
    int[] colindex, Object data)
```

To specify the number of rows and columns in the array, use the following:

```
public static MWLogicalArray newSparse(int[] rowindex,
    int[] colindex, Object data, int rows, int cols)
```

To specify the maximum number of nonzero elements in the array, use the following:

```
public static MWLogicalArray newSparse(int[] rowindex,
    int[] colindex, Object data, int rows, int cols,
    int nzmax)
```

Input Parameters

`data`

Data to initialize the array. See the list of valid data types below.

`rowIndex` and `colIndex`

Arrays of one-based row and column indices

Row and column index arrays are used to construct the sparse array such that the following holds true, with space allocated for `nzmax` nonzeros:

```
S(rowIndex(k), colIndex(k)) = data(k)
```

`rows` and `cols`

Number of rows and columns in the matrix

`nzmax`

Maximum number of nonzero elements

Valid types for the `data` parameter are as follows:

- `double[]`
- `float[]`
- `long[]`
- `int[]`
- `short[]`
- `byte[]`
- `boolean[]`
- One-dimensional arrays of any subclass of `java.lang.Number`
- One-dimensional arrays of `java.lang.Boolean`
- One-dimensional arrays of `java.lang.String`

rowIndex and colIndex Parameters

Exceptions

The `newSparse` method throws the following exceptions:

`NegativeArraySizeException`

Row or column size is negative.

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

`ArrayStoreException`

Incompatible array type or invalid array data

Example — Constructing a Sparse Logical Array Object

Create a sparse array of logical values using the `newSparse` method:

```
boolean[] Adata = {true, true, false, false, true};

int[] ri = {1, 1, 1, 1, 1};
int[] ci = {1, 2, 3, 4, 5};

MWLogicalArray A = MWLogicalArray.newSparse(ri, ci, Adata);

System.out.println(A.toString());
```

When run, the example displays this output:

```
   (1,1)        1
   (1,2)        1
   (1,5)        1
```

dispose. `MWLogicalArray` inherits this method from the `MWArray` class.

disposeArray. `MWLogicalArray` inherits this method from the `MWArray` class.

Methods to Return Information About an MWLogicalArray

Use these methods to return information about an object of class `MWLogicalArray`.

Method	Description
"classID" on page 4-112	Returns the MATLAB type of this array.
"getDimensions" on page 4-114	Returns an array containing the size of each dimension of this array.
"isEmpty" on page 4-114	Tests whether the array has no elements.
"numberOfDimensions" on page 4-114	Returns the number of dimensions of this array.
"numberOfElements" on page 4-114	Returns the total number of elements in this array.

classID. This method returns the MATLAB type of the `MWLogicalArray` object. The `classID` method of `MWLogicalArray` overrides the `classID` method of class `MWArray`.

The prototype for the `classID` method is

```
public MWClassID classID()
```

`classID` returns a field defined by the `MWClassID` class. For an `MWLogicalArray`, `classID` returns the value `MWClassID.LOGICAL`.

Input Parameters

None

Example — Getting the Class ID for a Logical Array Object

Return the class ID for `MWLogicalArray` object `Adata`:

```
boolean[][] Adata = {{true, false, false},
                     {false, true, false}};

MWLogicalArray A = new MWLogicalArray(Adata);

System.out.println("Class of A is " + A.classID());
```

When run, the example displays this output:

```
Class of A is logical
```

getDimensions. `MWLogicalArray` inherits this method from the `MWArray` class.

isEmpty. `MWLogicalArray` inherits this method from the `MWArray` class.

numberOfDimensions. `MWLogicalArray` inherits this method from the `MWArray` class.

numberOfElements. `MWLogicalArray` inherits this method from the `MWArray` class.

Methods to Get and Set Data in an MWLogicalArray

Use these methods to get and set values in an object of class `MWLogicalArray`.

Method	Description
"get" on page 4-114	Returns the element at the specified offset as an `Object`.
"getData" on page 4-114	Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.
"getBoolean" on page 4-115	Returns the `boolean` at the specified one-based offset.
"set" on page 4-116	Replaces the element at the specified one-based offset in this array with the specified element.
"toArray" on page 4-118	Returns an array containing a copy of the data in the underlying MATLAB array. The returned array has the same dimensionality as the MATLAB array.

get. `MWLogicalArray` inherits this method from the `MWArray` class.

getData. `MWLogicalArray` inherits this method from the `MWArray` class.

getBoolean. This method returns the element located at the specified one-based index of the `MWLogicalArray` object.

To get the element at a specific index, use one of the following:

```
public boolean getBoolean(int index)
public boolean getBoolean(int[] index)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`index`

Index of the requested element in the `MWLogicalArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWLogicalArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

Exceptions

The `getBoolean` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Getting a Boolean Value from a Logical Array

```
boolean[][] Adata = {{true, false, false},
                     {false, true, false}};

MWLogicalArray A = new MWLogicalArray(Adata);

int[] index = {2, 2};
System.out.println("A(2,2) is " + A.getBoolean(index));
```

When run, the example displays this output:

```
A(2,2) = true
```

set. This method returns the element located at the specified one-based index of the `MWLogicalArray` object.

To set the element at a specific index, use one of the following:

```
public void set(int index, boolean element)
public void set(int[] index, boolean element)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`element`

New element to replace at `index`

`index`

Index of the requested element in the `MWLogicalArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where index is of type int[], each element of the index vector is an index along one dimension of the MWLogicalArray object. The valid range for any index is 1 <= index[i] <= N[i], where N[i] is the size of the ith dimension.

Exceptions

The set method throws the following exception:

IndexOutOfBoundsException

The specified index parameter is invalid.

Example — Setting a Value in a Logical Array

Get and modify the value at A(2,3):

```
boolean[][] Adata = {{true, false, false},
                     {false, true, false}};

MWLogicalArray A = new MWLogicalArray(Adata);

int[] index = {2, 3};
Object d_out = A.get(index);
System.out.println("Array element A(2,3) is " +
                    d_out.toString() + "\n");

System.out.println("Setting A(2,3) to true\n");
A.set(index, true);

d_out = A.get(index);
System.out.println("Array element A(2,3) is " +
                    d_out.toString() + "\n");
```

When run, the example displays this output:

```
Array element A(2,3) is false
```

```
Setting A(2,3) to true

Array element A(2,3) is true
```

toArray. `MWLogicalArray` inherits this method from the `MWArray` class.

Methods to Copy, Convert, and Compare MWLogicalArrays

Use these methods to copy, convert, and compare objects of class `MWLogicalArray`.

Method	Description
"clone" on page 4-118	Creates and returns a deep copy of this array.
"compareTo" on page 4-119	Compares this array with the specified array for order.
"equals" on page 4-119	Indicates whether some other array is equal to this one.
"hashCode" on page 4-120	Returns a hash code value for the array.
"sharedCopy" on page 4-120	Creates and returns a shared copy of this array.
"toString" on page 4-120	Returns a string representation of the array.

clone. This method creates and returns a deep copy of this array. Because `clone` allocates a new array, any changes made to this new array are not reflected in the original.

The `clone` method of `MWLogicalArray` overrides the `clone` method of class `MWArray`.

The prototype for the `clone` method is

```
public Object clone()
```

Input Parameters

None

Exceptions

The `clone` method throws the following exception:

`java.lang.CloneNotSupportedException`

The object's class does not implement the `Cloneable` interface.

Example — Cloning a Logical Array Object

Create a clone of `MWLogicalArray` object `A`:

```
boolean[][] Adata = {{true, false, false},
                     {false, true, false}};

MWLogicalArray A = new MWLogicalArray(Adata);

Object C = A.clone();

System.out.println("Clone of logical matrix A is:");
System.out.println(C.toString());
```

When run, the example displays this output:

```
Clone of logical matrix A is:
     1     0     0
     0     1     0
```

compareTo. `MWLogicalArray` inherits this method from the `MWArray` class.

equals. `MWLogicalArray` inherits this method from the `MWArray` class.

hashCode. `MWLogicalArray` inherits this method from the `MWArray` class.

sharedCopy. This method creates and returns a shared copy of the `MWLogicalArray` object. The shared copy points to the underlying original MATLAB array. Any changes made to the copy are reflected in the original.

The `sharedCopy` method of `MWLogicalArray` overrides the `sharedCopy` method of class `MWArray`.

The prototype for the `sharedCopy` method is

```
public Object sharedCopy()
```

Input Parameters

None

Example — Making a Shared Copy of a Logical Array Object

Create a shared copy of `MWLogicalArray` object `A`:

```
boolean[][] Adata = {{true, false, false},
                     {false, true, false}};

MWLogicalArray A = new MWLogicalArray(Adata);

Object C = A.sharedCopy();

System.out.println("Shared copy of logical matrix A is:");
System.out.println(C.toString());
```

When run, the example displays this output:

```
Shared copy of logical matrix A is:
     1     0     0
     0     1     0
```

toString. `MWLogicalArray` inherits this method from the `MWArray` class.

Methods to Use on Sparse MWLogicalArrays

Use these methods to return information on sparse arrays of type `MWLogicalArray`. All are inherited from class `MWArray`.

Method	Description
"isSparse" on page 4-65	Tests whether the array is sparse.
"columnIndex" on page 4-66	Returns an array containing the column index of each nonzero element in the underlying MATLAB array.
"rowIndex" on page 4-67	Returns an array containing the row index of each nonzero element in the underlying MATLAB array.
"maximumNonZeros" on page 4-67	Returns the allocated capacity of a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements()`.
"numberOfNonZeros" on page 4-68	Returns the number of nonzero elements in a sparse array. If the underlying array is nonsparse, this method returns the same value as `numberOfElements()`.

`MWLogicalArray` inherits all the above methods from the `MWArray` class.

Using MWCharArray

This section covers the following topics:

- "Constructing an MWCharArray" on page 4-122
- "Methods to Create and Destroy an MWCharArray" on page 4-123
- "Methods to Return Information About an MWCharArray" on page 4-125
- "Methods to Get and Set Data in the MWCharArray" on page 4-126
- "Methods to Copy, Convert, and Compare MWCharArrays" on page 4-130

Constructing an MWCharArray

Use the tables in this section to construct an `MWCharArray` from a particular Java data type. See the examples in this section for more help.

Constructing an Empty Character Array. To construct an empty `MWCharArray`, use

```
MWCharArray()
```

To construct a `MWCharArray` object from a primitive Java `char` scalar, use the following prototype:

```
MWCharArray(char value)
```

To construct a `MWCharArray` object from a Java `Object`, use

```
MWCharArray(Object value)
```

Input Parameters

`value`

Value to initialize the array

Valid argument types for `value` are as follows:

- N-dimensional primitive `char` arrays
- `java.lang.String`
- N-dimensional arrays of `java.lang.String`
- `java.lang.Character`
- N-dimensional arrays of `java.lang.Character`

Example — Constructing an Initialized Character Array Object

Construct one `MWCharArray` object from a primitive character array:

```
char[] chArray1 = {'H', 'e', 'l', 'l', 'o'};
```

```
char[] chArray2 = {'W', 'o', 'r', 'l', 'd'};
MWCharArray A = new MWCharArray(chArray1);

System.out.println("The string in MWCharArray1 is \"" + A + "\"");
```

Construct a second `MWCharArray` from a `String` object:

```
String str = new String(chArray2);
MWCharArray A2 = new MWCharArray(str);

System.out.println("The string in MWCharArray2 is \"" +
                    A2 + "\"");
```

When run, the example displays this output:

```
The string in MWCharArray1 is "Hello"

The string in MWCharArray2 is "World"
```

Methods to Create and Destroy an MWCharArray

In addition to the `MWCharArray` constructor, you can use the `newInstance` method to construct a character array. This method offers better performance than using the class constructor. To destroy the array, use either `dispose` or `disposeArray`.

Method	Description
"newInstance" on page 4-123	Constructs a `char` array with the specified dimensions.
"dispose" on page 4-124	Frees the native MATLAB array contained by this array.
"disposeArray" on page 4-125	Frees all native MATLAB arrays contained in the input object.

newInstance. This method constructs a `char` array with the specified dimensions and initializes the array with the supplied data. The input array must be of type `char[]` or `java.lang.String`. The characters in the array are assumed to be stored in column-major order.

To construct a `MWCharArray` object with the specified dimensions, use

```
public static MWCharArray newInstance(int[] dims)
```

The elements of the array are all initialized to zero.

To construct a `MWCharArray` object with the specified dimensions and initialized with the supplied data, use

```
public static MWCharArray newInstance(int[] dims,
    Object data)
```

Input Parameters

`dims`

Array of dimension sizes. Each dimension size must be nonnegative.

`data`

Data to initialize the array

Example — Constructing a Character Array Object with newInstance

Create an `MWCharArray` object containing the text `Hello`:

```
int[] dims = {1, 5};
char[] chArray = {'H', 'e', 'l', 'l', 'o'};
String str = new String(chArray);

MWCharArray A =
   MWCharArray.newInstance(dims, str);

System.out.println("The array string is \"" + A + "\"");
```

When run, the example displays this output:

```
The array string is "Hello"
```

dispose. `MWCharArray` inherits this method from the `MWArray` class.

disposeArray. `MWCharArray` inherits this method from the `MWArray` class.

Methods to Return Information About an MWCharArray

Use these methods to return information about an object of class `MWCharArray`.

Method	Description
"classID" on page 4-125	Returns the MATLAB type of this array.
"getDimensions" on page 4-126	Returns an array containing the size of each dimension of this array.
"isEmpty" on page 4-126	Tests whether the array has no elements.
"numberOfDimensions" on page 4-126	Returns the number of dimensions of this array.
"numberOfElements" on page 4-126	Returns the total number of elements in this array.

classID. This method returns the MATLAB type of the `MWCharArray` object. The `classID` method of `MWCharArray` overrides the `classID` method of class `MWArray`.

The prototype for the `classID` method is

```
public MWClassID classID()
```

Input Parameters

None

Example — Getting the Class ID of a Character Array

Create an `MWCharArray` object and then display the class ID:

```
char[] chArray1 = {'H', 'e', 'l', 'l', 'o'};
MWCharArray A = new MWCharArray(chArray1);

System.out.println("The class of A is " + A.classID());
```

When run, the example displays this output:

```
The class of A is char
```

getDimensions. `MWCharArray` inherits this method from the `MWArray` class.

isEmpty. `MWCharArray` inherits this method from the `MWArray` class.

numberOfDimensions. `MWCharArray` inherits this method from the `MWArray` class.

numberOfElements. `MWCharArray` inherits this method from the `MWArray` class.

Methods to Get and Set Data in the MWCharArray

Use these methods to get and set values in an object of class `MWCharArray`.

Method	Description
"get" on page 4-126	Returns the element at the specified offset as an `Object`.
"getData" on page 4-126	Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.
"getChar" on page 4-127	Returns the character at the specified one-based offset.
"set" on page 4-128	Replaces the element at the specified one-based offset in this array with the specified element.
"toArray" on page 4-130	Returns an array containing a copy of the data in the underlying MATLAB array. The returned array has the same dimensionality as the MATLAB array.

get. `MWCharArray` inherits this method from the `MWArray` class.

getData. `MWCharArray` inherits this method from the `MWArray` class.

getChar. This method returns the character located at the specified one-based index of the `MWCharArray` object.

To get the element at a specific index, use one of

```
public char getChar(int index)
public char getChar(int[] index)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`index`

Index of the requested element in the `MWCharArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWCharArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

Exceptions

The `getChar` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Getting Character Array Data with getChar

Use `getChar` to display the string stored in `MWCharArray` object `A`:

```
char[] chArray = {'H', 'e', 'l', 'l', 'o'};
MWCharArray A = new MWCharArray(chArray);

for (int i = 1; i <= 5; i++)
   System.out.print(A.getChar(i));
```

When run, the example displays this output:

```
Hello
```

set. This method replaces the character located at the specified one-based offset in the `MWCharArray` object with the specified `char` value.

To set the element at a specific index, use one of

```
public void set(int index, char element);
public void set(int[] index, char element);
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`element`

New element to replace at `index`

`index`

Index of the requested element in the `MWCharArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where index is of type int[], each element of the index vector is an index along one dimension of the MWCharArray object. The valid range for any index is 1 <= index[i] <= N[i], where N[i] is the size of the ith dimension.

Exceptions

The set method throws the following exception:

IndexOutOfBoundsException

The specified index parameter is invalid.

Example — Setting Values in a Character Array

Display a phrase stored in MWCharArray object A, change one of the characters, and then display the modified phrase:

```
char[] chArray = {'G', 'a', 'r', 'y'};
MWCharArray A = new MWCharArray(chArray);

System.out.println("  I think " + A + " lives here." + "\n");

System.out.println("Changing the first character to M ...\n");
int[] index = {1, 1};
A.set(index, 'M');

System.out.println("  I think " + A + " lives here." + "\n");
```

When run, the example displays this output:

```
  I think Gary lives here.


Changing the first character to M ...

  I think Mary lives here.
```

toArray. `MWCharArray` inherits this method from the `MWArray` class.

Methods to Copy, Convert, and Compare MWCharArrays

Use these methods to copy, convert, and compare objects of class `MWCharArray`.

Method	Description
"clone" on page 4-130	Creates and returns a deep copy of this array.
"compareTo" on page 4-131	Compares this array with the specified array for order.
"equals" on page 4-131	Indicates whether some other array is equal to this one.
"hashCode" on page 4-131	Returns a hash code value for the array.
"sharedCopy" on page 4-131	Creates and returns a shared copy of this array.
"toString" on page 4-132	Returns a string representation of the array.

clone. This method creates and returns a deep copy of this array. Because `clone` allocates a new array, any changes made to this new array are not reflected in the original.

The `clone` method of `MWCharArray` overrides the `clone` method of class `MWArray`.

The prototype for the `clone` method is

```
public Object clone()
```

Input Parameters

None

Example — Cloning a Character Array Object

Create a clone of `MWCharArray` object `A`:

```
char[] chArray = {'H', 'e', 'l', 'l', 'o'};
MWCharArray A = new MWCharArray(chArray);

Object C = A.clone();

System.out.println("Clone of matrix A is:");
System.out.println(C.toString());
```

When run, the example displays this output:

```
Clone of matrix A is:
Hello
```

compareTo. `MWCharArray` inherits this method from the `MWArray` class.

equals. `MWCharArray` inherits this method from the `MWArray` class.

hashCode. `MWCharArray` inherits this method from the `MWArray` class.

sharedCopy. This method creates and returns a shared copy of the `MWCharArray` object. The shared copy points to the underlying original MATLAB array. Any changes made to the copy are reflected in the original.

The `sharedCopy` method of `MWCharArray` overrides the `sharedCopy` method of class `MWArray`.

The prototype for the `sharedCopy` method is

```
public Object sharedCopy();
```

Input Parameters

None

Example — Making a Shared Copy of a Character Array Object

Create a shared copy of MWCharArray object A:

```
char[] chArray = {'H', 'e', 'l', 'l', 'o'};
MWCharArray A = new MWCharArray(chArray);

Object S = A.sharedCopy();

System.out.print("Shared copy of matrix A is \"" +
                   S.toString() + "\"");
```

When run, the example displays this output:

```
Shared copy of matrix A is "Hello"
```

toString. MWCharArray inherits this method from the MWArray class.

Using MWStructArray

This section covers the following topics:

- "Constructing an MWStructArray" on page 4-132
- "Methods to Destroy an MWStructArray" on page 4-134
- "Methods to Return Information About an MWStructArray" on page 4-135
- "Methods to Get and Set Data in the MWStructArray" on page 4-138
- "Methods to Copy, Convert, and Compare MWStructArrays" on page 4-146

Constructing an MWStructArray

Use the tables in this section to construct an MWStructArray from a particular Java data type. See the examples at the end of this section for more help.

Constructing an Empty Structure Array. To construct an empty 0-by-0 MATLAB structure array, use

```
MWStructArray()
```

To construct an MWStructArray object with the specified dimensions and field names, use

```
MWStructArray(int[] dims, java.lang.String[] fieldnames)
```

To construct an MWStructArray object with the specified number of rows and columns, and field names, use

```
MWStructArray(int rows, int cols, java.lang.String[] fieldnames)
```

Input Parameters

`dims`

Array of dimension sizes. Each dimension size must be nonnegative.

`fieldnames`

Array of field names

`rows`

Number of rows in the array. This number must be nonnegative.

`cols`

Number of columns in the array. This number must be nonnegative.

Example — Constructing a Structure Array Object

This first example creates a 0-by-0 MWStructArray object:

```
MWStructArray S = new MWStructArray();
System.out.println("Structure array S: " + S);
```

When run, the example displays this output:

```
Structure array S: []
```

The second example creates a 1-by-2 `MWStructArray` object with fields `f1`, `f2`, and `f3`:

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};

MWStructArray S = new MWStructArray(sdims, sfields);

System.out.println("Structure array S: " + S);
```

When run, the example displays this output:

```
Structure array S: 1x2 struct array with fields:
    f1
    f2
    f3
```

Methods to Destroy an MWStructArray

To destroy the arrays, use either `dispose` or `disposeArray`.

Method	Description
"dispose" on page 4-134	Frees the native MATLAB array contained by this array.
"disposeArray" on page 4-135	Frees all native MATLAB arrays contained in the input object.

dispose. The `dispose` method of `MWStructArray` overrides the `dispose` method of class `MWArray`.

The prototype for the `dispose` method is

```
public void dispose()
```

Input Parameters

None

Example — Disposing of a Structure Array Object

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};

MWStructArray S = new MWStructArray(sdims, sfields);

System.out.println("Structure array S: " + S);
System.out.println("Now disposing of array S\n");
S.dispose();

System.out.println("Structure array S: " + S);
```

When run, the example displays this output:

```
Structure array S: 1x2 struct array with fields:
    f1
    f2
    f3

Now disposing of array S

Structure array S: []
```

disposeArray. `MWStructArray` inherits this method from the `MWArray` class.

Methods to Return Information About an MWStructArray

Use these methods to return information about an object of class `MWStructArray`.

Method	Description
"classID" on page 4-136	Returns the MATLAB type of this array.
"fieldNames" on page 4-136	Returns the field names in this array.
"getDimensions" on page 4-137	Returns an array containing the size of each dimension of this array.
"isEmpty" on page 4-137	Tests whether the array has no elements.

Method	Description
"numberOfDimensions" on page 4-137	Returns the number of dimensions of this array.
"numberOfElements" on page 4-137	Returns the total number of elements in this array.
"numberOfFields" on page 4-137	Returns the number of fields in this array.

classID. This method returns the MATLAB type of this array. The `classID` method of `MWStructArray` overrides the `classID` method of class `MWArray`.

The prototype for the `classID` method is

```
public MWClassID classID()
```

Input Parameters

None

Example — Getting the Class ID of a Structure Array

Create an `MWStructArray` object and display the class ID:

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};

MWStructArray S = new MWStructArray(sdims, sfields);

System.out.println("The class of S is " + S.classID());
```

When run, the example displays this output:

```
The class of S is struct
```

fieldNames. This method returns the field names in this array.

The prototype for the `fieldNames` method is

```
public java.lang.String[] fieldNames()
```

Input Parameters

None

Example — Getting the Field Names of a Structure Array

Create an `MWStructArray` object with three fields and display the field names:

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
MWStructArray S = new MWStructArray(sdims, sfields);

String[] str = S.fieldNames();

System.out.print("The structure has the fields: ");
  for (int i=0; i<S.numberOfFields(); i++)
     System.out.print(" " + str[i]);
```

When run, the example displays this output:

```
The structure has the fields:  f1 f2 f3
```

getDimensions. `MWStructArray` inherits this method from the `MWArray` class.

isEmpty. `MWStructArray` inherits this method from the `MWArray` class.

numberOfDimensions. `MWStructArray` inherits this method from the `MWArray` class.

numberOfElements. `MWStructArray` inherits this method from the `MWArray` class.

numberOfFields. This method returns the number of fields in this array.

The prototype for the `numberOfFields` method is

```
public int numberOfFields()
```

Input Parameters

None

Example — Getting the Number of Fields in a Structure Array

Create an `MWStructArray` object with three fields and display the number of fields:

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
MWStructArray S = new MWStructArray(sdims, sfields);

String[] str = S.fieldNames();

System.out.println("There are " + S.numberOfFields() +
                   " fields in this structure.");
```

When run, the example displays this output:

```
There are 3 fields in this structure.
```

Methods to Get and Set Data in the MWStructArray

Use these methods to get and set values in an object of class `MWStructArray`.

Method	Description
"get" on page 4-139	Returns the element at the specified offset as an `Object`.
"getData" on page 4-141	Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.

Method	Description
"getField" on page 4-142	Returns a shared copy of the element at the specified one-based offset and field name in this array as an `MWArray` instance.
"set" on page 4-143	Replaces the element at the specified one-based offset in this array with the specified element.
"toArray" on page 4-145	Returns an array containing a copy of the data in the underlying MATLAB array. The returned array has the same dimensionality as the MATLAB array.

get. This method returns the element at the specified one-based offset in this array. The returned element is converted to a Java array using default conversion rules.

To get the element at a specific index, use one of the following. Calling this method is equivalent to calling `getField(index).toArray()`.

```
public Object get(int index)
public Object get(int[] index)
```

To get the element at a specific index and structure field, use one of the following. Calling this method is equivalent to calling `getField(fieldname, index).toArray()`.

```
public Object get(String fieldname, int index)
public Object get(String fieldname, int[] index)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`fieldname`

Field name of the requested element

`index`

Index of the requested element in the `MWStructArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWStructArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

Exceptions

The `get` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Getting Structure Array Data with get

```
int[] cdims = {1, 3};
         MWStructArray C = new MWStructArray(cdims);
         Integer val = new Integer(15);
         int[] index2  = {1, 3};
         C.set(index2, val);
         Object x = C.get(index2);
         if (x instanceof int[][])
         {
          int[][] y = (int[][])x;
             System.out.println("B: Cell data C(1,3) is " + y[0][0]);
         }
```

When run, the example displays this output:

```
B: Cell data C(1,3) is 15
```

getData. This method returns a one-dimensional array containing a copy of the data in the underlying MATLAB array. The `getData` method of `MWStructArray` overrides the `getData` method of class `MWArray`.

The prototype for the `getData` method is

```
public Object getData()
```

`getData` returns a one-dimensional array of elements stored in column-wise order. Before converting, a new array is derived by transforming the struct array into a cell array such that an N-by-M-by-... struct array with P fields is transformed into a P-by-N-by-M-by-... cell array. Each element in the returned array is converted to a Java array when you call `MWArray.toArray()` on the corresponding cell.

Input Parameters

None

Example — Getting Structure Array Data with getData

Get the data stored in all fields and indices of `MWStructArray` object `S`:

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
MWStructArray S = new MWStructArray(sdims, sfields);

int count = S.numberOfElements() * S.numberOfFields();

// Initialize the structure.
Integer[] val = new Integer[6];
for (int i = 0; i < count; i++)
   val[i] = new Integer((i+1) * 15);

// Use getData to get data from the structure.
System.out.println("Data read from structure array S: \n");
MWArray[] x = (MWArray[]) S.getData();
for (int i = 0; i < x.length; i++)
```

```
    System.out.print(" " + x[i]);
```

When run, the example displays this output:

```
Data read from structure array S:

       15
       30
       45
       60
       75
       90
```

getField. This method returns a shared copy of the element at the specified one-based index array and field name in this array as an `MWArray` instance.

To get the element at a specific index, use one of

```
public MWArray getField(int index)
public MWArray getField(int[] index)
```

To get the element at a specific index and structure field, use one of

```
public MWArray getField(String fieldname, int index)
public MWArray getField(String fieldname, int[] index)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Dispose of the returned `MWArray` reference by calling `MWArray.dispose()`.

Input Parameters

`fieldname`

Field name of the requested element

`index`

Index of the requested element in the `MWStructArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWStructArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

Exceptions

The `get` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid

set. This method returns the element at the specified one-based offset in this array. The `set` method of `MWStructArray` overrides the `set` method of class `MWArray`.

To set the `element` at a specific `index`, use one of

```
public void set(int index, Object element)
public void set(int[] index, Object element)
```

To set the `element` at a specific `index` and structure field, use one of

```
public void set(String fieldname, int index, Object element)
public void set(String fieldname, int[] index, Object element)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`fieldname`

Field name of the requested element

`index`

Index of the requested element in the `MWStructArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWStructArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

`element`

New element to replace at `index`

If `element` is of type `MWArray`, the cell at `index` is set to a shared copy of the underlying MATLAB array. Otherwise, a new MATLAB array is created from `element` using default conversion rules and assigned to the cell at `index`.

Exceptions

The `set` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Setting Values in a Structure Array

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
```

```
MWStructArray S = new MWStructArray(sdims, sfields);

Integer[] val = new Integer[25];
for (int i = 0; i < 6; i++)
   val[i] = new Integer(i * 15);

for (int i = 0; i < 2; i++)
   for (int j = 0; j < sfields.length; j++)
      S.set(sfields[j], i+1, val[j + (i * 3)]);

// Use getData to get data from the structure.
System.out.println("Data read from structure array S: \n");
Object[] x = (Object[]) S.getData();
for (int i = 0; i < x.length; i++)
   System.out.print(" " + ((int[][]) x[i])[0][0]);
```

When run, the example displays this output:

```
Data read from structure array S:

 0 15 30 45 60 75
```

toArray. This method returns an array containing a copy of the data in the underlying MATLAB array.

The prototype for the `toArray` method is

```
public Object[] toArray()
```

`toArray` returns an array with the same dimensionality as the MATLAB array. Before converting, a new array is derived by transforming the struct array into a cell array such that an N-by-M-by-... struct array with P fields is transformed into a P-by-N-by-M-by-... cell array. Each element in the returned array is converted to a Java array when you call `MWArray.toArray()` on the corresponding cell.

Input Parameters

None

Example — Getting Structure Array Data with toArray

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
MWStructArray S = new MWStructArray(sdims, sfields);

Integer[] val = new Integer[25];
for (int i = 0; i < 6; i++)
   val[i] = new Integer(i * 15);

for (int i = 0; i < 2; i++)
   for (int j = 0; j < sfields.length; j++)
      S.set(sfields[j], i+1, val[j + (i * 3)]);

Object[][][] x = (Object[][][]) S.toArray();
System.out.println();

System.out.println("Data read from structure array S \n");
for (int j = 0; j < 2; j++)
   for (int i = 0; i < x.length; i++)
      System.out.print(" " + ((int[][]) x[i][0][j])[0][0]);
```

When run, the example displays this output:

```
Data read from structure array S

 0 15 30 45 60 75
```

Methods to Copy, Convert, and Compare MWStructArrays

Use these methods to copy, convert, and compare objects of class `MWStructArray`.

Method	Description
"clone" on page 4-147	Creates and returns a deep copy of this array.
"compareTo" on page 4-148	Compares this array with the specified array for order.

Method	Description
"equals" on page 4-148	Indicates whether some other array is equal to this one.
"hashCode" on page 4-148	Returns a hash code value for the array.
"sharedCopy" on page 4-148	Creates and returns a shared copy of this array.
"toString" on page 4-149	Returns a string representation of the array.

clone. This method creates and returns a deep copy of this array. Because `clone` allocates a new array, any changes made to this new array are not reflected in the original.

The `clone` method of `MWStructArray` overrides the `clone` method of class `MWArray`.

The prototype for the `clone` method is

```
public Object clone()
```

Input Parameters

None

Exceptions

The `clone` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Cloning a Structure Array Object

Create an `MWStructArray` object and then a clone of that object:

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
MWStructArray S = new MWStructArray(sdims, sfields);

Object C = S.clone();

System.out.println("Clone of structure S is:");
System.out.println(C.toString());
```

When run, the example displays this output:

```
Clone of structure S is:
1x2 struct array with fields:
    f1
    f2
    f3
```

compareTo. `MWStructArray` inherits this method from the `MWArray` class.

equals. `MWStructArray` inherits this method from the `MWArray` class.

hashCode. `MWStructArray` inherits this method from the `MWArray` class.

sharedCopy. This method creates and returns a shared copy of the `MWStructArray` object. The shared copy points to the underlying original MATLAB array. Any changes made to the copy are reflected in the original.

The `sharedCopy` method of `MWStructArray` overrides the `sharedCopy` method of class `MWArray`.

The prototype for the `sharedCopy` method is

```
public Object sharedCopy()
```

Input Parameters

None

Example — Making a Shared Copy of a Structure Array Object

Create an `MWStructArray` object and then a shared copy of that object:

```
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
MWStructArray S = new MWStructArray(sdims, sfields);

Object C = S.sharedCopy();

System.out.println("Shared copy of structure S is:");
System.out.println(C.toString());
```

When run, the example displays this output:

```
Shared copy of structure S is:
1x2 struct array with fields:
    f1
    f2
    f3
```

toString. `MWStructArray` inherits this method from the `MWArray` class.

Using MWCellArray

This section covers the following topics:

- "Constructing an MWCellArray" on page 4-150
- "Methods to Destroy an MWCellArray" on page 4-151
- "Methods to Return Information About an MWCellArray" on page 4-152
- "Methods to Get and Set Data in the MWCellArray" on page 4-154
- "Methods to Copy, Convert, and Compare MWCellArrays" on page 4-161

Constructing an MWCellArray

Use the tables in this section to construct an `MWCellArray` from a particular Java data type. See the examples at the end of this section for more help:

Constructing an Empty Cell Array. To construct an empty 0-by-0 MATLAB cell array, use

```
MWCellArray();
```

To construct an `MWCellArray` object with the specified dimensions, use

```
MWCellArray(int[] dims);
```

To construct an `MWCellArray` object with the specified number of rows and columns, use

```
MWCellArray(int rows, int cols);
```

Input Parameters

`dims`

Array of dimension sizes

`rows`

Number of rows

`cols`

Number of columns

Exceptions

The `MWCellArray` constructor throws the following exception:

`NegativeArraySizeException`

The specified `dims` parameter is negative.

Example — Constructing an Empty Cell Array Object

This first example creates an empty `MWCellArray` object:

```
MWCellArray C = new MWCellArray();
System.out.println("C = " + C.toString());
```

When run, the example displays this output:

```
C = []
```

Example — Constructing an Initialized Cell Array Object

The second example constructs and initializes a 2-by-3 `MWCellArray` object:

```
int[] cdims = {2, 3};
MWCellArray C = new MWCellArray(cdims);

Integer[] val = new Integer[6];
for (int i = 0; i < 6; i++)
   val[i] = new Integer(i * 15);

for (int i = 0; i < 2; i++)
   for (int j = 0; j < 3; j++)
      {
      int[] idx = {i+1, j+1};
      C.set(idx, val[j + (i * 3)]);
      }

System.out.println("C = " + C.toString());
```

When run, the example displays this output:

```
C =     [ 0]    [15]    [30]
        [45]    [60]    [75]
```

Methods to Destroy an MWCellArray

To destroy the arrays, use either `dispose` or `disposeArray`.

Method	Description
"dispose" on page 4-152	Frees the native MATLAB array contained by this array.
"disposeArray" on page 4-152	Frees all native MATLAB arrays contained in the input object.

dispose. This method frees the native MATLAB array contained by this array. The `dispose` method of `MWCellArray` overrides the `dispose` method of class `MWArray`.

The prototype for the `dispose` method is as follows:

```
public void dispose()
```

All `MWArray` references returned by `get(int)`, `toArray()`, or `getData()` are also disposed of.

Input Parameters

None

Example — Disposing of a Cell Array Object

Create a 2-by-3 `MWCellArray` object and then dispose of it.

```
int[] cdims = {2, 3};
MWCellArray C = new MWCellArray(cdims);

C.dispose();
```

disposeArray. `MWCellArray` inherits this method from the `MWArray` class.

Methods to Return Information About an MWCellArray

Use these methods to return information about an object of class `MWCellArray`.

Method	Description
“classID” on page 4-153	Returns the MATLAB type of this array.
“getDimensions” on page 4-153	Returns an array containing the size of each dimension of this array.
“isEmpty” on page 4-154	Tests whether the array has no elements.
“numberOfDimensions” on page 4-154	Returns the number of dimensions of this array.
“numberOfElements” on page 4-154	Returns the total number of elements in this array.

classID. This method returns the MATLAB type of this array. The `classID` method of `MWCellArray` overrides the `classID` method of class `MWArray`.

The prototype for the `classID` method is

```
public MWClassID classID()
```

Input Parameters

None

Example — Getting the Class ID of a Cell Array

Create an `MWCellArray` object and display its class:

```
int[] cdims = {2, 3};
MWCellArray C = new MWCellArray(cdims);

System.out.println("Class of C is " + C.classID());
```

When run, the example displays this output:

```
Class of C is cell
```

getDimensions. `MWCellArray` inherits this method from the `MWArray` class.

isEmpty. `MWCellArray` inherits this method from the `MWArray` class.

numberOfDimensions. `MWCellArray` inherits this method from the `MWArray` class.

numberOfElements. `MWCellArray` inherits this method from the `MWArray` class.

Methods to Get and Set Data in the MWCellArray

Use these methods to get and set values in an object of class `MWCellArray`.

Method	Description
"get" on page 4-154	Returns the element at the specified offset as an Object.
"getCell" on page 4-156	Returns a shared copy of the element at the specified one-based offset in this array as an `MWArray` instance.
"getData" on page 4-157	Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.
"set" on page 4-158	Replaces the element at the specified one-based offset in this array with the specified element.
"toArray" on page 4-160	Returns an array containing a copy of the data in the underlying MATLAB array. The returned array has the same dimensionality as the MATLAB array.

get. This method returns the element at the specified one-based offset in this array. The returned element is converted to a Java array using default conversion rules. Calling this method is equivalent to calling `getCell(index).toArray()`.

The `get` method of `MWCellArray` overrides the `get` method of class `MWArray`.

To get the element at a specific index, use one of the following:

```
public Object get(int index)
public Object get(int[] index)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`index`

Index of the requested element in the `MWCellArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWCellArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `ith` dimension.

Exceptions

The `get` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Getting Data from a Cell Array with get

```
int[] cdims = {1, 3};
        MWCellArray C = new MWCellArray(cdims);
        Integer val = new Integer(15);
        int[] index2  = {1, 3};
        C.set(index2, val);
        Object x = C.get(index2);
        if (x instanceof int[][])
        {
         int[][] y = (int[][])x;
            System.out.println("B: Cell data C(1,3) is " + y[0][0]);
        }
```

When run, the example displays this output:

```
B: Cell data C(1,3) is 15
```

getCell. This method returns a shared copy of the element at the specified one-based offset in this array as an `MWArray` instance.

To get the element at a specific index, use one of the following:

```
public MWArray getCell(int index)
public MWArray getCell(int[] index)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

`getCell` returns an `MWArray` instance representing the requested cell. When you are done using this instance, call `MWArray.dispose()` to dispose of it.

Input Parameters

`index`

Index of the requested element in the `MWCellArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWCellArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

Exceptions

The `getCell` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

getData. This method returns a one-dimensional array containing a copy of the data in the underlying MATLAB array. The `getData` method of `MWCellArray` overrides the `getData` method of class `MWArray`.

The prototype for the `getData` method is as follows:

```
public Object getData()
```

`getData` returns a one-dimensional array of elements stored in column-wise order. Each element in the returned array is converted to a Java array when you call `MWArray.toArray()` on the corresponding cell.

Input Parameters

None

Example — Getting Cell Array Data with getData

Use `getData` to read data from `MWCellArray` object `C`:

```
int[] cdims = {1, 3};
MWCellArray C = new MWCellArray(cdims);

Integer[] val = new Integer[3];
for (int i = 0; i < 3; i++)
   val[i] = new Integer(i * 15);

for (int i = 1; i <= 3; i++)
   C.set(i, val[i-1]);
```

```
System.out.println("Data read from cell array C: \n");
MWArray[] x = (MWArray[]) C.getData();

for (int i = 0; i < x.length; i++)
   System.out.println(x[i]);

System.out.println();
```

When run, the example displays this output:

```
Data read from cell array C:
     0
     0
     0
```

set. This method replaces the element at the specified one-based offset in this array with the specified element. The `set` method of `MWCellArray` overrides the `set` method of class `MWArray`.

To get the element at a specific index, use one of the following:

```
public void set(int index, Object element)
public void set(int[] index, Object element)
```

Use the first syntax (`int index`) to return the element at the specified one-based offset in the array, accessing elements in column-wise order. Use the second syntax (`int[] index`) to return the element at the specified array of one-based indices. The first syntax offers better performance than the second.

Input Parameters

`element`

New element to replace at index

If `element` is of type `MWArray`, the cell at `index` is set to a shared copy of the underlying MATLAB array. Otherwise, a new MATLAB array is created from `element` using default conversion rules and assigned to the cell at `index`.

`index`

Index of the requested element in the `MWCellArray`

In the case where `index` is of type `int`, the valid range for `index` is `1 <= index <= N`, where `N` is the total number of elements in the array.

In the case where `index` is of type `int[]`, each element of the `index` vector is an index along one dimension of the `MWCellArray` object. The valid range for any index is `1 <= index[i] <= N[i]`, where `N[i]` is the size of the `i`th dimension.

Exceptions

The `set` method throws the following exception:

`IndexOutOfBoundsException`

The specified `index` parameter is invalid.

Example — Setting Values in a Cell Array

Set the value of the `MWCellArray` object `C` at index `(1,3)`:

```
int[] cdims = {1, 3};
MWCellArray C = new MWCellArray(cdims);

Integer val = new Integer(15);
int[] index = {1, 3};

C.set(index, val);

Object x = C.get(index);
System.out.println("Cell data C(1,3) is " + x.toString());
```

When run, the example displays this output:

```
Cell data C(1,3) is     15
```

toArray. This method returns an array containing a copy of the data in the underlying MATLAB array.

The prototype for the `toArray` method is as follows:

```
public Object[] toArray()
```

`toArray` returns an array with the same dimensionality as the MATLAB array. Each element in the returned array is converted to a Java array when you call `MWArray.toArray()` on the corresponding cell.

Input Parameters

None

Example — Getting Cell Array Data with toArray

```
int[] cdims = {1, 3};
MWCellArray C = new MWCellArray(cdims);

System.out.println("Data read from cell array C \n");
Object x = (Object) C.toArray();
System.out.println();

for (int i = 0; i < x[0].length; i++)
   System.out.println(x[0][i]);
```

When run, the example displays this output:

```
Data read from cell array C
     []
     []
     []
```

Methods to Copy, Convert, and Compare MWCellArrays

Use these methods to copy, convert, and compare objects of class `MWCellArray`.

Method	Description
"clone" on page 4-161	Creates and returns a deep copy of this array.
"compareTo" on page 4-162	Compares this array with the specified array for order.
"equals" on page 4-162	Indicates whether some other array is equal to this one.
"hashCode" on page 4-162	Returns a hash code value for the array.
"sharedCopy" on page 4-162	Creates and returns a shared copy of this array.
"toString" on page 4-163	Returns a string representation of the array.

clone. This method creates and returns a deep copy of this array. Because `clone` allocates a new array, any changes made to this new array are not reflected in the original.

The `clone` method of `MWCellArray` overrides the `clone` method of class `MWArray`.

The prototype for the `clone` method is as follows:

```
public Object clone()
```

Input Parameters

None

Exceptions

The clone method throws the following exception:

IndexOutOfBoundsException

The specified index parameter is invalid.

Example — Cloning a Cell Array Object

Create an MWCellArray object and then a clone of that object:

```
int[] cdims = {1, 3};
MWCellArray C = new MWCellArray(cdims);

Object X = C.clone();

System.out.println("Clone of cell array C is:");
System.out.println(X.toString());
```

When run, the example displays this output:

```
Clone of cell array C is:
     []      []      []
```

compareTo. MWCellArray inherits this method from the MWArray class.

equals. MWCellArray inherits this method from the MWArray class.

hashCode. MWCellArray inherits this method from the MWArray class.

sharedCopy. This method creates and returns a shared copy of the MWCellArray object. The shared copy points to the underlying original MATLAB array. Any changes made to the copy are reflected in the original.

The sharedCopy method of MWCellArray overrides the sharedCopy method of class MWArray.

The prototype for the sharedCopy method is

```
public Object sharedCopy()
```

Input Parameters

None

Example — Making a Shared Copy of a Cell Array Object

Create an `MWCellArray` object and then a shared copy of that object:

```
int[] cdims = {1, 3};
MWCellArray C = new MWCellArray(cdims);

Object X = C.sharedCopy();

System.out.println("Shared copy of cell array C is:");
System.out.println(X.toString());
```

When run, the example displays this output:

```
Shared copy of cell array C is:
     []      []      []
```

toString. `MWCellArray` inherits this method from the `MWArray` class.

Using MWClassID

The `MWClassID` class enumerates all MATLAB array types. This class contains no public constructors. A set of public static `MWClassID` instances is provided, one for each MATLAB array type.

`MWClassID` extends class `java.lang.Object`.

`MWClassID` implements interface `java.io.Serializable`.

Fields of MWClassID

CELL. `CELL` represents MATLAB array type `cell`.

CHAR. `CHAR` represents MATLAB array type `char`.

DOUBLE. `DOUBLE` represents MATLAB array type `double`.

FUNCTION. `FUNCTION` represents MATLAB array type `function`.

Note MATLAB function arrays are not supported in the current release.

INT8. `INT8` represents MATLAB array type `int8`.

INT16. `INT16` represents MATLAB array type `int16`.

INT32. `INT32` represents MATLAB array type `int32`.

INT64. `INT64` represents MATLAB array type `int64`.

LOGICAL. `LOGICAL` represents MATLAB array type `logical`.

OBJECT. `OBJECT` represents MATLAB array type `object`.

Note MATLAB object arrays are not supported in the current release.

OPAQUE. `OPAQUE` represents MATLAB array type `opaque`.

SINGLE. `SINGLE` represents MATLAB array type `single`.

STRUCT. `STRUCT` represents MATLAB array type `struct`.

UINT8. `UINT8` represents MATLAB array type `uint8`.

UINT16. `UINT16` represents MATLAB array type `uint16`.

UINT32. `UINT32` represents MATLAB array type `uint32`.

UINT64. `UINT64` represents MATLAB array type `uint64`.

UNKNOWN. UNKNOWN represents MATLAB empty array type.

Example — Specifying an MWClassID Value

Construct a scalar numeric array of type MWClassID.INT16:

```
double AReal = 24;

MWNumericArray A = new MWNumericArray(AReal, MWClassID.INT16);
System.out.println("Array A of type " + A.classID() + " = \n" + A);
```

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

```
Array A of type int16 =
    24
```

Methods of MWClassID

equals. This method indicates whether some other MWClassID is equal to this one. The equals method of MWClassID overrides the equals method of class java.lang.Object.

The prototype for equals is as follows:

```
public final boolean equals(java.lang.Object obj)
```

getSize. This method returns the size in bytes of an array element of this type.

The prototype for getSize is as follows:

```
public final int getSize()
```

hashCode. This method returns a hash code value for the type. The hashCode method of MWClassID overrides the hashCode method of class java.lang.Object.

The prototype for hashCode is as follows:

```
public final int hashCode()
```

isNumeric. This method tests if this type is numeric.

The prototype for `isNumeric` is as follows:

```
public boolean isNumeric()
```

toString. This method returns a string representation of the property. The `toString` method of `MWClassID` overrides the `toString` method of class `java.lang.Object`.

The prototype for `toString` is as follows:

```
public final java.lang.String toString()
```

Using MWComplexity

The `MWComplexity` class enumerates the MATLAB real/complex array property. This class contains no public constructors. A set of public static `MWComplexity` instances is provided, one to represent `real` and one for `complex`.

`MWComplexity` extends class `java.lang.Object`.

`MWComplexity` implements interface `java.io.Serializable`.

Fields of MWComplexity

REAL. `REAL` represents a real numeric value. The prototype for `REAL` is as follows:

```
public static final MWComplexity REAL
```

COMPLEX. `COMPLEX` represents a complex numeric value containing both real and imaginary parts. The prototype for `COMPLEX` is as follows:

```
public static final MWComplexity COMPLEX
```

Example – Determining the Complexity of an Array

Determine whether matrix `A` is real or complex. The `complexity` method of `MWNumericArray` returns an enumeration of type `MWComplexity`.

```
double AReal = 24;
double AImag = 5;

MWNumericArray A = new MWNumericArray(AReal, AImag);
System.out.println("A is a " + A.complexity() + " matrix");
```

When run, the example displays this output:

```
A is a complex matrix
```

Methods of MWComplexity

toString. This method returns a string representation of the property. The `toString` method of `MWComplexity` overrides the `toString` method of class `java.lang.Object`.

The prototype for the `toString` method is as follows:

```
public java.lang.String toString()
```

Sample Java Applications

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

Plot Example

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

- Use MATLAB Builder for Java to convert a MATLAB function (`drawplot`) to a method of a Java class (`plotter`) and wrap the class in a Java component (`plotdemo`).
- Access the component in a Java application (`createplot.java`) by instantiating the `plotter` class and using the `MWArray` class library to handle data conversion.

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

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

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

Plot Example: Step-by-Step Procedure

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

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

   ```
   matlabroot\toolbox\javabuilder\Examples\PlotExample
   ```

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

2 If you have not already done so, set the environment variables that are required on a development machine. See “Settings for Environment Variables (Development Machine)” on page 7-2.

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

   The following code defines the `drawplot` function:

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

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

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

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

```
deploytool
```

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

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

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

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

9 Select **Generate Verbose Output**.

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

11 Save the project.

12 Build the component.

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

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

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

The program listing is shown here.

createplot.java

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

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

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

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

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

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

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

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

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

The program does the following:

- Creates two arrays of double values, using `MWNumericArray` to represent the data needed to plot the equation.
- Instantiates the `plotter` class as `thePlot` object, as shown:

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

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

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

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

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

a. On Windows, execute the following command:

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

b. On UNIX, execute this command:

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

15 Run the application.

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

On Windows, type

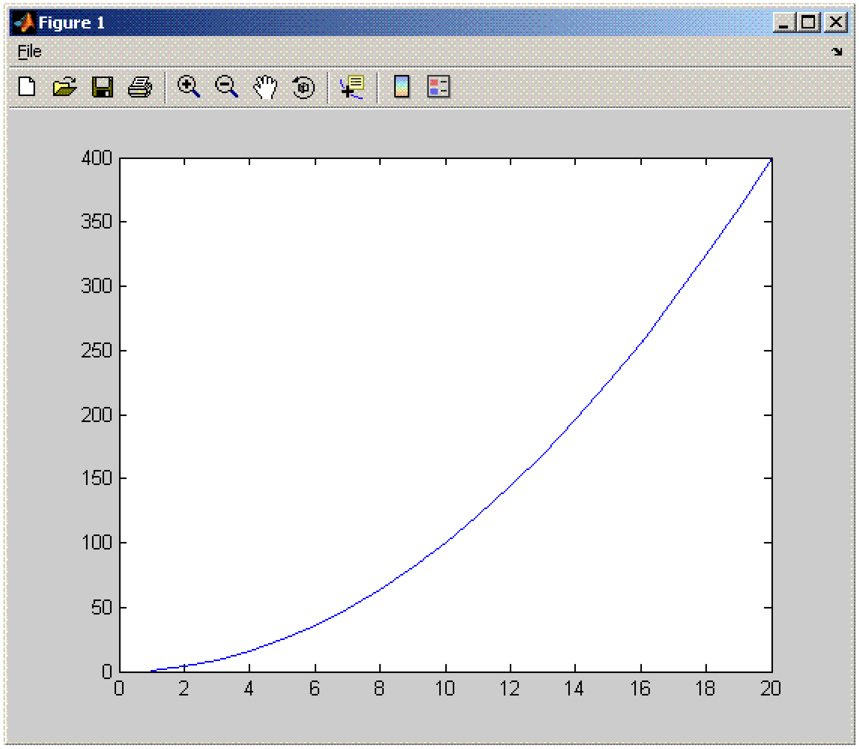
```
java -classpath
  .;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
  .\distrib\plotdemo.jar
  createplot
```

On UNIX, type

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

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

The `createplot` program should display the output:



Spectral Analysis Example

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

- How to use MATLAB Builder for Java to create a component (`spectralanalysis`) containing a class that has a private method that is automatically encapsulated.
- How to access the component in a Java application (`powerspect.java`), including use of the `MWArray` class hierarchy to represent data.

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

- How to build and run the application

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

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

computefft.m

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

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

plotfft.m

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

Spectral Analysis Example: Step-by-Step Procedure

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

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

      matlabroot`\toolbox\javabuilder\Examples\SpectraExample`

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

2. If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 7-2.

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

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

4. While in MATLAB, issue the following command to open the Deployment Tool dialog box:

   ```
   deploytool
   ```

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

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

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

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

9. Select **Generate Verbose Output**.

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

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

---

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

12. Build the component.

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

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

The program listing is shown here.

powerspect.java

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

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

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

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

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

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

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

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

The program does the following:

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

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

- Instantiates a `fourier` object
- Calls the `plotfft` method, which calls `computeftt` and plots the data
- Uses a `try/catch` block to handle exceptions
- Frees native resources using `MWArray` methods

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

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

b. On Windows, execute the following command:

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

c. On UNIX, execute the following command:

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

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

---

15 Run the application.

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

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

- On UNIX, execute the `powerspect` class file as follows:

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

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

The `powerspect` program should display the output:

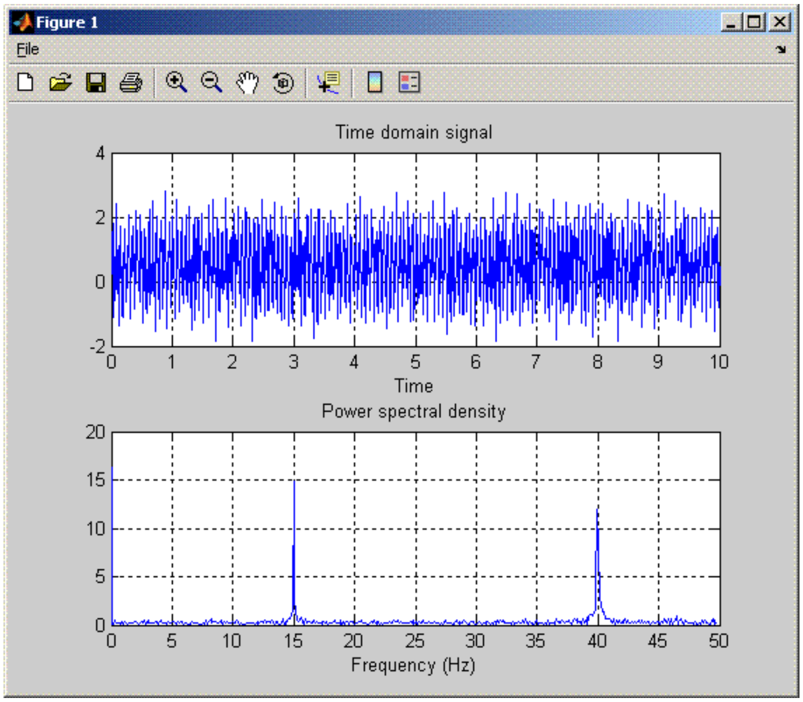


Figure 1
File
Time domain signal
4
2
0
-2
0
1
2
3
4
5
6
7
8
9
10
Time
Power spectral density
20
15
10
5
0
0
5
10
15
20
25
30
35
40
45
50
Frequency (Hz)

Matrix Math Example

In this section...

Example Overview

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

- How to assign more than one MATLAB function to a component class.
- How to manually handle native memory management.
- How to access the component in a Java application (`getfactor.java`) by instantiating `Factor` and using the `MWArray` class library to handle data conversion.

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

- How to build and run the `MatrixMathDemoApp` application

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

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

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

MATLAB Functions to Be Encapsulated

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

cholesky.m

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

%   Copyright 2001-2007 The MathWorks, Inc.

L = chol(A);
```

ludecomp.m

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

%   Copyright 2001-2007 The MathWorks, Inc.

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

qrdecomp.m

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

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

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

Step-by-Step Procedure

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

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

   ```
   matlabroot\toolbox\javabuilder\Examples\MatrixMathExample
   ```

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

2 If you have not already done so, set the environment variables that are required on a development machine. See "Settings for Environment Variables (Development Machine)" on page 7-2.

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

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

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

   ```
   deploytool
   ```

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

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

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

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

9 Select **Generate Verbose Output**.

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

11 Save the project.

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

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

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

The program listing is shown here.

getfactor.java

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

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

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

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

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

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

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

   if (args.length > 1 && args[1].equals("sparse"))
      a = MWNumericArray.newSparse(dims[0], dims[1],n+2*(n-1), MWClassID.DOUBLE, MWComplexity.REAL);
   else
      a = MWNumericArray.newInstance(dims,MWClassID.DOUBLE, MWComplexity.REAL);

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

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

```
                  a.set(index, -1.0);
            }
         }

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

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

         /* Compute cholesky factorization and print results. */
         result = theFactor.cholesky(1, a);
         System.out.println("Cholesky factorization:");
         System.out.println(result[0]);
         MWArray.disposeArray(result);

         /* Compute LU factorization and print results. */
         result = theFactor.ludecomp(2, a);
         System.out.println("LU factorization:");
         System.out.println("L matrix:");
         System.out.println(result[0]);
         System.out.println("U matrix:");
         System.out.println(result[1]);
         MWArray.disposeArray(result);

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

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

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

The statement:

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

creates an instance of the class `factor`.

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

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

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

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

- On Windows, execute the following command:

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

- On UNIX, execute the following command:

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

15 Run the application.

Run `getfactor` using a nonsparse matrix

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

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

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

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

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

Output for the Matrix Math Example

```
Original matrix:
     2    -1     0     0
    -1     2    -1     0
     0    -1     2    -1
     0     0    -1     2
Cholesky factorization:
    1.4142   -0.7071         0         0
```

```
         0    1.2247   -0.8165         0
         0         0    1.1547   -0.8660
         0         0         0    1.1180


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


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


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


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

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

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

Output for a Sparse Matrix

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


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


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


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


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


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

Understanding the getfactor Program

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

The main method has three parts:

- The first part sets up the input matrix, creates a new factor object, and calls the `cholesky`, `ludecomp`, and `qrdecomp` methods. This part is executed inside of a `try` block. This is done so that if an exception occurs during execution, the corresponding `catch` block will be executed.
- The second part is the `catch` block. The code prints a message to standard output to let the user know about the error that has occurred.
- The third part is a `finally` block to manually clean up native resources before exiting.

Phonebook Example

In this section...

The makephone Function

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

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

Phonebook Example: Step-by-Step Procedure

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

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

```
matlabroot\toolbox\javabuilder\Examples\PhoneExample
```

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

2 If you have not already done so, set the environment variables that are required on a development machine. See “Settings for Environment Variables (Development Machine)” on page 7-2.

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

The following code defines the `makephone` function:

```
function book = makephone(friends)
%MAKEPHONE Add a structure to a phonebook structure
%   BOOK = MAKEPHONE(FRIENDS) adds a field to its input structure.
%   The new field EXTERNAL is based on the PHONE field of the original.
%   This file is used as an example for MATLAB
%   Builder for Java.

%   Copyright 2006-2007 The MathWorks, Inc.

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

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

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

```
deploytool
```

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

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

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

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

9 Select **Generate Verbose Output**.

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

11 Save the project.

12 Build the component.

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

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

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

The program listing is shown here.

getphone.java

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

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

import phonebookdemo.*;

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

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

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

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

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

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

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

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

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

```
                thePhonebook.dispose();
            }
        }

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

The program does the following:

- Creates a structure array, using MWStructArray to represent the example phonebook data.
- Instantiates the plotter class as `thePhonebook` object, as shown: `thePhonebook = new phonebook();`
- Calls the `makephone` method to create a modified copy of the structure by adding an additional field, as shown: `result = thePhonebook.makephone(1, friends);`
- Utilizes a try-catch block to catch and handle any exceptions.

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

a. On Windows, execute the following command:

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

b. On UNIX, execute this command:

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

15 Run the application.

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

On Windows, type

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

On UNIX, type

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

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

The `getphone` program should display the output:

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

Entire structure:
Number of Elements: 4
Dimensions: 2-by-2
Number of Fields: 3
Standard MATLAB view:
2x2 struct array with fields:
    name
    phone
    external
Walking structure:
Element 1
```

```
   name: Jordan Robert
   phone: 3386
   external: (508) 555-3386
Element 2
   name: Mary Smith
   phone: 3912
   external: (508) 555-3912
Element 3
   name: Stacy Flora
   phone: 3238
   external: (508) 555-3238
Element 4
   name: Harry Alpert
   phone: 3077
   external: (508) 555-3077
```

Buffered Image Creation Example

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

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

```
function w = getSurfsFigure
 f = figure;
 set(f,'Visible', 'off');

 f = surf(peaks);
 w = hardcopy(gcf,'-dOpenGL','-r0');
end
```

Note There is minimal error handling in this example. Integrate this code with whatever logging is currently in place for your Java layer.

Note Be aware that `hardcopy` is currently an undocumented function and subject to change.

---

1. Create a Java object from the function above by doing the following:
   a. Start the Deployment Tool by entering `deploytool` from the MATLAB Command Prompt.
   b. Select **File > New > Deployment Project** from the MATLAB interface.
   c. Select a **MATLAB Builder for Java Project** and **Java Package**, then give the project a name and location. Click **OK**.
   d. Click **Settings** in the Deployment Tool and enter the package name as `com.mathworks.deploy.peaks`. Click **OK**.
   e. In the Deployment Tool, change the ***`project_name`*****class** name to `Peaks` by right-clicking on the class folder and selecting **Rename**.

f Add the function `getSurfsFigure.m` to the Peaks class by dragging the function from the MATLAB Current Directory browser to the **Peaks** class folder in the Deployment Tool.

g Click the **Build** icon in the Deployment Tool toolbar and build your Java object. Be sure to select the **Include MCR** build option.

2 From the output **distrib** directory of your build, copy `peaks.jar` to the directory where you are building your application.

3 Create the `SurfPeaks.java` code:

```
//imported classes from JRE
import java.awt.Image;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.ImageIcon;

//imported classes from javabuilder.jar
import com.mathworks.toolbox.javabuilder.Images;
import com.mathworks.toolbox.javabuilder.MWArray;
import com.mathworks.toolbox.javabuilder.MWNumericArray;
import com.mathworks.toolbox.javabuilder.MWException;

//Import for the Deployment Project peaks.jar
import com.mathworks.deploy.peaks.Peaks;

//The Goal of this class is to show how you would deal with
//  static images coming from a java deployment of a matlab figure.
//This can be run as a stand alone java application.
//
//The matlab function being deployed is surf(peaks).
//The hardcopy function is used to
//  output the figure window as an array.
//M Code:
//*********************************************
//    function w = getSurfsFigure
//        f = figure;
//        surf(peaks);
//        w = hardcopy(gcf,'-dOpenGL','-r0');
//        close(f);
//    end
//********************************************
//
//For this example you must have the deployment project
//  jar and the javabuilder.jar on your classpath.
//
//Note:
//For this example there is minimal error handling.
//Typically you would want to integrate this with whatever
//  logging is currently in place for your java layer.
public class SurfPeaks
{
    //This initializes and executes the JFrame.
    public static void main(String args[])
    {
        ImageIcon icon = new ImageIcon(getSurfImage());
        JLabel label =  new JLabel(icon);
        JFrame frame = new JFrame();
        frame.setSize(icon.getIconWidth(),icon.getIconHeight());
        frame.setContentPane(label);
        frame.setVisible(true);
    }

    //This method is basically our "business logic" method.
    //It is responsible for instantiating our Matlab deployment,
    //passing in any needed inputs, and dealing with any outputs.
    //In this example we have no inputs, and the only output is the
    //figure in hardcopy format (three dimensional MWNumericArray)
    private static Image getSurfImage()
    {
        try
        {
            //Our deployment uses native resources and
            //should be disposed of as soon as possible.
            Peaks matlabModel = new Peaks();
            try
            {
                //If we had any inputs to our method
                //they would be passed in here.
                Object[] results = matlabModel.getSurfsFigure(1);
```

```
                    //This array uses native resources and
                    //should be disposed of as soon as possible.
                    MWArray mwArray = (MWArray)results[0];
                    try
                    {
                        //Since we want this method to return only
                        //     non matlab data
                        //  we convert the matlab figure to a
                        //     buffered image and return it.
                        return Images.renderArrayData
                             ((MWNumericArray)mwArray);
                    }
                    finally
                    {
                        MWArray.disposeArray(mwArray);
                    }
                }
                finally
                {
                    matlabModel.dispose();
                }
            }
            catch(MWException mwe)
            {
                mwe.printStackTrace();
                return null;
            }
        }
    }
```

4 Compile the program using `javac` and the following command:

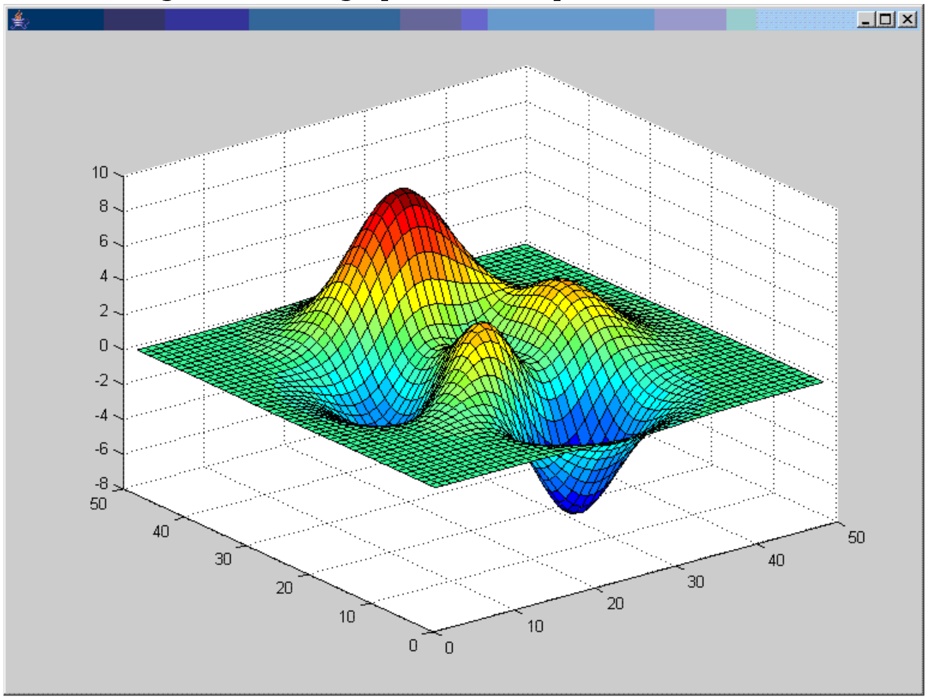
```
javac -classpath javabuilder.jar;peaks.jar SurfPeaks.java
```

Ensure that `javabuilder.jar` and `peaks.jar` (compiled in an earlier step) are both in the directory you compile from (or define their full paths).

5 Run `SurfPeaks.class` using the following `java` command. Ensure that `javabuilder.jar` and `peaks.jar` (compiled in an earlier step) are both in the directory you compiled in (or define their full paths).

```
java -classpath javabuilder.jar;peaks.jar;. SurfPeaks
```

6 The following Surf Peaks graphic should open:



Optimization Example

In this section...

About This Example

This example shows how to:

- Use MATLAB Builder for Java to create a component (`OptimDemo`) that applies MATLAB optimization routines to objective functions implemented as Java objects.
- Access the component in a Java application (`PerformOptim.java`), including use of the `MWJavaObjectRef` class to create a reference to a Java object (`BananaFunction.java`) and pass it to the component.

Note For complete reference information about the `MWArray` class hierarchy, see the `com.mathworks.toolbox.javabuilder` Javadoc package in the MATLAB Help or on the Web.

- Build and run the application.

The OptimDemo Component

The component OptimDemo finds a local minimum of an objective function and returns the minimal location and value. The component uses the MATLAB optimization function `fminsearch`, and this example optimizes the Rosenbrock banana function used in the `fminsearch` documentation. The class, Optimizer, performs an unconstrained nonlinear optimization on an objective function implemented as a Java object. A method of this class, `doOptim`, accepts an initial guess and Java object that implements the objective function, and returns the location and value of a local minimum. The second method, `displayObj`, is a debugging tool that lists the characteristics of a Java object. These two methods, `doOptim` and `displayObj`, encapsulate MATLAB functions. The MATLAB code for these two methods is in `doOptim.m` and `displayObj.m`, which can be found in `*matlabroot*\toolbox\javabuilder\Examples\ObjectRefExample\ObjectRefDemoComp`.

Optimization Example: Step-by-Step Procedure

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

   a. Copy the following directory that ships with MATLAB to your work directory: `*matlabroot*\toolbox\javabuilder\Examples\ObjectRefExample`

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

2. If you have not already done so, set the environment variables that are required on a development machine. See “Settings for Environment Variables (Development Machine)” on page 7-2.

3. Write the M-code that you want to access. This example uses `doOptim.m` and `displayObj.m`, which are already in your work directory in `ObjectRefExample\ObjectRefDemoComp`.

4. While in MATLAB, issue the following command to open the Deployment Tool dialog box:

   ```
   deploytool
   ```

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

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

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

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

9. Select **Generate Verbose Output**.

10. Add the `doOptim.m` and `displayObj.m` M-files to the project.

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

12 Build the component.

13 Write source code for a class that implements an object function to optimize. The sample application for this example is in `ObjectRefExample\ObjectRefDemoJavaApp\BananaFunction.java`. The program listing is shown here:

```
STOPSTOP BananaFunction.java
```

The class implements the Rosenbrock banana function described in the `fminsearch` documentation.

14 Write source code for an application that accesses the component. The sample application for this example is in `ObjectRefExample\ObjectRefDemoJavaApp\PerformOptim.java`. The program listing is shown here:

```
STOPSTOP PerformOptim.java
```

The program does the following:

- Instantiates an object of the `BananaFunction` class above to be optimized.
- Creates an `MWJavaObjectRef` that references the `BananaFunction` object, as shown: `origRef = new MWJavaObjectRef(objectiveFunction);`
- Instantiates an Optimizer object
- Calls the `displayObj` method to verify that the Java object is being passed correctly
- Calls the `doOptim` method, which uses `fminsearch` to find a local minimum of the objective function
- Uses a try/catch block to handle exceptions
- Frees native resources using `MWArray` methods

15 Compile the `PerformOptim.java` application and `BananaFunction.java` helper class using `javac`. When entering this command, ensure there are no spaces between pathnames in the *`matlabroot`* argument. For

example, there should be no space between `javabuilder.jar;` and `.\distrib\OptimDemo.jar` in the example below.

a Open a Command Prompt window and `cd` to the *`matlabroot`*`\work\ObjectRefExample` directory.

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

- On Windows, execute the following command:

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

- On UNIX, execute the following command:

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

16 Execute the PerformOptim class file as follows:

- On Windows:

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

- On UNIX:

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

```
PerformOptim
```

Note Valid architectures on UNIX are `glnx86`, `glnxa64`, and `sol64`.

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

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

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

h =

BananaFunction@1766806


className =

BananaFunction

  Name      Size            Bytes  Class              Attributes

  h         1x1                    BananaFunction


Methods for class BananaFunction:
```

```
BananaFunction    getClass         notifyAll
equals            hashCode         toString
evaluateFunction  notify           wait

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

directEval =

   24.2000


wrapperEval =

   24.2000


x =

    1.0000    1.0000


fval =

   8.1777e-10

Optimization successful
** Finished DOOPTIM *********************************
Location of minimum:
1.0000    1.0000
Function value at minimum:
8.1777e-10
```

Deploying a Java Component Over the Web

Creating a Deployable Web Application

In this section...

Example Overview

This example demonstrates how to display a plot created by a Java Servlet calling a component created with Java Builder over a Web interface. This example uses MATLAB `varargin` and `varargout` for optional input and output to the `varargexample.m` function. For more information about `varargin` and `varargout`, see "How Does MATLAB Builder for Java Handle Data?" on page 2-5

Before You Begin

- "Ensure You Have the Required Products" on page 6-2
- "Ensure Your Web Server is Java Compliant" on page 6-3
- "Install the javabuilder.jar Library" on page 6-3

This section describes what you need to know and do before you create the Web deployment example.

Ensure You Have the Required Products

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

MATLAB, MATLAB Compiler, MATLAB Builder for Java. This example was tested with version R2007b.

Java Development Kit (JDK). Ensure you have Sun JDK v1.6.0 installed on your system. You can download it from Sun Microsystems, Inc.

Ensure Your Web Server is Java Compliant

In order to run this example, your Web server must be capable of running accepted Java frameworks like J2EE. Running the WebFigures demo ("Delivering Interactive Graphics Over the Web with WebFigures" on page 6-9) also requires the ability to run Servlets in WARs (Web Archives).

Install the javabuilder.jar Library

Ensure that the `javabuilder.jar` library (`*matlabroot*/toolbox/javabuilder/jar/javabuilder.jar`) has been installed into your Web server's common library directory.

Download the Demo Files

Download the demo files from MATLAB Central. Search on the keyword `java_web_vararg_demo`.

Contents of the Demo Files

The demo files contain the following three directories:

- `mcode` — Contains all of the MATLAB source code.
- `JavaCode` — Contains the required Java files and libraries.
- `compile` — Contains some helpful MATLAB functions to compile and clean up the demo.

Note As an alternative to compiling the demo code manually and creating the application WAR (Web Archive) manually, you can run `compileVarArgServletDemo.m` in the compile directory. If you choose this option and wish to change the locations of the output files, edit the values in `getVarArgServletDemoSettings.m`.

If you choose to run `compileVarArgServletDemo.m`, consult the `readme` file in the download for additional information and then skip to "Running the Web Deployment Demo" on page 6-6 in this procedure.

Build Your Java Component

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

Note For a more detailed explanation of building a Java component, including further details on setting up your Java environment, the **src** and **distrib** directories, and other information, see Getting Started in the MATLAB Builder for Java User's Guide documentation.

1. Start `deploytool` from the MATLAB command prompt.
2. Select **New Project > MATLAB Builder for Java Project**.
3. Specify the project name as `vararg_java` and click **OK**.
4. In the Deployment Tool, right-click on **vararg_javaclass** and select **Add File**.
5. Using MATLAB's Current Directory Browser, navigate to the demo directory `mcode` and add the `varargexample.m` M-file to the class `vararg_javaclass` by dragging it to the **vararg_javaclass** folder in the Deployment Tool GUI.

6 Click the **Build** icon on the Deployment Tool toolbar to build your project, creating `vararg_java.jar` in the `vararg_java distrib` output directory.

Compile Your Java Code

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

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

Ensure your `classpath` is set to include:

- `javabuilder.jar` (shipped with MATLAB Builder for Java)
- `servlet-api.jar` (in the demo directory `JavaCode\lib`)

For more details about using `javac`, see the Sun Microsystems, Inc. Web site and Chapter 1, "Getting Started" in the MATLAB Builder for Java User's Guide documentation.

Generating the Web Archive (WAR) File

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

1 Copy the JAR file created using MATLAB Builder for Java into the `JavaCode\build\WEB-INF\lib` demo directory.

2 Copy the compiled Java class to the `JavaCode\build\WEB-INF\classes` demo directory.

3 Use the `jar` command to generate the final WAR as follows:

```
jar cf VarArgServlet.war -C build .
```

Caution Don't omit the `.` parameter above, which denotes current working directory.

For more information about the `jar` command, refer to the Sun Microsystems, Inc. Web site.

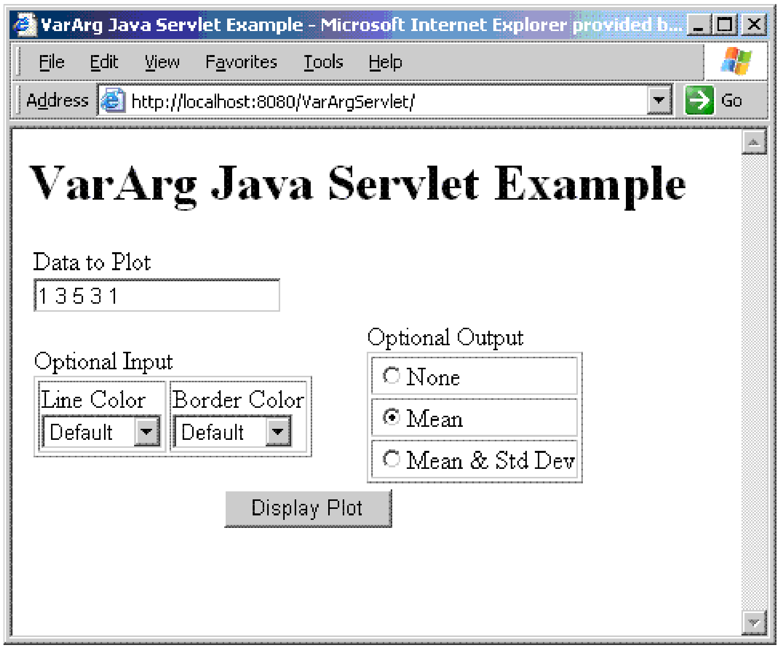
Running the Web Deployment Demo

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

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

Using the Web Application

To use the application, do the following on the `http://localhost/VarArgServlet` Web page:


VarArg Java Servlet Example - Microsoft Internet Explorer provided b...
File Edit View Favorites Tools Help
Address http://localhost:8080/VarArgServlet/
Go

VarArg Java Servlet Example

Data to Plot: 1 3 5 3 1

Optional Input
Line Color: Default
Border Color: Default

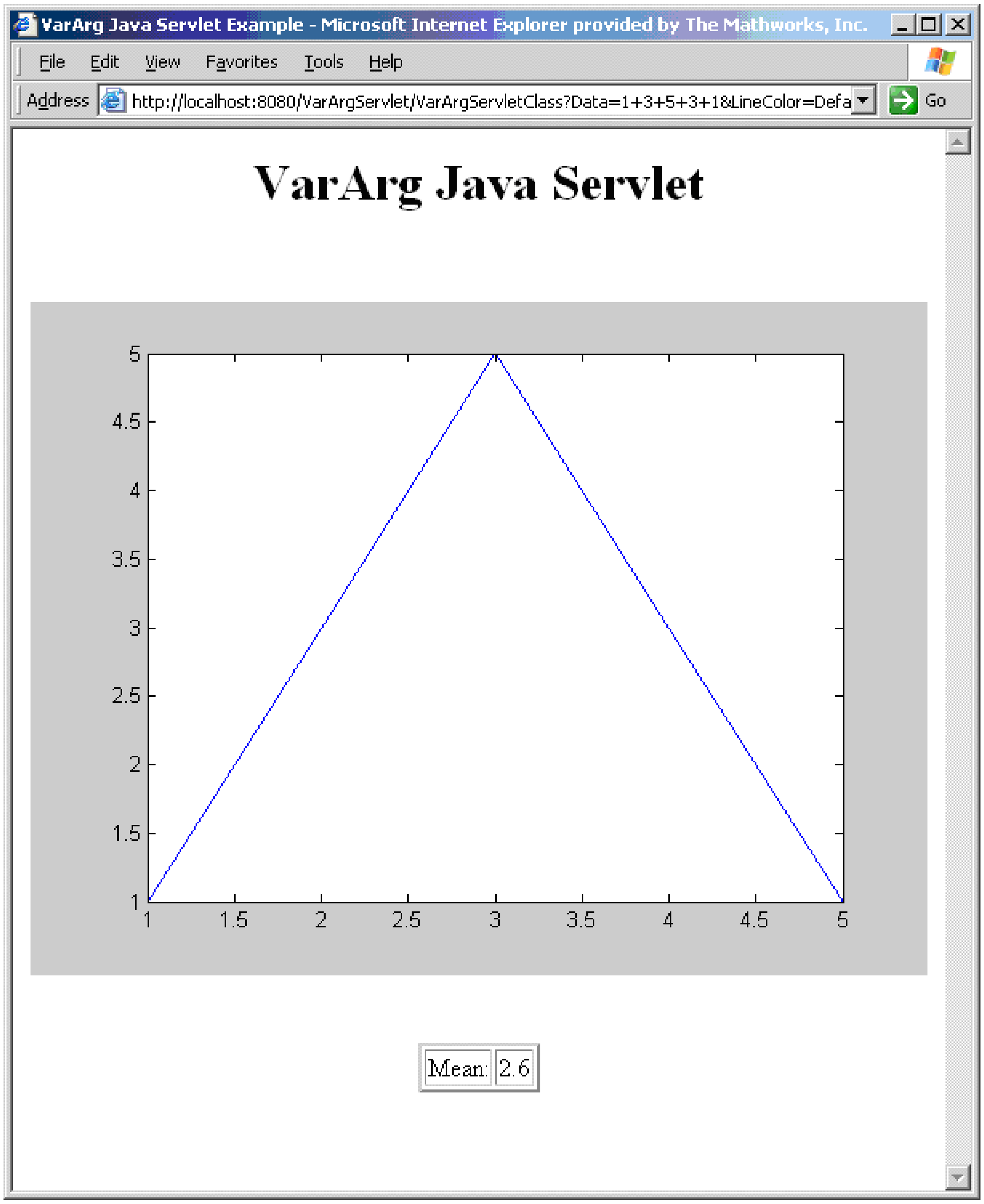
Optional Output
( ) None
(●) Mean
( ) Mean & Std Dev

1 Enter any amount of numbers to plot in the **Data to Plot** field.

2 Select **Line Color** and **Border Color** using the **Optional Input** drop-down boxes. Note that these optional inputs are passed as `varargin` to the compiled M-code.

3 Select additional information you want to output, such as mean and standard deviation, by selecting a radio button in the **Optional Output** area. Note that these optional outputs are set as `varargout` from the compiled M-code.

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


VarArg Java Servlet Example - Microsoft Internet Explorer provided by The Mathworks, Inc.
File
Edit
View
Favorites
Tools
Help
Address
Go

http://localhost:8080/VarArgServlet/VarArgServletClass?Data=1+3+5+3+1&LineColor=Defa

VarArg Java Servlet

5
4.5
4
3.5
3
2.5
2
1.5
1
1
1.5
2
2.5
3
3.5
4
4.5
5

Mean: 2.6

Delivering Interactive Graphics Over the Web with WebFigures

In this section...

Before You Begin

See the VarArg Web demo section "Before You Begin" on page 6-2 for information on properly setting up your Java environment before you run the example in this section.

Download the Example Files

You will need several files to run the example in this section. Download them from MATLAB Central. Search on the keyword `java_web_figures_demo`

The WebFigures Feature

You can provide interactive Web graphics using the WebFigures feature of Java Builder. WebFigures provides thin client delivery of interactive Web graphics through a single Web application server running a single JVM on a single physical machine.

To implement WebFigures, you utilize the Java `WebFigure` class package. This class encapsulates an individual figure and is a serializable, data-semantics class that can be attached to the following context scopes within J2EE:

- Application
- Page

- Session

Preparing to Implement WebFigures

To prepare to implement WebFigures, perform the following steps:

1. "Modifying the WebFigure Creation Code" on page 6-10
2. "Modify the Code to Attach the WebFigure to the J2EE Context Scope" on page 6-11
3. "Embed the WebFigure as Part of the Response HTML Page" on page 6-12
4. "Enable User Interaction by Creating Mapping in a Web Deployment Descriptor File" on page 6-15
5. "Enable Disposal of the WebFigure" on page 6-15

Modifying the WebFigure Creation Code

Add the code that will allow the WebFigure's creation by doing the following:

1. Add the `webfigure` statement to your existing M code. The syntax of the statement is:

   ```
   webfigure_handle = webfigure(figure_handle);
   ```

   When `webfigure` is executed, a "snapshot" or picture is taken of an open MATLAB figure. The example code below opens a new figure window, plots some data in the window, creates the `WebFigure` object, and closes the figure. Note the `WebFigure` is valid even after the MATLAB figure has been closed.

   ```
   function w = getplot
   f = figure;
   plot(1:10);
   w = webfigure(f);
   close(f);
   ```

2. Add the reference to the `WebFigure` object in your `MWArray` interface Java code so it can be received by the controller Servlet or Scriptlet as follows:

```
    // generate the plot
    Object[] results = matlabModel.getplot(1);

    //unpack the WebFigure
    MWJavaObjectRef ref = (MWJavaObjectRef)results[0];
    WebFigure f = (WebFigure)ref.get();
```

When this code is run, the `WebFigure` is created and is ready for further manipulation by the Java application's controller code.

Modify the Code to Attach the WebFigure to the J2EE Context Scope

Change the Java application's controller code to allow the figure to be accessible to the view layer. You do this by modifying the controller code so the figure is attached to a J2EE context scope, such as the session. Attaching to a session assumes that the figure will be interacted with by one user only. Attach the figure to a broader context scope, if needed, depending on the scope of your user audience. Alternate scopes are page or application.

Note For more information on Web-specific terms and concepts such as context scopes, see the Sun Microsystems, Inc. Web site.

Attach the figure to a context scope using the following syntax:

```
context_scope_object.setAttribute(name,
webfigure_object_variable);
```

For example, to attach to the session context scope from a Servlet's service method, the following statement should be added to your controller code:

```
request.getsession().setAttribute("UserPlot", f);
```

When this statement is run, the `WebFigure` will have a name (`UserPlot`) which will be referred to by the view layer in the next step.

Embed the WebFigure as Part of the Response HTML Page

To enable the user to interact with the figure, it must be embedded in an HTML page. To do this, allow the controller code to delegate displaying a response to the view layer.

Two methods are available for embedding the WebFigure, depending on whether the view layer is implemented as either a Servlet or a JSP page.

See the following table for a listing of the required and optional parameters used in the constructor and the `getHTMLEmbedString` method.

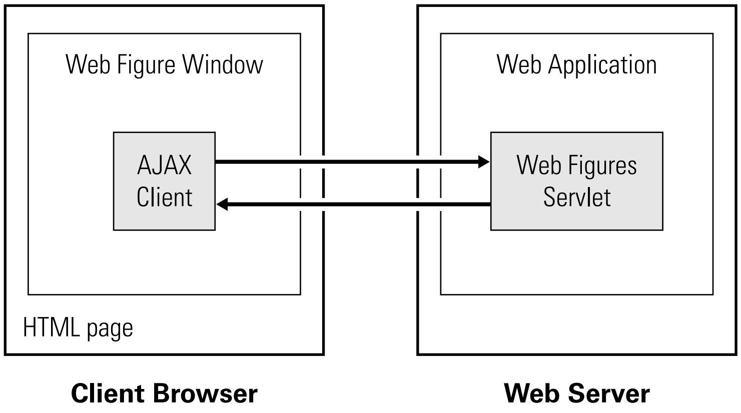
Parameter	Definition
WebFigures constructor parameters	
`root`	URL where the WebFigures servlet is mapped in the deployment descriptor (web.xml)
`servletContext`	Reference to the servlet context for the current servlet
getHTMLEmbedString method parameters	
`webfigure`	Handle to the webfigure being displayed
`name`	The attribute name used to store the WebFigure object
`scope`	The scope where the attribute is set; possible values can be `session`, `page`, or `application`
`width`	The width of the figure window in the HTML page – can be specified in pixels or as a percentage of the width of the containing HTML element **Note** If a value is not provided (for JSP) or if it is null (for Servlet), the MATLAB figure window dimensions will be used when rendering the figure on the Web page.

Parameter	Definition
`height`	The height of the figure window in the HTML page – can be specified in pixels or as a percentage of the height of the containing HTML element **Note** If a value is not provided (for JSP) or if it is null (for Servlet), the MATLAB figure window dimensions will be used when rendering the figure on the Web page.
`style`	Element-level CSS properties (from the `style` tag) used to customize the look of the figure interface on the HTML page

Embedding from a Servlet. To embed from a servlet, create an instance of the class `WebFigures` using the servlet API. Supply an argument to the `WebFigures` constructor that equates to the URL where the WebFigures servlet is mapped.

```
// The argument to the WebFigures constructor is the URL where the
// WebFigures servlet is mapped (relative to the Web application
                                  and Servlet context)
WebFigures webFigures = new WebFigures("WebFigures", f,
                                         getServletContext());
responseWriter.print(webFigures.getHtmlEmbedString(wb, name,
                                                   scope, width,
                                               height, options));
```

The following graphic depicts an AJAX client, running WebFigures, communicating with a view layer implemented as a Servlet:



Embedding from a JSP Page. To embed from the JSP page, you must import the `webfigures` tag library and use the `web-figure` tag. Do this by adding the tags displayed in the bolded lines in the following example:

```
<%@ taglib prefix="wf" uri="/WEB-INF/webfigures.tld" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<html>
   <head><title>MATLAB WebFigures Demo</title></head>
   <body>
      <table border=0 cellspacing=2 cellpadding=0 style="width:100%;height:100%">
      <tr><td>Use the controls below to interact with the surface plot.</td></tr>
      <tr><td height=100%>
      <wf:web-figure name="UserPlot" scope="session" root="WebFigures"
                     width="100%" height="100%"/>
      <td></tr>
     </table>
    </body>
</html>
```

By default, the following client settings have these values when the WebFigure is displayed for the first time:

- `zoom-to-fit` is `true` — The figure is automatically resized to fit the entire view port when the embedded HTML element is resized.

- `camera angle` is the same as in the original MATLAB figure when the WebFigure was created.

Enable User Interaction by Creating Mapping in a Web Deployment Descriptor File

Users send requests from the client to the WebFigures Servlet, which must be mapped to a URL in the Web application using WebFigures. Do this by creating or customizing the Web deployment descriptor file `/WEB-INF/web.xml`.

Use the following example template of the descriptor file, or use a similar template provided with your IDE:

```
<web-app>
    <servlet>
        <servlet-name>WebFigures</servlet-name>
        <servlet-class>
            com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WebFigures</servlet-name>
        <url-pattern>/WebFigures/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The `url-pattern` tag specifies that all URLs in `/Webfigures` are to be mapped to the `WebFiguresServlet` (accomplished by the `*` wildcard character).

Note The `WebFiguresServlet` does not have to be mapped onto `/WebFigures`. Choose the `url-pattern` most appropriate for your application.

Enable Disposal of the WebFigure

To free up resources, you must provide a method to dispose of the created WebFigure. Do this by binding the `WebFigure` to the lifetime of an HTTP session context so that it will automatically be disposed of when the session expires (via timeout, server shutdown, or explicit invalidation in Controller logic).

This is performed through use of a new utility class, `com.mathworks.toolbox.javabuilder.web.MWHttpSessionBinder`.

For example:

```
// bind the figure's lifetime to the session
sessionContext.setAttribute("UserPlotBinder", new MWHttpSessionBinder(figure));
```

This method ensures that `dispose` will be called on the given object when the attribute `UserPlotBinder` is unbound from the session. When the session expires, all its attributes are unbound. To disable this behavior after having bound the figure to the session, the following code may be used:

```
MWHttpSessionBinder binder =
      (MWHttpSessionBinder)sessionContext.getAttribute("UserPlotBinder");
binder.setObject(null);
sessionContext.removeAttribute("UserPlotBinder");
```

Setting the binder's object property to `null` ensures that `dispose` will not be called when `removeAttribute` is called.

Implementing WebFigures

You are now ready to compile and run your WebFigures application. Ensure you have the following at hand:

- From the files prepared in "Preparing to Implement WebFigures" on page 6-10:
  - M-code (`getplot.m`)
  - A controller Servlet (`ModelRunnerServlet.java`) **or** A JSP page (`response.jsp`)
  - An HTML page that serves as the entry point to the application (`index.html`)

  - The Web descriptor file, `web.xml`
- Provided as part of MATLAB Builder for Java WebFigures:
  - WebFigures Servlet (`com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet`) included in `javabuilder.jar`.
  - The WebFigures JSP tag library descriptor (`webfigures.tld`)

Perform the following steps to compile and run your WebFigures application.

1. "Compile the M-Code Into a Java Object" on page 6-17
2. "Write Code to Instantiate the Java Object and Create the WebFigure" on page 6-18
3. "Adapt the HTML to Display the WebFigure" on page 6-22
4. "Add Servlet Mapping for the WebFiguresServlet" on page 6-23
5. "Add the index.html to Create the Point of Entry for the Application" on page 6-24

Compile the M-Code Into a Java Object

Create the Java object from the plot function example outlined in "Modifying the WebFigure Creation Code" on page 6-10 by doing the following:

1. Start `deploytool` from the MATLAB command prompt.
2. Select **File > New Deployment Project**.
3. In the next screen, select **MATLAB Builder for Java** in the left pane and **Java Package** in the right pane.
4. Enter any name for the project in the **Name** field, verify or change the **Location**, and click **OK**.
5. Click **Settings**. In the **Package Name** field, specify the package name as `com_mathworks_examples_plot`.

6 In Deployment Tool, right-click the ***project_name*class** folder name, select **Rename Class** and enter the name `plotter`.

7 Add the file `getplot.m` to the **plotter** class by dragging the file from the MATLAB Current Directory browser to the class folder in the Deployment Tool.

8 Click the **Build** icon on the Deployment Tool toolbar to build your project, creating `plot.jar` in the **distrib** output directory.

9 Copy the `plot.jar` files to your Web application directory tree `WEB-INF/lib/`.

Write Code to Instantiate the Java Object and Create the WebFigure

Add code to the `ModelRunnerServlet.java` controller Servlet to create the Java object and the WebFigure in real-time.

1 Add the following code to the `super.init()` method to instantiate the object:

```
try {
    // create a new plotter object
    matlabModel = new plotter();
}
catch (MWException mcrInitError) {
    mcrInitError.printStackTrace();
}
```

2 Add the following code to the `doGet()` method to create the WebFigure object

```
// find the plotter object associated with this session
WebFigure userPlot = (WebFigure)session.getAttribute("UserPlot");

// if this is the first time doGet has been called for this session,
//  create the plot and WebFigure object
if (null == userPlot) {
    try {
```

```
                // generate the plot
                Object[] results = matlabModel.getplot(/* nargout = */ 1);

                try {
                    // unpack the WebFigure
                    MWJavaObjectRef ref = (MWJavaObjectRef)results[0];
                    userPlot = (WebFigure)ref.get();

                    // store the figure in the session context
                    session.setAttribute("UserPlot", userPlot);

                    // bind the figure's lifetime to the session
                    session.setAttribute("UserPlotBinder",
                                         new MWHttpSessionBinder(userPlot));
                }
                finally {
                    // free MCR-related resources held by the results
                    MWArray.disposeArray(results);
                }
            }
            catch (MWException getplotError) {
                getplotError.printStackTrace();
            }
        }
```

The complete `ModelRunnerServlet.java` program should look like this:

```
// imported classes from JRE
import java.io.IOException;

// imported classes from servlet-api.jar
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.RequestDispatcher;
```

```
// imported classes from javabuilder.jar
import com.mathworks.toolbox.javabuilder.webfigures.WebFigure;
import com.mathworks.toolbox.javabuilder.MWJavaObjectRef;
import com.mathworks.toolbox.javabuilder.MWException;
import com.mathworks.toolbox.javabuilder.MWArray;
import com.mathworks.toolbox.javabuilder.web.MWHttpSessionBinder;

// imported classes from plot.jar
import com.mathworks.examples.plot.Plotter;

public class ModelRunnerServlet extends HttpServlet
{
    private plotter matlabModel = null;

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);


        try {
            // create a new plotter object
            matlabModel = new plotter();
        }
        catch (MWException mcrInitError) {
            mcrInitError.printStackTrace();
        }
    }

    public void destroy()
    {
        super.destroy();
        // free MCR-related resources
        matlabModel.dispose();
    }

    protected void doGet(final HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        HttpSession session = request.getSession();
        ServletContext servletContext = session.getServletContext();
```

```
            // find the plotter object associated with this session
            WebFigure userPlot = (WebFigure)session.getAttribute("UserPlot");

            // if this is the first time doGet has been called for this session,
            //  create the plot and WebFigure object
            if (null == userPlot) {
                try {
                    // generate the plot
                    Object[] results = matlabModel.getplot(/* nargout = */ 1);

                    try {
                        // unpack the WebFigure
                        MWJavaObjectRef ref = (MWJavaObjectRef)results[0];
                        userPlot = (WebFigure)ref.get();

                        // store the figure in the session context
                        session.setAttribute("UserPlot", userPlot);

                        // bind the figure's lifetime to the session
                        session.setAttribute("UserPlotBinder",
                                             new MWHttpSessionBinder(userPlot));
                    }
                    finally {
                        // free MCR-related resources held by the results
                        MWArray.disposeArray(results);
                    }
                }
                catch (MWException getplotError) {
                    getplotError.printStackTrace();
                }
            }

            // forward the request to the View layer (response.jsp)
            RequestDispatcher dispatcher = request.getRequestDispatcher("/response.jsp");
            dispatcher.forward(request, response);
        }
    }
```

Adapt the HTML to Display the WebFigure

Customize the `response.jsp` code in `webfigures.tld` to display the WebFigure to the user by doing the following:

1. Copy `webfigures.tld` from *`matlabroot`*`/toolbox/javabuilder/webfigures/webfigures.tld` to the directory `WEB-INF/` under the Web application's directory tree.

2. Add the following declaration to reference `webfigures.tld` in the file `response.jsp`:

   ```
   <%@ taglib prefix="wf" uri="/WEB-INF/webfigures.tld" %>
   ```

3. Add the following `web-figure` tag to display the figure:

   ```
   <wf:web-figure name="UserPlot" scope="session" root="WebFigures"
                  width="100%" height="100%"/>
   ```

   When finished, the `response.jsp` page code should look like this:

   ```
   <%@ taglib prefix="wf" uri="/WEB-INF/webfigures.tld" %>
   <%@ page contentType="text/html;charset=UTF-8" language="java" %>

   <html>
     <head><title>MATLAB WebFigures Demo</title></head>
     <body>
       <table border=0 cellspacing=2 cellpadding=0 style="width:100%;height:100%">
         <tr><td>
             Use the console below to interact with the surface plot.
         </td></tr>
         <tr><td height=100%>
             <wf:web-figure name="UserPlot" scope="session" root="WebFigures"
                            width="100%" height="100%"/>
         </td></tr>
       </table>
     </body>
   </html>
   ```

Add Servlet Mapping for the WebFiguresServlet

Add the following mapping to `WEB-INF/web.xml` so that the Servlet can locate the WebFigure:

```
<servlet>
    <servlet-name>WebFigures</servlet-name>
    <servlet-class>
        com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>WebFigures</servlet-name>
    <url-pattern>/WebFigures/*</url-pattern>
</servlet-mapping>
```

Afterwards, the complete `WEB-INF/web.xml` file should look similar to this:

```
<?xml version="version_number" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
         <servlet-name>ModelRunnerServlet</servlet-name>
         <servlet-class>ModelRunnerServlet</servlet-class>
    </servlet>
    <servlet-mapping>
         <servlet-name>ModelRunnerServlet</servlet-name>
         <url-pattern>/run_model</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>WebFigures</servlet-name>
        <servlet-class>
        com.mathworks.toolbox.javabuilder.webfigures.WebFiguresServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WebFigures</servlet-name>
        <url-pattern>/WebFigures/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Add the index.html to Create the Point of Entry for the Application

Add the following `index.html` code:

```
<html>
<head><title>Welcome to the MATLAB WebFigures demo</title></head>
<body>Click <a href="run_model">here</a> to begin.</body>
</html>
```

End-User Interaction with WebFigures

An end-user can interact with a deployed WebFigure. They can use the dynamic menu interface to:

- Zoom (increase or decrease the size of the figure)
- Pan (change placement of figure on screen)
- Rotate (manipulate a figure to see all perspectives)

To use the dynamic menu interface, hover the mouse over the top of the WebFigure until you see three icons: a magnifying glass (zoom icon) a hand (pan icon) and a circle with a wraparound arrow (rotation icon). Click on the icon that corresponds to the action you want to accomplish and use the mouse to manipulate the figure.

Note Currently, the WebFigures axis can only accommodate one subplot.

The following limitations currently exist from the end-user interface:

- Guide GUIs cannot be deployed as WebFigures
- Multiple sub-plotting in figures is available, but it is not possible to interact with all sub-plots.

Creating Scalable Web Applications With RMI

In this section...

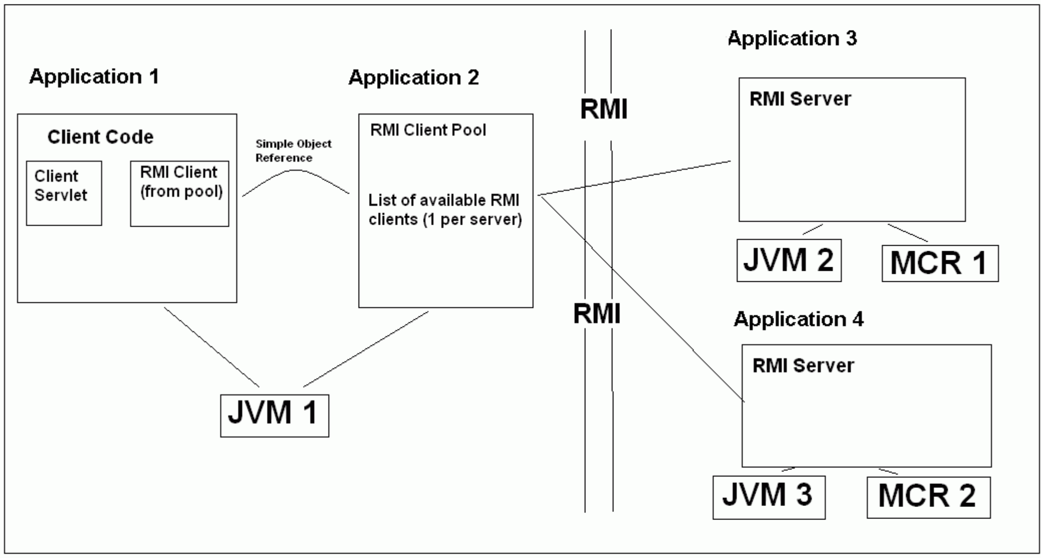
Using RMI

You can expand your application's throughput capacity by taking advantage of Java Builder's use of RMI, Java's native remote procedure call (RPC) mechanism. Java Builder's implementation of RMI technology provides for automatic generation of interface code to enable components to be started in separate processes, on one or more computers, making your applications scalable and adaptable to future performance demands.

The following example utilizes RMI in the following ways:

- Running a client and server on a single machine
- Running a client on one machine and a server on another

Note While running on UNIX, ensure you use `:` as the path separator in calls to `java` and `javac`. `;` is used as a path separator only on Windows.



Before You Begin

See the VarArg Web demo section "Before You Begin" on page 6-2 for information on properly setting up your Java environment before you run the example in this section.

Running Client and Server On a Single Machine

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

1 Compile the Java Builder component by issuing the following at the MATLAB Command Prompt:

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

```
updateField.m -v
```

2 Compile the server Java code by issuing the following `javac` command. Ensure there are no spaces between `javabuilder.jar;` and *`directory containing component.`*

```
 javac -classpath
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
directory_containing_component\dataTypesComp.jar
DataTypesServer.java
```

3 Compile the client Java code by issuing the following `javac` command. Ensure there are no spaces between `javabuilder.jar;` and *`directory containing component`*.

```
javac -classpath
matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
directory containing component\dataTypesComp.jar
DataTypesClient.java
```

4 Run the client and server as follows:

   a Open two command windows on DOS or UNIX, depending on what platform you are using.

   b If running Windows, ensure that *`matlabroot`*`/bin/`*`arch`* is defined on the system path. If running UNIX, ensure `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` are set properly.

   c Run the server by issuing the following `java` command. Ensure there are no spaces between `dataTypesComp.jar;` and *`matlabroot`*.

```
 java -classpath
 .;directory_containing_component\dataTypesComp.jar;
matlabroot\toolbox\javabuilder\jar\javabuilder.jar

-Djava.rmi.server.codebase="file:///matlabroot/toolbox/javabuilder/jar/javabuilder.jar
 file:///directory_containing_component/dataTypesComp.jar" DataTypesServer
```

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

```
java -classpath
```

```
 .;directory_containing_component\dataTypesComp.jar;
matlabroot\toolbox\javabuilder\jar\javabuilder.jar
 DataTypesClient
```

5 If successful, the following output appears in the command window running the server:

```
Please wait for the server registration notification.
            Server registered and running successfully!!

            EVENT 1: Initializing the structure on server
                     and sending it to client:
                     Initialized empty structure:

                   Name: []
                Address: []

            ##################################

            EVENT 3: Partially initialized structure as received by server:

                   Name: []
                Address: [1x1 struct]

            Address field as initialized from the client:

                Street: '3, Apple Hill Drive'
                  City: 'Natick'
                 State: 'MA'
                   Zip: '01760'

            ##################################

            EVENT 4: Updating 'Name' field before sending the
structure back to the client:

                  Name: 'The MathWorks'
               Address: [1x1 struct]

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

If successful, the following output appears in the command window running the client:

```
Running the client application!!

           EVENT 2: Initialized structure as received in client applications:

                  Name: []
               Address: []

           Updating the 'Address' field to :

               Street: '3, Apple Hill Drive'
                 City: 'Natick'
                State: 'MA'
                  Zip: '01760'

           #################################


           EVENT 5: Final structure as received by client:

                  Name: 'The MathWorks'
               Address: [1x1 struct]

           Address field:

               Street: '3, Apple Hill Drive'
                 City: 'Natick'
                State: 'MA'
                  Zip: '01760'

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

Running Client and Server On Separate Machines

To implement RMI with a client on one machine and a server on another, you must:

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

After this is done, follow the steps in “Running Client and Server On a Single Machine” on page 6-27 .

Reference Information for Java

Requirements for MATLAB Builder for Java

In this section...

System Requirements

System requirements and restrictions on use for MATLAB Builder for Java are as follows:

- All requirements for MATLAB Compiler; see "Installation and Configuration" in the MATLAB Compiler documentation.
- Java Development Kit (JDK) 1.4 or later must be installed.
- Java Runtime Environment (JRE) that is used by MATLAB and MCR.

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

Limitations and Restrictions

In general, limitations and restrictions on the use of Java Builder are the same as those for MATLAB Compiler. See "Limitations and Restrictions" in the MATLAB Compiler documentation for details.

Settings for Environment Variables (Development Machine)

Before starting to program, you must set the environment on your development machine to be compatible with MATLAB Builder for Java.

Specify the following environment variables:

- “JAVA_HOME Variable” on page 7-3
- “Java CLASSPATH Variable” on page 7-4
- “Native Library Path Variables” on page 7-6

JAVA_HOME Variable

Java Builder uses the `JAVA_HOME` variable to locate the Java Software Development Kit (SDK) on your system. It also uses this variable to set the versions of the `javac.exe` and `jar.exe` files it uses during the build process.

Note If you do not set `JAVA_HOME`, Java Builder assumes that `\jdk\bin` is on the system path.

Setting JAVA_HOME on Windows (Development Machine). If you are working on Windows, set your `JAVA_HOME` variable by entering the following command in your DOS command window. (In this example, your Java SDK is installed in directory `C:\java\jdk\j2sdk1.6.0`.)

```
set JAVA_HOME=C:\java\jdk\j2sdk1.6.0
```

Alternatively, you can add *`jdk_directory`*`/bin` to the system path. For example:

```
set PATH=%PATH%;c:\java\jdk\j2sdk1.6.0\bin
```

You can also set these variables globally using the Windows Control Panel. Consult your Windows documentation for instructions on setting system variables.

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

Setting JAVA_HOME on UNIX (Development Machine). If you are working on a UNIX system, set your `JAVA_HOME` variable by entering the following commands at the command prompt. (In this example, your Java SDK is installed in directory `/java/jdk/j2sdk1.6.0`.)

```
setenv JAVA_HOME /java/jdk/j2sdk1.6.0
```

Alternatively, you can add `jdk_directory\bin` to the system path.

Java CLASSPATH Variable

To build and run a Java application that uses a Java Builder generated component, the system needs to find `.jar` files containing the MATLAB libraries and the class and method definitions that you have developed and built with Java Builder. To tell the system how to locate the `.jar` files it needs, specify a `classpath` either in the `javac` command or in your system environment variables.

Java uses the `CLASSPATH` variable to locate user classes needed to compile or run a given Java class. The class path contains directories where all the `.class` and/or `.jar` files needed by your program reside. These `.jar` files contain any classes that your Java class depends on.

When you compile a Java class that uses classes contained in the `com.mathworks.toolbox.javabuilder` package, you need to include a file called `javabuilder.jar` on the Java class path. This file comes with Java Builder; you can find it in the following directory:

```
matlabroot/toolbox/javabuilder/jar % (development machine)
mcrroot/toolbox/javabuilder/jar % (end-user machine)
```

where *`matlabroot`* refers to the root directory into which the MATLAB installer has placed the MATLAB files, and `mcrroot` refers to the root directory under which mcr is installed. Java Builder automatically includes this `.jar` file on the class path when it creates the component. To use a class generated by Java Builder, you need to add this *`matlabroot`*`/toolbox/javabuilder/jar/javabuilder.jar` to the class path.

In addition, you need to add to the class path the `.jar` file created by Java Builder for your compiled `.class` files.

Example: Setting CLASSPATH on Windows. Suppose your MATLAB libraries are installed in `C:\`*`matlabroot`*`\bin\win32`, and your component `.jar` files are in `C:\mycomponent`.

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

To set your `CLASSPATH` variable on your development machine, enter the following command at the DOS command prompt:

```
set CLASSPATH=.;C:\matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
C:\mycomponent\mycomponent.jar
```

Alternatively, if the Java SDK is installed, you can specify the class path on the Java command line as follows. When entering this command, ensure there are no spaces between pathnames in the *`matlabroot`* argument. For example, there should be no space between `javabuilder.jar;` and `c:\mycomponent\mycomponent.jar` in the example below.

```
javac
   -classpath .;C:\matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
   C:\mycomponent\mycomponent.jar usemyclass.java
```

where `usemyclass.java` is the file to be compiled.

It is recommended that you globally add any frequently used class paths to the `CLASSPATH` system variable via the Windows Control Panel.

Example: Setting CLASSPATH on UNIX (Development Machine). Suppose your UNIX environment is as follows:

- Your MATLAB libraries are installed in `/`*`matlabroot`*`/bin/`*`arch`*, (where *`arch`* is either `glnx86`, `glnxa64`, `mac`, or `sol64`, depending on the operating system of the machine.
- Your component `.jar` files are in `/mycomponent`.

To set your `CLASSPATH` variable, enter the following command at the prompt:

```
setenv CLASSPATH .:/matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
/mycomponent/mycomponent.jar
```

Like Windows, you can specify the class path directly on the Java command line. To compile `usemyclass.java`, type the following:

```
javac -classpath
.:/matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
/mycomponent/mycomponent.jar usemyclass.java
```

where `usemyclass.java` is the file to be compiled.

Native Library Path Variables

The operating system uses the native library path to locate native libraries that are needed to run your Java class. See the following list of variable names according to operating system:

Windows	`PATH`
Linux	`LD_LIBRARY_PATH`
Solaris	`LD_LIBRARY_PATH`
Macintosh	`DYLD_LIBRARY_PATH`

For information on how to set these path variables, see the following topics in the "Standalone Applications" section of the MATLAB Compiler documentation:

- See "Testing the Application" for information on setting your path on a development machine.
- See "Running the Application" for information on setting your path on an end-user machine.

Data Conversion Rules

In this section…

Java to MATLAB Conversion

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

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

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

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

Java to MATLAB Conversion Rules

Java Type	MATLAB Type
`double`	`double`
`float`	`single`
`byte`	`int8`
`int`	`int32`
`short`	`int16`
`long`	`int64`
`char`	`char`

Java to MATLAB Conversion Rules (Continued)

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

MATLAB to Java Conversion

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

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

MATLAB to Java Conversion Rules

MATLAB Type	Java Type (Primitive)	Java Type (Object)
`cell`	N/A	`Object` **Note** Cell arrays are constructed and accessed as arrays of `MWArray`.
`structure`	N/A	`Object` **Note** Structure arrays are constructed and accessed as arrays of `MWArray`.
`char`	`char`	`java.lang.Character`
`double`	`double`	`java.lang.Double`
`single`	`float`	`java.lang.Float`
`int8`	`byte`	`java.lang.Byte`
`int16`	`short`	`java.lang.Short`
`int32`	`int`	`java.lang.Integer`
`int64`	`long`	`java.lang.Long`
`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.

MATLAB to Java Conversion Rules (Continued)

MATLAB Type	Java Type (Primitive)	Java Type (Object)
`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	
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 Java Builder

In this section...

APIs Based on MATLAB Function Signatures

Java Builder generates two kinds of interfaces to handle MATLAB function signatures.

- A *standard* signature in Java.

  This interface specifies input arguments for each overloaded method as one or more input arguments of class `java.lang.Object` or any subclass (including subclasses of `MWArray`). The standard interface specifies return values, if any, as a subclass of `MWArray`.

- `mlx` API

  This interface allows the user to specify the inputs to a function as an Object array, where each array element is one input argument. Similarly, the user also gives the mlx interface a preallocated Object array to hold the outputs of the function. The allocated length of the output array determines the number of desired function outputs.

  The mlx interface may also be accessed using `java.util.List` containers in place of Object arrays for the inputs and outputs. Note that if List containers are used, the output List passed in must contain a number of elements equal to the desired number of function outputs.

  For example, this would be incorrect usage:

  ```
  java.util.List outputs = new ArrayList(3);
  myclass.myfunction(outputs, inputs); // outputs contains 0 elements!
  ```

  And this would be the correct usage:

```
    java.util.List outputs = Arrays.asList(new Object[3]);
    myclass.myfunction(outputs, inputs); // ok, list contains 3 elements
```

Typically you use the standard interface when you want to call MATLAB functions that return a single array. In other cases you probably need to use the `mlx` interface.

Standard API

The standard calling interface returns an array of one or more `MWArray` objects.

The standard API for a generic function with none, one, more than one, or a variable number of arguments, is shown in the following table.

Arguments	API to Use
Generic MATLAB function	`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 )`

Arguments	**API to Use**

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

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

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

Argument	**Description**	**Details About Argument**
`numArgsOut`	Number of outputs	An integer indicating the number of outputs you want the method to return. To return no arguments, omit this argument. The value of *`numArgsOut`* must be less than or equal to the MATLAB function `nargout`. The *`numArgsOut`* argument must always be the first argument in the list.

Argument	Description	Details About Argument
In1, In2, ...InN	Required input arguments	All arguments that follow *numArgsOut* in the argument list are inputs to the method being called. Specify all required inputs first. Each required input must be of class `MWArray` or any class derived from `MWArray`.
varargin	Optional inputs	You can also specify optional inputs if your M-code uses the `varargin` input: list the optional inputs, or put them in an `Object[]` argument, placing the array last in the argument list.
Out1, Out2, ...OutN	Output arguments	With the standard calling interface, all output arguments are returned as an array of `MWArrays`.

mlx API

For a function with the following structure:

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

Java Builder generates the following API, as the mlx interface:

```
public void foo (List outputs, List inputs) throws MWException;
public void foo (Object[] outputs, Object[] inputs) throws MWException;
```

Code Fragment: Signatures Generated for myprimes Example

For a specific example, look at the `myprimes` method. This method has one input argument, so Java Builder generates three overloaded methods in Java.

When you add `myprimes` to the class `myclass` and build the component, Java Builder generates the `myclass.java` file. A fragment of `myclass.java` is listed to show the three overloaded implementations of the `myprimes` method in the Java code. The first implementation shows the interface to be used if there are no input arguments, the second shows the implementation to be used if there is one input argument, and the third shows the feval 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)
   }
```

The standard interface specifies inputs to the function within the argument list and outputs as return values.

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.

See “APIs Based on MATLAB Function Signatures” on page 7-11 for details about the interfaces.

MWArray Class Specification

For complete reference information about the `MWArray` class hierarchy, see `com.mathworks.toolbox.javabuilder.MWArray`, which is in the *`matlabroot`*`/help/toolbox/javabuilder/MWArrayAPI/` directory.

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

Using the Command-Line Interface

You can use the MATLAB command-line interface (or the operating system command line), instead of the GUI to create Java objects. Do this by issuing the `mcc` command with options. If you use `mcc`, you do not create a project.

The following table provides an overview of some `mcc` options related to creating Java components, along with syntax and examples of their usage.

Using the Command Line to Create Java Components

Action to Perform	mcc Option to Use	Description
Create a class encapsulating one or more M-files.	`-W java:`	Tells Java Builder to generate a Java component that contains a class that encapsulates the specified files.
	Syntax `mcc -W 'java:component_name[,class_name]' file1 [file2...fileN]` *`component_name`* is a fully qualified package name for your component. The name is a period-separated list. *`class_name`* is the name for the Java class to be created. The default *`class_name`* is the last item in the list specified by *`component_name`*. *`file1`* [*`file2...fileN`*] are M-files to be encapsulated as methods in *`class_name`*.	

	Example
	`mcc -W 'java:com.mycompany.mycomponent,myclass'`
	`     foo.m bar.m`
	
	The example creates a Java component that has a fully qualified package name, `com.mycompany.mycomponent`. The component contains a single Java class, `myclass`, which contains methods `foo` and `bar`.
	
	To use `myclass`, place the following statement in your code:
	`import com.mycompany.mycomponent.myclass;`	

Using the Command Line to Create Java Components (Continued)

Action to Perform	mcc Option to Use	Description
Add additional classes to a Java component.	`class{...}`	Used with `-W java:`. Tells Java Builder to create *`class_name`*, which encapsulates one or more M-files that are specified in a comma-separated list.
	Syntax `class{`*`class_name`*`:`*`file1`*` [`*`file2...fileN`*`]}`	

	Example
	`mcc -W 'java:com.mycompany.mycomponent,myclass'`
	`    foo.m bar.m class{myclass2:foo2.m,bar2.m}`
	The example creates a Java component named `mycomponent` with two classes:
	`myclass` has methods `foo` and `bar`.
	`myclass2` has methods `foo2` and `bar2`.	

Simplify the command-line input for components.	`-B`	Tells Java Builder to replace a specified file with the command-line information it contains.
	Syntax `mcc -B '`*`bundlefile`*`'[:`*`arg1`*`, `*`arg2`*`, ..., `*`argN`*`]`	

	Example
	Suppose a `myoptions` file contains
	`-W 'java:mycomponent,myclass'`
	In this case,
	`mcc -B 'myoptions' foo.m bar.m`
	produces the same results as
	`mcc -W 'java:[mycomponent,myclass]' foo.m bar.m`	

Using the Command Line to Create Java Components (Continued)

Action to Perform	mcc Option to Use	Description
Control how each Java class uses the MCR.	`-S`	Tells Java Builder to create a single MCR when the first Java class is instantiated. This MCR is reused and shared among all subsequent class instances within the component, resulting in more efficient memory usage and eliminating the MCR startup cost in each subsequent class instantiation. By default, a new MCR instance is created for each instance of each Java class in the component. Use `-S` to change the default. When using `-S`, note that all class instances share a single MATLAB workspace and share global variables in the M-files used to build the component. This makes properties of a Java class behave as static properties instead of instance-wise properties.
	Example `mcc -S 'java:mycomponent,myclass' foo.m bar.m` The example creates a Java component called `mycomponent` containing a single Java class named `myclass` with methods `foo` and `bar`. (See the first example in this table). If and when multiple instances of `myclass` are instantiated in an application, only one MCR is initialized, and it is shared by all instances of `myclass`.	
Specify a directory for output	`-d` *`directoryname`*	Tells Java Builder to create a directory and copy the output files to it. (If you use `mcc` instead of the GUI, the *`project_directory`*`\src` and *`project_directory`*`\distrib` directories are not automatically created.)

Note All of these command-line examples produce the `mycomponent.jar` file (component `jar` file)

Notice that the component name used to create these files is derived from the last item on the period-separated list that specifies the fully qualified name of the class.

8

Functions — Alphabetical List

deploytool

Purpose Open GUI for MATLAB Builder for Java and MATLAB Compiler

Syntax

```
deploytool
```

Description The `deploytool` command opens the Deployment Tool dialog box, which is the graphical user interface (GUI) for MATLAB Builder for Java and for MATLAB Compiler.

Examples

Use this list to find examples in the documentation.

Handling Data

Handling Errors

Handling Memory

COM Components

Sample Applications (Java)

Index