MATLAB® Builder for Java™

The Language of Technical Computing

User’s Guide

Version 1
## Getting Started

### 1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is MATLAB Builder for Java?</td>
<td>1-2</td>
</tr>
<tr>
<td>Support for MATLAB Features in Java</td>
<td>1-2</td>
</tr>
<tr>
<td>Known Issue in Data Returned by toArray Referencing</td>
<td>1-2</td>
</tr>
<tr>
<td>Sparse Format</td>
<td>1-2</td>
</tr>
<tr>
<td>Using the Deployment Tool</td>
<td>1-3</td>
</tr>
<tr>
<td>Creating a Java Component</td>
<td>1-5</td>
</tr>
<tr>
<td>Using the Command-Line Interface</td>
<td>1-6</td>
</tr>
<tr>
<td>Developing an Application</td>
<td>1-12</td>
</tr>
<tr>
<td>Deploying an Application</td>
<td>1-14</td>
</tr>
<tr>
<td>Example: Magic Square</td>
<td>1-15</td>
</tr>
<tr>
<td>Magic Square Example: Step-by-Step Procedure</td>
<td>1-15</td>
</tr>
<tr>
<td>Understanding the Magic Square Example</td>
<td>1-23</td>
</tr>
<tr>
<td>Importing Classes</td>
<td>1-23</td>
</tr>
<tr>
<td>Creating an Instance of the Class</td>
<td>1-23</td>
</tr>
<tr>
<td>Calling Class Methods from Java</td>
<td>1-23</td>
</tr>
<tr>
<td>For More Information</td>
<td>1-25</td>
</tr>
</tbody>
</table>

### 2

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is a Project?</td>
<td>2-3</td>
</tr>
<tr>
<td>Classes and Methods</td>
<td>2-3</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>2-3</td>
</tr>
</tbody>
</table>
How Does MATLAB Builder for Java Handle Data? .................. 2-4
Understanding the API Data Conversion Classes .................. 2-4
Automatic Conversion to MATLAB Types ......................... 2-5
Understanding Function Signatures Generated by Java Builder ........................................ 2-6
Returning Data from MATLAB to Java ........................... 2-7

What Happens in the Build Process? ................................. 2-9

What Happens in the Package Process? .............................. 2-10

How Does Component Deployment Work? .......................... 2-11

Programming

3

Import Classes ...................................................... 3-2

Creating an Instance of the Class ................................ 3-3
Code Fragment: Instantiating a Java Class ......................... 3-3

Passing Arguments to and from Java ............................ 3-6
Manual Conversion of Data Types ................................. 3-6
Automatic Conversion to a MATLAB Type ......................... 3-7
Specifying Optional Arguments .................................. 3-9
Handling Return Values .......................................... 3-14

Handling Errors ................................................... 3-19
Handling Checked Exceptions ...................................... 3-19
Handling Unchecked Exceptions .................................. 3-22

Managing Native Resources ...................................... 3-25
Using Garbage Collection Provided by the JVM .................. 3-25
Using the dispose Method ......................................... 3-26
Overriding the Object.Finalize Method ......................... 3-27
Handling Data Conversion Between Java and MATLAB

Calling MWArray Methods ........................................ 3-28

Using MWArray Classes

Guidelines for Working with MWArray Classes ........... 4-2
Overview of the MWArray API ................................. 4-2
Understanding the MWArray Base Class ............... 4-2
Constructing Numeric Arrays ......................... 4-7
Working with Logical Arrays .......................... 4-22
Working with Character Arrays ................. 4-26
Working with Cell Arrays ......................... 4-31

Using Class Methods ........................................ 4-38
Using MWArray ........................................ 4-38
Using MWNumericArray ................................. 4-58
Using MWLogicalArray ................................. 4-92
Using MWCharArray .................................. 4-107
Using MWStructArray ................................. 4-118
Using MWCCellArray ................................. 4-135
Using MWClassID .................................. 4-149
Using MWComplexity ................................ 4-152

Sample Applications (Java)

Plot Example ........................................ 5-2
Spectral Analysis Example ................................. 5-8
Matrix Math Example ........................................ 5-16
Understanding the getfactor Program ............... 5-26
# Getting Started

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is MATLAB Builder for Java? (p. 1-2)</td>
<td>Brief description of what the product does and how it works</td>
</tr>
<tr>
<td>Creating a Java Component (p. 1-5)</td>
<td>Step-by-step procedure to create and package a Java™ component that encapsulates MATLAB® code</td>
</tr>
<tr>
<td>Developing an Application (p. 1-12)</td>
<td>Step-by-step procedure to access the component in an application</td>
</tr>
<tr>
<td>Deploying an Application (p. 1-14)</td>
<td>What you have to do to support end users who run applications using your components</td>
</tr>
<tr>
<td>Example: Magic Square (p. 1-15)</td>
<td>Step-by-step example, including code for a simple MATLAB function and a simple Java application</td>
</tr>
<tr>
<td>Understanding the Magic Square Example (p. 1-23)</td>
<td>Details about the code in the Magic Square application</td>
</tr>
<tr>
<td>For More Information (p. 1-25)</td>
<td>Where to find out about concepts, techniques, examples, and reference information</td>
</tr>
</tbody>
</table>
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, you must include supporting files generated by Java Builder as well as the MATLAB Component Runtime (MCR), which is provided by the product. Your users do not have to purchase and install MATLAB.

**Note** MATLAB Builder for Java is also referred to in this documentation as Java Builder, for ease of use.

Support for MATLAB Features in Java

Java Builder provides robust data conversion, indexing, and array formatting capabilities to preserve the flexibility of MATLAB when called from Java code. To support the MATLAB data types, Java Builder provides the `MWArray` class hierarchy. You can use `MWArray` and other Java class members in your application to convert native arrays to MATLAB arrays and vice versa. Java Builder also provides automatic data conversion for passing arguments that are Java types.

Known Issue in Data Returned by `toArray` Referencing Sparse Format

There is a known issue where the data returned by calling `toArray` on a `MWNumericArray` or `MWLogicalArray` object that references a MATLAB array stored in sparse format may be incorrect or corrupted. If this use case applies to your work, see “Version 1.0 (R2006b) MATLAB Builder for Java” in the Release Notes for a link to information and a patch to fix this problem.
Using the Deployment Tool
The Deployment Tool provides a graphical user interface to Java Builder. While you are still in MATLAB, issue the following command to open the Deployment Tool:

deploystream
Use the Deployment Tool to perform the following tasks:
Creating a Java Component

To create a component you need to write M-code (or use existing code) and then create a project in MATLAB Builder for Java that encapsulates the code. In general, the steps are as follows:

1. Write, test, and save the MATLAB code to be used as the basis for the Java component.

2. Set the environment variables that are required on a development machine. See “Settings for Environment Variables (Development Machine)” on page 6-2.

3. While you are still in MATLAB, issue the following command to open the Deployment Tool:

```
deploytool
```

4. Use the Deployment Tool to create a project that contains one or more classes.
   a. Create the project by clicking the New Project icon in the toolbar.
   b. Specify the project name and location.

      By default the project name is assigned to be the name of the package to be created. You can change the default.
   c. Add class names for classes that you want to create as part of the Java package.
   d. Add one or more M-files that you want to encapsulate in each class.
   e. Add helper files as needed to support the classes.
   f. Save the project.

5. Build the package.

   The build process for a project copies a Java wrapper class in the `\src` subdirectory of your project directory. It also copies a `.jar` file and `.ctf` file in the `\distrib` subdirectory of your project directory. The files in the `\distrib` directory define your Java component.
The .ctf is a component technology file, which is required to support components that encapsulate MATLAB functions when running them on a user machine that does not have the MATLAB desktop installed.

6 Test the component and rebuild it as needed.

You probably want to test your component before using it in an application or before preparing it for use by others. After testing the component on your development platform, you can reopen the project if necessary and proceed to the next step.

7 Optionally, create a package to distribute the component and the required files to developers. This step is necessary only if you want to make the component available to other application developers on a different development machine.

**Note** On Windows platforms, the Deployment Tool creates a self-extracting executable. On non-Windows platforms, the package is a .zip file rather than a self-extracting executable.

8 Save the project.

Java Builder saves the project in a .prj file.

**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.

**Note** See the MATLAB Compiler documentation for a complete description of the `mcc` command and its options.

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

<table>
<thead>
<tr>
<th>Action to Perform</th>
<th>mcc Option to Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a class encapsulating one or more M-files.</td>
<td>-W java:</td>
<td>Tells Java Builder to generate a Java component that contains a class that encapsulates the specified files.</td>
</tr>
</tbody>
</table>

## 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)

<table>
<thead>
<tr>
<th>Action to Perform</th>
<th>mcc Option to Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add additional classes to a Java component.</td>
<td>class{...}</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Syntax**

class{class_name:file1 [file2...fileN]}

**Example**

```bash
mcc -W 'java:com.mycompany.mycomponent,myclass'
   foo.m bar.m class{myclass2:foo2.m,bar2.m}
```

The example creates a Java component named mycomponent with two classes:

- myclass has methods foo and bar.
- myclass2 has methods foo2 and bar2.
Using the Command Line to Create Java Components (Continued)

<table>
<thead>
<tr>
<th>Action to Perform</th>
<th>mcc Option to Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplify the command-line input for components.</td>
<td>-B</td>
<td>Tells Java Builder to replace a specified file with the command-line information it contains.</td>
</tr>
</tbody>
</table>

**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

See “Using Bundle Files” for more information.
## Using the Command Line to Create Java Components (Continued)

<table>
<thead>
<tr>
<th>Action to Perform</th>
<th>mcc Option to Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control how each Java class uses the MCR.</td>
<td>-S</td>
<td>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.</td>
</tr>
<tr>
<td>Specify a directory for output</td>
<td>-d directoryname</td>
<td>Tells Java Builder to create a directory and copy the output files to it. (If you use <code>mcc</code> instead of the GUI, the <code>project_directory\src</code> and <code>project_directory\distrib</code> directories are not automatically created.)</td>
</tr>
</tbody>
</table>

### Example

```bash
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`. 

1-10
**Note** All of these command-line examples produce the following files:

- `mycomponent.jar` (component jar file)
- `mycomponent.ctf` (component ctf 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.
Developing an Application

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 was created in the last step in “Creating a Java Component” on page 1-5.

   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.

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

   The first step is not necessary if you are developing your application on the same machine where you created the Java component.

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

3 Import the MATLAB libraries and the component classes into your code with the Java `import` function. For example:

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

4 Use the new function in your Java code to create an instance of each class you want to use in the application.

5 Call the class methods as you would do with any Java class.

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 MArray classes in the package com.mathworks.toolbox.javabuilder. See Chapter 4, “Using MArray Classes” for an introduction to the classes and see com.mathworks.toolbox.javabuilder.MArray (available online only) for reference information for this class library.

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

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

To deploy your application you must make sure that the installer you create for the application takes care of supporting the components created by MATLAB Builder for Java. In general, this means that the MCR must be installed on the target machine, in addition to the application files.

Users must also set paths and environment variables correctly. See “Deploying to End Users” in the MATLAB Compiler documentation.
Example: Magic Square

This example shows you how to:

- Access the online examples provided with MATLAB Builder for Java.
- Use MATLAB Builder for Java to encapsulate a simple MATLAB function in a Java component.
- Use the MWArray class library to handle data conversion in a sample application.

About the Examples The examples for MATLAB Builder for Java are in matlabroot\toolbox\javabuilder\Examples.

The Magic Square example shows you how to create magicsquare, a Java component, which contains the magic class. The class encapsulates a MATLAB function, makesqr, which computes a magic square. It represents the magic square as a two-dimensional array.

The sample application, getmagic, does the following:

- Displays the array returned by the makesqr method
- Converts the array returned by makesqr to a native array and displays it

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

Magic Square Example: Step-by-Step Procedure

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

2 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:
This procedure assumes that you are working on Windows and your work directory is `D:}\Work`.

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

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

Here is the code for the `makesqr` function:

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

This code is already in your directory in `MagicSquareExample\MagicDemoComp\makesqr.m`.

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

```
deploytool
```

5 Create a new project

a. Click the New Deployment Project button in the Deployment Tool toolbar.

Alternatively you can select **File > New Deployment Project** from the MATLAB menu bar.

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

c. From the component list, select **Java** as the kind of component you want to create.

d. Click **Browse** to drill and select the location for your project.

e. Type the project name as `magicsquare` and click **OK**.

---

**Note** By default, the project name is also the component name.
When you create a new project, the Deployment Tool dialog box shows the folders that are part of the project. By default the top folder represents a Java class belonging to the component. By default the class name is the same as the project name.

**Note** When a new project first opens, the folders in the project are empty.

6 Enter the settings for the project.

   a. Right-click the top folder, which represents the Java class you are going to create, and select **Rename Class**.

   b. In the Rename Class dialog box, enter `magic` and click **OK**.

   c. Add the `makesqr.m` file to the project by dragging the file from the **Current Directory** pane in the MATLAB desktop to the `magic` folder in the Deployment Tool dialog box.

   d. Select **Generate Verbose Output**.

   e. Save the project by clicking **(Save)** in the Deployment Tool toolbar.

7 Build the project by clicking **(Build)** in the Deployment Tool toolbar.

   The build process begins, and a log of the build is created. The files that are needed for the component are generated in two newly created directories, `src` and `distrib`, in the project directory. A copy of the build log is placed in the `src` directory.

**Note** To create and build the `magicsquare` component without using the Deployment Tool to create a project, issue the following command at the MATLAB prompt:

```
mcc -W 'java:magicsquare,magic' makesqr.m
```

If you build your component using the `mcc` command, Java Builder does not create the `src` and `distrib` subdirectories.

See “What Happens in the Build Process?” on page 2-9 for more information.
8 Access the component in a Java application. The sample application for this example is in MagicSquareExample\MagicDemoJavaApp\getmagic.java.

getmagic.java

/* getmagic.java
 * This file is used as an example for the MATLAB
 * Builder for Java product.
 *
 * Copyright 2001-2006 The MathWorks, Inc.
 */

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

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

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

            /* Convert and print input value*/
            n = new MWNumericArray(Double.valueOf(args[0]),MWClassID.DOUBLE);
In this sample application, the following code imports the libraries and classes that are needed:

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

The following line instantiates the `magic` class:

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

The following line calls the `makesqr` method:

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

In this method call, the first input argument specifies the number of outputs the method is to return. This input is equivalent in value to the
MATLAB function, `nargout`, in the method being called. The second input argument is the input specified in the function declaration statement of the `makesqr` M-file.

**Note** The previous calling syntax is not the only way that you can call the `makesqr` method. When Java Builder encapsulates `makesqr.m`, it overloads the function, creating several signatures by which you can call the `makesqr` method. This lets you specify optional arguments when calling from Java. See “Understanding Function Signatures Generated by Java Builder” on page 2-6 for more details.

Note that the program uses a `try-catch` block to catch and handle any exceptions.

9 Compile the application.

Compile the `getmagic` application using the `javac-classpath` command at the DOS or UNIX command prompt.

The compile yields a `getmagic.class` file.

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

a. On Windows, execute the following command:

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

**Note** If you created the component using `mcc`, Java Builder does not create a \distrib directory to contain the .java file.
b. On UNIX, execute this command:

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

10 Run the application.

If you used the Deployment Tool to create it, you can use the following commands to run the `getmagic.class` file:

On Windows, type

```
java -classpath
  .;matlabroot\toolbox\javabuilder\jar\javabuilder.jar;
  .\distrib\magicsquare.jar
-Djava.library.path=matlabroot/bin\win32;\distrib
getmagic 5
```

On UNIX, type

```
java -classpath
  .:matlabroot/toolbox/javabuilder/jar/javabuilder.jar:
  ./distrib/magicsquare.jar
-Djava.library.path=matlabroot/bin/arch:./distrib
getmagic 5
```

**Note** If you set your system path to include a directory pointed by `java.library.path`, you do not need to use the `-D` option.

See “Settings for Environment Variables (Development Machine)” on page 6-2 and for more information about setting paths correctly.

**Note** The supported JRE version is 1.5.0. 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>`. 
When you run the program you pass an 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.

The getmagic program should display the following output:

```
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
```
Understanding the Magic Square Example

The Magic Square example shows the following aspects of writing an application using components created by MATLAB Builder for Java:

- “Importing Classes” on page 1-23
- “Creating an Instance of the Class” on page 1-23
- “Calling Class Methods from Java” on page 1-23

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:

```java
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:

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

Calling Class Methods from Java

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

```java
result = theMagic.makesqr(1, n);
```
where \( n \) is an instance of an \texttt{MWArray} class. See the following code fragment for the declaration of \( n \):

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

Understanding concepts needed to use MATLAB Builder for Java
Chapter 2, “Concepts”

Writing Java applications that can access Java methods that encapsulate M-code
Chapter 3, “Programming”

Using the MWArray API to handle data conversion
Chapter 4, “Using MWArray Classes”

Sample applications that access methods developed in MATLAB
Chapter 5, “Sample Applications (Java)”

Reference information about automatic data conversion rules
“Data Conversion Rules” on page 6-10
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.

**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.

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

- What Is a Project? (p. 2-3)
- How Does MATLAB Builder for Java Handle Data? (p. 2-4)
- How MATLAB Builder for Java uses the specifications in a project
- How MATLAB Builder for Java supports data conversion between Java types and MATLAB types
What Happens in the Build Process? (p. 2-9) Details about the process of building a Java component

What Happens in the Package Process? (p. 2-10) Details about the packaging process

How Does Component Deployment Work? (p. 2-11) Details about deploying to an end user
What Is a Project?

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” on page 2-3
- “Naming Conventions” on page 2-3

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.
How Does MATLAB Builder for Java Handle Data?

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” on page 2-4
- “Automatic Conversion to MATLAB Types” on page 2-5
- “Understanding Function Signatures Generated by Java Builder” on page 2-6
- “Returning Data from MATLAB to Java” on page 2-7

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 MWArray Classes”.

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 6-10 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 does not support optional function arguments in the way that MATLAB does with varargin and varargout. To support this feature of MATLAB, Java Builder always generates a basic set of overloaded Java methods when it encapsulates each MATLAB function. Each of these overloaded Java methods has the same name as the original M-function, but has a different number or type of arguments. Each overloaded function corresponds to one of the possible signatures in MATLAB.

In general, a function with \( N \) input arguments in MATLAB generates \( N+3 \) overloaded methods in Java. There is an overloaded method for each input argument \( (N) \). The additional three methods handle (1) the case of no input arguments and (2) the mlx function signature.

**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 2-5 for more details.
Understanding MATLAB Function Signatures
As background, recall that the generic MATLAB function has the following structure:

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

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

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

Each argument represents a MATLAB type. When you include the varargin or varargout argument, you can specify an arbitrary 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 a set of overloaded methods that implement the MATLAB functions. Each of these overloaded methods corresponds to a call to the generic MATLAB function for each combination of the possible number and type of input arguments.

In addition to these methods 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 the 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 6-14 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 MWCell1Array 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.
   - `mycomponentMCR.java`: Contains the CTF decryption keys and code to initialize the MCR for the component.

2. Compiles the Java code produced in step 1.


4. Creates a component technology file (.ctf) which contains encrypted MATLAB files generated by Java Builder.

5. 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 non-Windows platforms). The package contains at least the following:

- The Java Builder component
- The .ctf file for the component
- The MCR Installer (if the Install MCR option was selected when the component was built)

**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). That is, 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 as follows: 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.

  To accomplish this kind of deployment, create a package, using the GUI, as described in “Creating a Java Component” on page 1-5.

- 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 and its .ctf file.

For more information about the deployment process, see “Deploying to End Users” in the MATLAB Compiler documentation.
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:

- **Import Classes (p. 3-2)**: How to reference the classes
- **Creating an Instance of the Class (p. 3-3)**: Sample code for instantiating a class that encapsulates MATLAB code
- **Passing Arguments to and from Java (p. 3-6)**: How to match up data types between MATLAB and Java
- **Handling Errors (p. 3-19)**: How to handle an error generated by MATLAB
- **Managing Native Resources (p. 3-25)**: How to free memory used by the MWArray data conversion classes
- **Handling Data Conversion Between Java and MATLAB (p. 3-28)**: Call signatures for passing arguments and returning output

**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 Applications (Java)”.

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

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

- Import MATLAB libraries with the Java import function, for example:
  
  ```java
  import com.mathworks.toolbox.javabuilder.*;
  ```

- Import the component classes created by Java Builder, for example:
  
  ```java
  import componentname.*; or import componentname.classname;
  ```

**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). That is, 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 as follows: although the .jar file itself might be platform-independent, the .jar file is dependent on the .ctf file, which is not platform-independent.
Creating an Instance of the Class

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:

```java
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.

```java
/*
 * 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();
}

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 this 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:

```matlab
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 2001-2006 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

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” on page 3-6
- “Automatic Conversion to a MATLAB Type” on page 3-7
- “Specifying Optional Arguments” on page 3-9
- “Handling Return Values” on page 3-14

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 MWArray Classes”.

Code Fragment: Using MWNumericArray

The getmagic example shows 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.

```java
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:

```java
    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 6-10.

**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:

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

In this case, a Java double is passed as arg[0].
Here is another example:

```java
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`.

```java
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`.

```java
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:

```java
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:

```matlab
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 2001-2006 The MathWorks, Inc.
```

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

```java
/* 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:

```java
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:

```java
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:

```java
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:

```matlab
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 2001-2006 The MathWorks, Inc.

    for i=1:nargout
```
This function returns a list of random double vectors such that the length of the ith vector is equal to i. The MATLAB Compiler generates a Java interface to this function as follows:

```java
/* 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:

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

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

      for (int i = 0; i < y.length; i++)
         ret[i] = (double[])((MWArray)y[i]).getData();
   }
```
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 two basic ways to do this. You can 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.

### Code Fragment: Using Java Reflection

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

```java
public void getprimes(int n) throws MWException {
    myclass cls = null;
    Object[] y = null;
    return ret;
}
```
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 4-49) 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 4-40) 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.

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

    try
    {
        cls = new myclass();
        y = cls.myprimes(1, new Double((double)n));
```
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)
```java
{
    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();
}
```
Handling Errors

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:

```java
/* 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.

```java
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);
   }
```

3-20
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 3-25.

**Code Fragment: Handling an Exception in the Calling Function**

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

```java
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)
```
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. For information on the exceptions that can occur for each method of MWArray and its subclasses, see Using MWArray Classes.

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.

```java
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

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.

You can use any of the following techniques to free memory:

- “Using Garbage Collection Provided by the JVM” on page 3-25
- “Using the dispose Method” on page 3-26
- “Overriding the Object.Finalize Method” on page 3-27

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.

```java
/* 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:

```java
/* Allocate a huge array */
try
{
    int[] dims = {1000, 1000};
    MWNumericArray 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

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:

```java
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.

```java
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:

```java
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
Using MWArray Classes

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 (p. 4-2)  How to use the MWArray API to handle various kinds of data
Using Class Methods (p. 4-38)  How to use each class in the MWArray API
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.

<table>
<thead>
<tr>
<th>Method in MWArray Base Class</th>
<th>Override Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>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.</td>
</tr>
<tr>
<td>hashCode</td>
<td>Overrides Object.hashCode to allow MWArray to function properly with hash-based collections.</td>
</tr>
<tr>
<td>toString</td>
<td>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.</td>
</tr>
<tr>
<td>finalize</td>
<td>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.</td>
</tr>
</tbody>
</table>

Java Interfaces Implemented by MWArray

MWArray implements the standard Java interfaces shown in the following table.
Java Interfaces Implemented by MWArray

<table>
<thead>
<tr>
<th>Interface</th>
<th>Method in MWArray Base Class</th>
<th>Description of Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloneable</td>
<td>clone (public method)</td>
<td>Produces a new MWArray object that contains a deep copy of the underlying MATLAB array.</td>
</tr>
<tr>
<td>Comparable</td>
<td>compareTo (public method)</td>
<td>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.</td>
</tr>
<tr>
<td>Serializable</td>
<td>writeObject</td>
<td>Provides serialization support is as required by the Serializable interface.</td>
</tr>
<tr>
<td></td>
<td>readObject (private methods)</td>
<td></td>
</tr>
</tbody>
</table>

Additional MWArray Methods

MWArray

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWArray()</td>
<td>Constructs an empty array.</td>
</tr>
<tr>
<td>classID()</td>
<td>Returns the MATLAB type of the array.</td>
</tr>
<tr>
<td>Method</td>
<td>Usage</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>columnIndex()</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>dispose()</code></td>
<td>Frees the native resources associated with the underlying MATLAB array.</td>
</tr>
<tr>
<td><code>disposeArray(Object)</code></td>
<td>Calls <code>dispose</code> on all <code>MWArray</code> instances contained in the input.</td>
</tr>
<tr>
<td><code>get(int)</code></td>
<td>Returns the elements at the specified one-based offset.</td>
</tr>
<tr>
<td><code>get(int[])</code></td>
<td>Returns the elements at the specified one-based index array.</td>
</tr>
<tr>
<td><code>getData()</code></td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• If the underlying MATLAB array is complex, the real part is returned.</td>
</tr>
<tr>
<td></td>
<td>• If the underlying array is sparse, an array containing the nonzero elements is returned.</td>
</tr>
<tr>
<td></td>
<td>• If the underlying array is a cell or struct array, <code>toArray</code> is recursively called on each element.</td>
</tr>
<tr>
<td><code>getDimensions()</code></td>
<td>Returns an array of dimensions for the array.</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Tests if the array is empty.</td>
</tr>
<tr>
<td><code>isSparse()</code></td>
<td>Tests if the array is sparse.</td>
</tr>
<tr>
<td>Method</td>
<td>Usage</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>maximumNonZeros()</td>
<td>Returns the current allocated capacity of nonzero elements for a sparse array.</td>
</tr>
<tr>
<td>numberOfDimensions()</td>
<td>Returns the number of dimensions in the array.</td>
</tr>
<tr>
<td>numberOfElements()</td>
<td>Returns the number of elements in the array.</td>
</tr>
<tr>
<td>numberOfNonZeros()</td>
<td>Returns the current number of nonzero elements for a sparse array.</td>
</tr>
<tr>
<td>rowIndex()</td>
<td>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.</td>
</tr>
<tr>
<td>set(int, Object)</td>
<td>Replaces the element at the one-based index with the supplied value.</td>
</tr>
<tr>
<td>set(int[], Object)</td>
<td>Replaces the element at the one-based index array with the supplied value.</td>
</tr>
</tbody>
</table>
### 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.

### 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.
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWNumericArray()</td>
<td>Empty double array</td>
</tr>
<tr>
<td>MWNumericArray (MWClassID)</td>
<td>Empty array of type specified by MWClassID</td>
</tr>
<tr>
<td>MWNumericArray(type, MWClassID)</td>
<td>Real array of type specified by MWClassID</td>
</tr>
<tr>
<td>MWNumericArray(type)</td>
<td>Real array with type determined from default conversion rules</td>
</tr>
<tr>
<td>MWNumericArray(type, type, MWClassID)</td>
<td>Complex array of type specified by MWClassID</td>
</tr>
<tr>
<td>MWNumericArray(type, type)</td>
<td>Complex array with type determined from default conversion rules</td>
</tr>
</tbody>
</table>
**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:

```java
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:

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

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 $i$:

```java
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:
Using MWArray Classes

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.

<table>
<thead>
<tr>
<th>Factory Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>newInstance(int[], MWClassID, MWComplexity)</td>
<td>Numeric array of specified type and</td>
</tr>
<tr>
<td></td>
<td>complexity. Values are initialized to 0.</td>
</tr>
</tbody>
</table>
### Guidelines for Working with MWArray Classes

#### Factory Method

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newInstance(int[], Object, MWClassID)</code></td>
<td>Real numeric array of specified type. Values are initialized with supplied arrays.</td>
</tr>
<tr>
<td><code>newInstance(int[], Object, Object, MWClassID)</code></td>
<td>Complex numeric array of specified type. Values are initialized with supplied arrays.</td>
</tr>
<tr>
<td><code>newSparse(int[], int[], Object, int, int, int, MWClassID)</code></td>
<td>Real sparse numeric matrix of specified type, dimensions, and maximum nonzeros. Values are initialized with supplied row, column, and data arrays.</td>
</tr>
<tr>
<td><code>newSparse(int[], int[], Object, int, int, MWClassID)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>newSparse(int[], int[], Object, MWClassID)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>newSparse(int[], int[], Object, Object, int, int, MWClassID)</code></td>
<td>Complex sparse numeric matrix of specified type, dimensions, and maximum nonzeros. Values are initialized with supplied row, column, and data arrays.</td>
</tr>
<tr>
<td><code>newSparse(int[], int[], Object, Object, int, int, MWClassID)</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
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:

```java
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 =
```
Guidelines for Working with MWArray Classes

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:

\[
\begin{bmatrix}
2 & -1 & 0 & 0 \\
-1 & 2 & -1 & 0 \\
0 & -1 & 2 & -1
\end{bmatrix}
\]
Calling newSparse. The call to newSparse passes three arrays: an array of matrix data \( \mathbf{x} \), an array containing the row indices of \( \mathbf{x} \) (\( \text{rowindex} \)), and an array of column indices of \( \mathbf{x} \) (\( \text{colindex} \)). The number of rows (4) and columns (4) are also passed, as well as the type (\text{MWClassID.DOUBLE}):

\[
\begin{align*}
\text{double[]} \mathbf{x} &= \{ 2.0, -1.0, -1.0, 2.0, -1.0, \\
&\quad \quad \quad \quad -1.0, 2.0, -1.0, -1.0, 2.0 \}; \\
\text{int[]} \text{rowindex} &= \{1, 2, 1, 2, 3, 2, 3, 4, 3, 4\}; \\
\text{int[]} \text{colindex} &= \{1, 1, 2, 2, 2, 3, 3, 3, 4, 4\};
\end{align*}
\]

\( \text{MWNumericArray a = MWNumericArray.newSparse(rowindex, colindex, x, 4, 4, MWClassID.DOUBLE); } \)

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:

\( \text{MWNumericArray a = MWNumericArray.newSparse(rowindex, colindex, x, MWClassID.DOUBLE); } \)

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:

\[
\begin{align*}
\text{double}[][] \mathbf{x} &= \{\{ 2.0, -1.0, 0.0, 0.0\}, \\
&\quad \quad \quad \quad \{-1.0, 2.0, -1.0, 0.0\}, \\
&\quad \quad \quad \quad \{ 0.0, -1.0, 2.0, -1.0\}, \\
&\quad \quad \quad \quad \{ 0.0, 0.0, -1.0, 2.0 \}\}; \\
\text{MWNumericArray a = MWNumericArray.newSparse(x, MWClassID.DOUBLE); }
\end{align*}
\]
Guidelines for Working with MWArray Classes

**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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_type(int)</code></td>
<td>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).</td>
</tr>
<tr>
<td><code>get_type(int[])</code></td>
<td>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).</td>
</tr>
<tr>
<td><code>get_imag_type(int)</code></td>
<td>Returns the imaginary part of the element at the one-based index. Return value is of the type specified (e.g., getImgDouble returns a double).</td>
</tr>
<tr>
<td><code>get_imag_type(int[])</code></td>
<td>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).</td>
</tr>
<tr>
<td><code>set(int, type)</code></td>
<td>Replaces the real part of the element at the one-based index with the supplied value</td>
</tr>
<tr>
<td><code>set(int[], type)</code></td>
<td>Replaces the real part of the element at the one-based index array with the supplied value</td>
</tr>
</tbody>
</table>
Using MWArray Classes

### Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setImag(int, type)</code></td>
<td>Replaces the imaginary part of the element at the one-based index with the supplied value</td>
</tr>
<tr>
<td><code>setImag(int[], type)</code></td>
<td>Replaces the imaginary part of the element at the one-based index array with the supplied value</td>
</tr>
</tbody>
</table>

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:

```java
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:

```java
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:

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

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

```java
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.println();
    }
}
```
The next example sums the real parts of all the elements in a numeric array and returns the result as a double value:

```java
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[][]:

```java
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.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWLogicalArray()</td>
<td>Empty logical array</td>
</tr>
<tr>
<td>MWLogicalArray(type)</td>
<td>Logical array with values initialized with supplied data</td>
</tr>
</tbody>
</table>

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:

```java
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:

```java
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:

```java
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}}
};
```

```java
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.

<table>
<thead>
<tr>
<th>Factory Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>newInstance(int[])</td>
<td>New logical array. Values are initialized to false.</td>
</tr>
</tbody>
</table>
Using MWArray Classes

<table>
<thead>
<tr>
<th>Factory Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>newInstance(int[], Object)</td>
<td>New logical array. Values are initialized with supported data.</td>
</tr>
<tr>
<td>newSparse(int[], int[], Object, int, int, int)</td>
<td>Sparse logical matrix of specified dimensions and maximum nonzeros. Values are initialized with supplied row, column, and data arrays.</td>
</tr>
<tr>
<td>newSparse(int[], int[], Object, int, int)</td>
<td>Sparse logical matrix of specified dimensions. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros are computed from input data.</td>
</tr>
<tr>
<td>newSparse(int[], int[], Object)</td>
<td>Sparse logical matrix. Values are initialized with supplied row, column, and data arrays. Maximum nonzeros and dimensions are computed from input data.</td>
</tr>
<tr>
<td>newSparse(Object)</td>
<td>Sparse logical matrix. Values are initialized from supplied full matrix.</td>
</tr>
</tbody>
</table>

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:

```java
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};
```
Guidelines for Working with MWArray Classes

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get(int)</code></td>
<td>Returns the element at the one-based index as type <code>java.lang.Boolean</code> (inherited from MWArray).</td>
</tr>
<tr>
<td><code>get(int[])</code></td>
<td>Returns the element at the one-based index array as type <code>java.lang.Boolean</code> (inherited from MWArray).</td>
</tr>
<tr>
<td><code>getBoolean(int)</code></td>
<td>Returns the element at the one-based index as type <code>boolean</code>.</td>
</tr>
<tr>
<td><code>getBoolean(int[])</code></td>
<td>Returns the element at the one-based index array as type <code>boolean</code>.</td>
</tr>
<tr>
<td><code>set(int, Object)</code></td>
<td>Replaces the element at the one-based index with the supplied value (inherited from MWArray).</td>
</tr>
<tr>
<td><code>set(int[], Object)</code></td>
<td>Replaces the element at the one-based index array with the supplied value (inherited from MWArray).</td>
</tr>
<tr>
<td><code>set(int, boolean)</code></td>
<td>Replaces the element at the one-based index with the supplied boolean value.</td>
</tr>
<tr>
<td><code>set(int[], boolean)</code></td>
<td>Replaces element at the one-based index array with the supplied boolean value.</td>
</tr>
</tbody>
</table>

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`, `object`.

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

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWCharArray()</td>
<td>Empty char array</td>
</tr>
<tr>
<td>MWCharArray(type)</td>
<td>char array with values initialized with supplied data</td>
</tr>
</tbody>
</table>

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:

```java
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:

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>Factory Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>newInstance(int[])</td>
<td>New char array. Values are initialized to Null.</td>
</tr>
<tr>
<td>newInstance(int[] Object)</td>
<td>New char array. Values are initialized with supported data.</td>
</tr>
</tbody>
</table>

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:

```java
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'),
};
```
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:

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(int)</td>
<td>Returns the element at the one-based index as type java.lang.Character (inherited from MWArray).</td>
</tr>
<tr>
<td>get(int[])</td>
<td>Returns the element at the one-based index array as type java.lang.Character (inherited from MWArray).</td>
</tr>
<tr>
<td>getChar(int)</td>
<td>Returns the element at the one-based index as type char.</td>
</tr>
<tr>
<td>getChar(int[])</td>
<td>Returns the element at the one-based index array as type char.</td>
</tr>
</tbody>
</table>
### Method Usage

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>set(int, Object)</td>
<td>Replaces the element at the one-based index with the supplied value (inherited from MWArray).</td>
</tr>
<tr>
<td>set(int[], Object)</td>
<td>Replaces the element at the one-based index array with the supplied value (inherited from MWArray).</td>
</tr>
<tr>
<td>set(int, char)</td>
<td>Replaces the element at the one-based index with the supplied char value.</td>
</tr>
<tr>
<td>set(int[], char)</td>
<td>Replaces element at the one-based index array with the supplied char value.</td>
</tr>
</tbody>
</table>

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:

```java
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:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWCellArray()</td>
<td>Empty cell array.</td>
</tr>
<tr>
<td>MWCellArray(int[])</td>
<td>New cell array with specified dimensions. All cells are initialized to empty.</td>
</tr>
<tr>
<td>MWCellArray(gint, int)</td>
<td>New cell matrix with specified number of rows and columns.</td>
</tr>
</tbody>
</table>

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:

```java
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.

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(int)</td>
<td>Returns the element at the one-based index as a Java array (inherited from MWArray).</td>
</tr>
<tr>
<td>get(int)</td>
<td>Returns the element at the one-based index array as a Java array (inherited from MWArray).</td>
</tr>
<tr>
<td>getCell(int)</td>
<td>Returns the element at the one-based index as an MWArray instance.</td>
</tr>
<tr>
<td>getCell(int[])</td>
<td>Returns the element at the one-based index array as an MWArray instance.</td>
</tr>
</tbody>
</table>
## Guidelines for Working with MWArray Classes

### Method Usage

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set(int, Object)</code></td>
<td>Replaces the element at the one-based index with the supplied value (inherited from <code>MWArray</code>).</td>
</tr>
<tr>
<td><code>set(int[], Object)</code></td>
<td>Replaces the element at the one-based index array with the supplied value (inherited from <code>MWArray</code>).</td>
</tr>
</tbody>
</table>

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:

```java
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;
        }
    }
    catch (Exception e)
    {
        // Handle exception
    }
}
```
Using MWArray Classes

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:

```java
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.

```java
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.
Using MWArray Classes

Using Class Methods

Use MWArray classes to construct data types as needed in your Java code. You can use the following classes to create objects that correspond to the basic MATLAB data types:

- Using MWArray
- Using MWNumericArray
- Using MWLogicalArray
- Using MWCharArray
- Using MWStructArray
- Using MWCellArray
- MWClassID
- Using MWComplexity

Using MWArray

This section covers topics on MWArray:

- “Constructing an MWArray” on page 4-38
- “Methods to Create and Destroy an MWArray” on page 4-39
- “Methods to Return Information About an MWArray” on page 4-40
- “Methods to Get and Set Data in the MWArray” on page 4-44
- “Methods to Copy, Convert, and Compare MWArrays” on page 4-49
- “Methods to Use on Sparse MWArrays” on page 4-54

Constructing an MWArray

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

```java
MWArray()
```

The type given to this object is MWClassID.UNKNOWN.
**Example.** Construct an empty MWArray object:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“dispose” on page 4-39</td>
<td>Frees the native MATLAB array contained by this array.</td>
</tr>
<tr>
<td>“disposeArray” on page 4-40</td>
<td>Frees all native MATLAB arrays contained in the input object.</td>
</tr>
</tbody>
</table>

**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:

```java
public void dispose()
```

**Input Parameters**

None

**Example — Constructing an MWArray Object**

Construct and then destroy an MWArray object:

```java
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:

```java
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:

```java
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“classID” on page 4-41</td>
<td>Returns the MATLAB type of the array.</td>
</tr>
<tr>
<td>“getDimensions” on page 4-42</td>
<td>Returns the size of each dimension of the array.</td>
</tr>
<tr>
<td>“isEmpty” on page 4-42</td>
<td>Tests if the array has no elements.</td>
</tr>
<tr>
<td>“numberOfDimensions” on page 4-43</td>
<td>Returns the number of dimensions of the array.</td>
</tr>
<tr>
<td>“numberOfElements” on page 4-43</td>
<td>Returns the total number of elements in the array.</td>
</tr>
</tbody>
</table>

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

```java
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:

```java
public MWClassID classID()
```

**Input Parameters**

None

**Example — Getting the Class ID of an MWArray**

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

```java
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:

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

Input Parameters

None

Example — Getting Array Dimensions of an MWArray

```java
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:

```java
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:

```java
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:

```java
public int numberOfDimensions()
```

**Input Parameters**

None

Example — Getting the Number of Dimensions of an MWArray

Display the number of dimensions for array object A:

```java
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:

```java
public int numberOfElements()
```

**Input Parameters**

None

**Example — Getting the Number of MWArray Elements**

Display the number of elements in array object A:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“get” on page 4-45</td>
<td>Returns the element at the specified one-based offset or index array as an Object.</td>
</tr>
<tr>
<td>“getData” on page 4-46</td>
<td>Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array. The array is in column-wise order.</td>
</tr>
<tr>
<td>“set” on page 4-47</td>
<td>Replaces the element at the specified one-based offset, or index array, in this array with the specified element.</td>
</tr>
<tr>
<td>“toArray” on page 4-48</td>
<td>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.</td>
</tr>
</tbody>
</table>
**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:

```java
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

Retrieve element (2, 4) from array object A:

```java
int[] index = {2, 4};

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 10

**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:

```java
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`:

```java
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:

```java
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:

```java
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:

```java
    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:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“clone” on page 4-50</td>
<td>Creates and returns a deep copy of this array.</td>
</tr>
<tr>
<td>“compareTo” on page 4-51</td>
<td>Compares this array with the specified array for order.</td>
</tr>
<tr>
<td>“equals” on page 4-52</td>
<td>Indicates whether some other array is equal to this one.</td>
</tr>
<tr>
<td>“hashCode” on page 4-52</td>
<td>Returns a hash code value for the array.</td>
</tr>
</tbody>
</table>
Using MWArray Classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“sharedCopy” on page 4-53</td>
<td>Creates and returns a shared copy of this array.</td>
</tr>
<tr>
<td>“toString” on page 4-54</td>
<td>Returns a string representation of the array.</td>
</tr>
</tbody>
</table>

**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:

```java
public Object clone()
```

**Input Parameters**

None

**Exceptions**

The clone method throws the following exception:

```java
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:

```java
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:

```java
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:

```java
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:

```java
public int hashCode()
```

**Input Parameters**

**None**
Example — Getting an MWArray Hash Code

Obtain the hash code for MWArray object A:

```java
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:

```java
public Object sharedCopy()
```

Input Parameters

None

Example — Making a Shared Copy of an MWArray

Create a shared copy of MWArray object A:

```java
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:

```java
public java.lang.String toString()
```

Input Parameters

None

Example — Converting an MWArray to a String

Display the contents of MWArray object A:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“isSparse” on page 4-55</td>
<td>Tests whether the array is sparse.</td>
</tr>
<tr>
<td>“columnIndex” on page 4-56</td>
<td>Returns an array containing the column index of each nonzero element in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“rowIndex” on page 4-57</td>
<td>Returns an array containing the row index of each nonzero element in the underlying MATLAB array.</td>
</tr>
</tbody>
</table>
Using Class Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| “maximumNonZeros”           | Returns the allocated capacity of a sparse array. If the underlying array is
| on page 4-57                | nonsparse, this method returns the same value as numberOfElements().        |
| “numberOfNonZeros”          | Returns the number of nonzero elements in a sparse array. If the underlying  |
| on page 4-58                | 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 4-66 method of MWNumericArray:

```java
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:

```java
public boolean isSparse()
```
Input Parameters

None

Example — Testing an MWArray for Sparseness

Test the MWArray object A created previously for sparseness:

```java
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:

```java
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:

```java
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 non-sparse, 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 MWAarray

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 MWAarray

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 4-59
• “Methods to Create and Destroy an MWNumericArray” on page 4-63
• “Methods to Get and Set the Real Parts of an MWNumericArray” on page 4-75
• “Methods to Get and Set the Imaginary Parts of an MWNumericArray” on page 4-79
• “Methods to Copy, Convert, and Compare MWNumericArrays” on page 4-87
• “Methods to Use on Sparse MWNumericArrays” on page 4-90
• “Methods to Return Special Constant Values” on page 4-90

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 4-59
• “Constructing a Real or Complex Numeric Scalar” on page 4-60
• “Constructing a Real or Complex Numeric Array” on page 4-61

In addition to using the MWNumericArray constructor, you can also use “newSparse” on page 4-66 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:

```java
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:

```java
MWNumericArray(javaType realValue)
```

Or use this syntax to create a complex scalar `MWNumericArray` from a primitive Java type:

```java
MWNumericArray(javaType realValue, javaType imagValue)
```

The class ID for the returned `MWNumericArray` is shown in the following table:

<table>
<thead>
<tr>
<th>javatype Input</th>
<th>Class ID of MWNumericArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>MWClassID.DOUBLE</td>
</tr>
<tr>
<td>float</td>
<td>MWClassID.SINGLE</td>
</tr>
<tr>
<td>long</td>
<td>MWClassID.INT64</td>
</tr>
<tr>
<td>int</td>
<td>MWClassID.INT32</td>
</tr>
<tr>
<td>short</td>
<td>MWClassID.INT16</td>
</tr>
<tr>
<td>byte</td>
<td>MWClassID.INT8</td>
</tr>
</tbody>
</table>

**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`:

```java
double AReal = 24;
MWNumericArray A = new MWNumericArray(AReal, MWClassID.INT16);
System.out.println("Array A of type " + A.classID() + " = 
" + 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:

```java
double AReal = 24;
double AImag = 5;
MWNumericArray A = new MWNumericArray(AReal, AImag);
System.out.println("Array A of type " + A.classID() + " = 
" + 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:

```java
MWNumericArray(javatype realValue, MWClassID classid)
```

Or use this syntax to create a complex nonscalar `MWNumericArray` from a primitive Java type:

```java
MWNumericArray(javatype realValue, javatype imagValue,
```
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:

```java
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 = \
+ 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:

```java
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() + " = 
" + 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“newInstance” on page 4-63</td>
<td>Constructs an array with the specified dimensions and complexity.</td>
</tr>
<tr>
<td>“newSparse” on page 4-66</td>
<td>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.</td>
</tr>
<tr>
<td>“dispose” on page 4-71</td>
<td>Frees the native MATLAB array contained by this array.</td>
</tr>
<tr>
<td>“disposeArray” on page 4-71</td>
<td>Frees all native MATLAB arrays contained in the input object.</td>
</tr>
</tbody>
</table>

**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[]`
Using Class Methods

- 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.

```java
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:

\[
\begin{array}{ccccccc}
A = & 1 & 2 & 3 & 4 & 5 & 6 \\
& 7 & 8 & 9 & 10 & 11 & 12 \\
& 13 & 14 & 15 & 16 & 17 & 18 \\
\end{array}
\]

**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

\[
\text{newSparse}(\text{int rows, int cols, int nzmax, MWClassID classid, MWComplexity cmplx})
\]

Constructing a Sparse Matrix of Real Numbers

To construct a real sparse array from an existing nonsparse two-dimensional array, use

\[
\text{newSparse}(\text{Object realData, MWClassID classid})
\]

To construct and initialize a new real sparse array, use

\[
\text{newSparse}(\text{int[] rowindex, int[] colindex, Object realData, MWClassID classid})
\]

To construct and initialize a new real sparse array, specifying its dimensions, use

\[
\text{newSparse}(\text{int[] rowindex, int[] colindex, Object realData, int rows, int cols, MWClassID classid})
\]

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 \( \text{nzmax} \) nonzeros:

\[
S(\text{rowIndex}(k), \text{columnIndex}(k)) = \text{realData}(k) + \text{imagData}(k) \cdot i
\]

If you assign multiple values to a single \( \text{rowIndex} \) and \( \text{colIndex} \) pair, then the element at that index is assigned the sum of these values.

\( \text{rows and cols} \)

Number of rows and columns in the matrix

\( \text{nzmax} \)

Maximum number of nonzero elements

\( \text{classID} \)

\( \text{MWClassID} \) representing the MATLAB type of the array. The only \( \text{classID} \) currently supported is \( \text{MWClassID.DOUBLE} \).

Valid types for the \( \text{realData} \) and \( \text{imagData} \) parameters are as follows:

- \( \text{double[]} \)
- \( \text{float[]} \)
- \( \text{long[]} \)
- \( \text{int[]} \)
- \( \text{short[]} \)
- \( \text{byte[]} \)
- \( \text{boolean[]} \)
- One-dimensional arrays of any subclass of \( \text{java.lang.Number} \)
- One-dimensional arrays of \( \text{java.lang.Boolean} \)
- One-dimensional arrays of \( \text{java.lang.String} \)
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:

```java
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
```
Example — Using newSparse with Row and Column Indices

Construct a sparse MWNumericArray from vector Adata:

```java
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.

```java
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);
```

```
4-70
```
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“classID” on page 4-72</td>
<td>Returns the MATLAB type of this array.</td>
</tr>
<tr>
<td>“complexity” on page 4-72</td>
<td>Returns the complexity of this array.</td>
</tr>
<tr>
<td>“getDimensions” on page 4-72</td>
<td>Returns an array containing the size of each dimension of this array.</td>
</tr>
<tr>
<td>“isEmpty” on page 4-72</td>
<td>Tests whether the array has no elements.</td>
</tr>
<tr>
<td>“isFinite” on page 4-72</td>
<td>Tests for finiteness in a machine-independent manner.</td>
</tr>
<tr>
<td>“isInf” on page 4-73</td>
<td>Tests for infinity in a machine-independent manner.</td>
</tr>
<tr>
<td>“isNaN” on page 4-74</td>
<td>Tests for NaN (not a number) in a machine-independent manner.</td>
</tr>
<tr>
<td>“numberOfDimensions” on page 4-75</td>
<td>Returns the number of dimensions of this array.</td>
</tr>
<tr>
<td>“numberOfElements” on page 4-75</td>
<td>Returns the total number of elements in this array.</td>
</tr>
</tbody>
</table>
**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

```java
public MWComplexity complexity()
```

**Input Parameters**

None

**Example — Testing for a Complex Array**

Determine whether matrix A is real or complex:

```java
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:

```java
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:

```java
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

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“get” on page 4-76</td>
<td>Returns the element at the specified offset as an Object.</td>
</tr>
<tr>
<td>“getData” on page 4-76</td>
<td>Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“getDouble” on page 4-76</td>
<td>Returns the real part at the specified offset as a double value.</td>
</tr>
<tr>
<td>“getFloat” on page 4-77</td>
<td>Returns the real part at the specified offset as a float value.</td>
</tr>
<tr>
<td>“getLong” on page 4-77</td>
<td>Returns the real part at the specified offset as a long value.</td>
</tr>
<tr>
<td>“getInt” on page 4-77</td>
<td>Returns the real part at the specified offset as an int value.</td>
</tr>
<tr>
<td>“getShort” on page 4-77</td>
<td>Returns the real part at the specified offset as a short value.</td>
</tr>
<tr>
<td>“getByte” on page 4-78</td>
<td>Returns the real part at the specified offset as a byte value.</td>
</tr>
<tr>
<td>“set” on page 4-79</td>
<td>Replaces the real part at the specified offset with the specified value.</td>
</tr>
<tr>
<td>“toArray” on page 4-79</td>
<td>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.</td>
</tr>
</tbody>
</table>
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:

```java
public type getType(int index)
pUBLIC type getType(int[] index)
```

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

```java
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[]`:

```java
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[]`:

```java
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[]`:

```java
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[]`:

```java
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[]`:

```java
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[]`:

```java
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:

```java
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)`:  

```java
int[] idx = {2, 3};

System.out.println("A(2, 3) is " + A.get(idx).toString());
System.out.println("\n");
```
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`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“getImag” on page 4-81</td>
<td>Returns the imaginary part at the specified index array in this array.</td>
</tr>
<tr>
<td>“getImagData” on page 4-82</td>
<td>Returns a one-dimensional array containing a copy of the imaginary data in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“getImagDouble” on page 4-83</td>
<td>Returns the imaginary part at the specified offset as a double value.</td>
</tr>
</tbody>
</table>
Using MWArray Classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“getImagFloat” on page 4-84</td>
<td>Returns the imaginary part at the specified offset as a float value.</td>
</tr>
<tr>
<td>“getImagLong” on page 4-84</td>
<td>Returns the imaginary part at the specified offset as a long value.</td>
</tr>
<tr>
<td>“getImagInt” on page 4-84</td>
<td>Returns the imaginary part at the specified offset as an int value.</td>
</tr>
<tr>
<td>“getImagShort” on page 4-84</td>
<td>Returns the imaginary part at the specified offset as a short value.</td>
</tr>
<tr>
<td>“getImagByte” on page 4-85</td>
<td>Returns the imaginary part at the specified offset as a byte value.</td>
</tr>
<tr>
<td>“setImag” on page 4-85</td>
<td>Replaces the imaginary part at the specified index array in this array with the specified double value.</td>
</tr>
<tr>
<td>“toImagArray” on page 4-86</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 \( \text{index} \) is of type \( \text{int} \), the valid range for \( \text{index} \) is \( 1 \leq \text{index} \leq N \), where \( N \) is the total number of elements in the array.

In the case where \( \text{index} \) is of type \( \text{int[]} \), each element of the \( \text{index} \) vector is an index along one dimension of the \text{MWNumericArray} object. The valid range for any index is \( 1 \leq \text{index}[i] \leq N[i] \), where \( N[i] \) is the size of the \( i \)th dimension.

**Exceptions.** These methods throw the following exception:

\text{IndexOutOfBoundsException}

The specified \( \text{index} \) parameter is invalid.

**getImag.** This method returns the imaginary part of the \text{MWNumericArray} element located at the specified one-based index or index array. The type of the return value is \text{Object}.

Use either of the following prototypes for the \text{getImag} method, where \( \text{index} \) can be of type \( \text{int} \) or \( \text{int[]} \):

\begin{verbatim}
public Object getImag(int index)
public Object getImag(int[] index)
\end{verbatim}

**Example — Getting the Real and Imaginary Parts of an Array**

Start by creating a two-dimensional array of complex values:

\begin{verbatim}
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 =");
\end{verbatim}
System.out.println(A.toString());

Here is the complex array that is displayed:

\[
\begin{array}{ccc}
2.0000 + 6.0000i & 3.0000 + 5.0000i & 4.0000 + 14.0000i \\
8.0000 + 7.0000i & 9.0000 + 1.0000i & 10.0000 + 23.0000i \\
14.0000 + 1.0000i & 15.0000 + 1.0000i & 16.0000 + 9.0000i
\end{array}
\]

Now, use get and getImag to read the real and imaginary parts of the element at index (2, 3):

```java
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:

```java
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 4-81, get the entire contents of the complex array:

```java
int[] index = {2, 3};
double[] x;
```
Using Class Methods

```java
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 4-81, 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[]:

```java
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[]:

```java
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[]:

```java
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[]:

```java
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[]:

```java
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[]:

```java
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 4-81, get and display a copy of the imaginary part of that array:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“clone” on page 4-87</td>
<td>Creates and returns a deep copy of this array.</td>
</tr>
<tr>
<td>“compareTo” on page 4-88</td>
<td>Compares this array with the specified array for order.</td>
</tr>
<tr>
<td>“equals” on page 4-89</td>
<td>Indicates whether some other array is equal to this one.</td>
</tr>
<tr>
<td>“hashCode” on page 4-89</td>
<td>Returns a hash code value for the array.</td>
</tr>
<tr>
<td>“sharedCopy” on page 4-89</td>
<td>Creates and returns a shared copy of this array.</td>
</tr>
<tr>
<td>“toString” on page 4-89</td>
<td>Returns a string representation of the array.</td>
</tr>
</tbody>
</table>

**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

```java
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:

```java
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:

```java
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:

```java
public Object sharedCopy()
```

Input Parameters

None

Example — Making a Shared Copy of a Numeric Array Object

Create a shared copy of MWArray object A:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“newSparse” on page 4-66</td>
<td>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.</td>
</tr>
<tr>
<td>“isSparse” on page 4-55</td>
<td>Tests whether the array is sparse.</td>
</tr>
<tr>
<td>“columnIndex” on page 4-56</td>
<td>Returns an array containing the column index of each nonzero element in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“rowIndex” on page 4-57</td>
<td>Returns an array containing the row index of each nonzero element in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“maximumNonZeros” on page 4-57</td>
<td>Returns the allocated capacity of a sparse array. If the underlying array is nonsparse, this method returns the same value as numberOfElements().</td>
</tr>
<tr>
<td>“numberOfNonZeros” on page 4-58</td>
<td>Returns the number of nonzero elements in a sparse array. If the underlying array is nonsparse, this method returns the same value as numberOfElements().</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th><strong>Method</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>“getEps” on page 4-91</td>
<td>Get the value represented by EPS (floating-point relative accuracy) in MATLAB.</td>
</tr>
<tr>
<td>“getInf” on page 4-91</td>
<td>Get the value represented by INF (infinity) in MATLAB.</td>
</tr>
<tr>
<td>“getNaN” on page 4-92</td>
<td>Get the value represented by NaN (Not a Number) in MATLAB.</td>
</tr>
</tbody>
</table>

**getEps.** This method returns the MATLAB concept of EPS, which stands for the floating-point relative accuracy.

The prototype for the getEps method is

```java
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

```java
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

\[
\text{public static double getNaN()}
\]

Input Parameters
None

Exceptions
None

**Using MWLogicalArray**
This section covers the following topics:

- “Constructing an MWLogicalArray” on page 4-92
- “Methods to Create and Destroy an MWLogicalArray” on page 4-93
- “Methods to Return Information About an MWLogicalArray” on page 4-98
- “Methods to Get and Set Data in an MWLogicalArray” on page 4-100
- “Methods to Copy, Convert, and Compare MWLogicalArrays” on page 4-104
- “Methods to Use on Sparse MWLogicalArrays” on page 4-107

**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
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:

\[
\text{MWLogicalArray(javatype array)}
\]

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

```java
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 4-94 | Constructs a logical array with the specified dimensions.
“newSparse” on page 4-95 | Constructs a sparse logical matrix from the supplied full matrix.
“dispose” on page 4-98 | Frees the native MATLAB array contained by this array.
“disposeArray” on page 4-98 | 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.

```java
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 96. Any parameters not specified are given their default values.
To construct a sparse logical matrix with no nonzero elements, use the
following:

public static MWWLogicalArray newSparse(int rows, int cols,
        int nzmax)

To construct a sparse logical matrix from a supplied full matrix, use the
following:

public static MWWLogicalArray newSparse(Object data)

To specify what data is assigned to each element, use the following:

public static MWWLogicalArray newSparse(int[] rowindex,
        int[] colindex, Object data)

To specify the number of rows and columns in the array, use the following:

public static MWWLogicalArray 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 MWWLogicalArray 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:
\[ S(\text{rowIndex}(k), \text{colIndex}(k)) = \text{data}(k) \]

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:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“classID” on page 4-99</td>
<td>Returns the MATLAB type of this array.</td>
</tr>
</tbody>
</table>
Using Class Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“getDimensions” on page 4-100</td>
<td>Returns an array containing the size of each dimension of this array.</td>
</tr>
<tr>
<td>“isEmpty” on page 4-100</td>
<td>Tests whether the array has no elements.</td>
</tr>
<tr>
<td>“numberOfDimensions” on page 4-100</td>
<td>Returns the number of dimensions of this array.</td>
</tr>
<tr>
<td>“numberOfElements” on page 4-100</td>
<td>Returns the total number of elements in this array.</td>
</tr>
</tbody>
</table>

**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:**

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“get” on page 4-100</td>
<td>Returns the element at the specified offset as an Object.</td>
</tr>
<tr>
<td>“getData” on page 4-100</td>
<td>Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“getBoolean” on page 4-101</td>
<td>Returns the boolean at the specified one-based offset.</td>
</tr>
<tr>
<td>“set” on page 4-102</td>
<td>Replaces the element at the specified one-based offset in this array with the specified element.</td>
</tr>
<tr>
<td>“toArray” on page 4-104</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 ith 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

```java
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 \(1 \leq index \leq N\), 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):

```java
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);

A.set(index, true);
```

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“clone” on page 4-104</td>
<td>Creates and returns a deep copy of this array.</td>
</tr>
<tr>
<td>“compareTo” on page 4-105</td>
<td>Compares this array with the specified array for order.</td>
</tr>
<tr>
<td>“equals” on page 4-105</td>
<td>Indicates whether some other array is equal to this one.</td>
</tr>
<tr>
<td>“hashCode” on page 4-106</td>
<td>Returns a hash code value for the array.</td>
</tr>
<tr>
<td>“sharedCopy” on page 4-106</td>
<td>Creates and returns a shared copy of this array.</td>
</tr>
<tr>
<td>“toString” on page 4-106</td>
<td>Returns a string representation of the array.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“isSparse” on page 4-55</td>
<td>Tests whether the array is sparse.</td>
</tr>
<tr>
<td>“columnIndex” on page 4-56</td>
<td>Returns an array containing the column index of each nonzero element in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“rowIndex” on page 4-57</td>
<td>Returns an array containing the row index of each nonzero element in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“maximumNonZeros” on page 4-57</td>
<td>Returns the allocated capacity of a sparse array. If the underlying array is nonsparse, this method returns the same value as numberOfElements().</td>
</tr>
<tr>
<td>“numberOfNonZeros” on page 4-58</td>
<td>Returns the number of nonzero elements in a sparse array. If the underlying array is nonsparse, this method returns the same value as numberOfElements().</td>
</tr>
</tbody>
</table>

MWLogicalArray inherits all the above methods from the MWArray class.

Using MWCharArray
This section covers the following topics:

- “Constructing an MWCharArray” on page 4-108
- “Methods to Create and Destroy an MWCharArray” on page 4-109
- “Methods to Return Information About an MWCharArray” on page 4-111
- “Methods to Get and Set Data in the MWCharArray” on page 4-112
- “Methods to Copy, Convert, and Compare MWCharArrays” on page 4-116
**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

```java
MWCharArray()
```

To construct a MWCharArray object from a primitive Java char scalar, use the following prototype:

```java
MWCharArray(char value)
```

To construct a MWCharArray object from a Java Object, use

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;newInstance&quot;</td>
<td>Constructs a char array with the specified dimensions.</td>
</tr>
<tr>
<td>on page 4-109</td>
<td></td>
</tr>
<tr>
<td>&quot;dispose&quot;</td>
<td>Frees the native MATLAB array contained by this array.</td>
</tr>
<tr>
<td>on page 4-110</td>
<td></td>
</tr>
<tr>
<td>&quot;disposeArray&quot;</td>
<td>Frees all native MATLAB arrays contained in the input object.</td>
</tr>
<tr>
<td>on page 4-111</td>
<td></td>
</tr>
</tbody>
</table>

**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 + \"\\\n");
```

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“classID” on page 4-111</td>
<td>Returns the MATLAB type of this array.</td>
</tr>
<tr>
<td>“getDimensions” on page 4-112</td>
<td>Returns an array containing the size of each dimension of this array.</td>
</tr>
<tr>
<td>“isEmpty” on page 4-112</td>
<td>Tests whether the array has no elements.</td>
</tr>
<tr>
<td>“numberOfDimensions” on page 4-112</td>
<td>Returns the number of dimensions of this array.</td>
</tr>
<tr>
<td>“numberOfElements” on page 4-112</td>
<td>Returns the total number of elements in this array.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“get” on page 4-112</td>
<td>Returns the element at the specified offset as an Object.</td>
</tr>
<tr>
<td>“getData” on page 4-112</td>
<td>Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“getChar” on page 4-113</td>
<td>Returns the character at the specified one-based offset.</td>
</tr>
<tr>
<td>“set” on page 4-114</td>
<td>Replaces the element at the specified one-based offset in this array with the specified element.</td>
</tr>
<tr>
<td>“toArray” on page 4-116</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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

```java
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 ith 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:

```java
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

```java
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 $1 \leq index[i] \leq N[i]$, 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 Character Array

Display a phrase stored in `MWCharArray` object $A$, change one of the characters, and then display the modified phrase:

```java
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 ...");
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 MWCharArray**

Use these methods to copy, convert, and compare objects of class MWCharArray.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“clone” on page 4-116</td>
<td>Creates and returns a deep copy of this array.</td>
</tr>
<tr>
<td>“compareTo” on page 4-117</td>
<td>Compares this array with the specified array for order.</td>
</tr>
<tr>
<td>“equals” on page 4-117</td>
<td>Indicates whether some other array is equal to this one.</td>
</tr>
<tr>
<td>“hashCode” on page 4-117</td>
<td>Returns a hash code value for the array.</td>
</tr>
<tr>
<td>“sharedCopy” on page 4-117</td>
<td>Creates and returns a shared copy of this array.</td>
</tr>
<tr>
<td>“toString” on page 4-118</td>
<td>Returns a string representation of the array.</td>
</tr>
</tbody>
</table>

**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

```java
public Object clone()
```

**Input Parameters**

None
Example — Cloning a Character Array Object

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

```java
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

```java
public Object sharedCopy();
```

Input Parameters

None
Example — Making a Shared Copy of a Character Array Object

Create a shared copy of `MWCharArray` object `A`:

```java
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"

String. `MWCharArray` inherits this method from the `MWArray` class.

Using MWStructArray

This section covers the following topics:

- “Constructing an MWStructArray” on page 4-118
- “Methods to Destroy an MWStructArray” on page 4-120
- “Methods to Return Information About an MWStructArray” on page 4-121
- “Methods to Get and Set Data in the MWStructArray” on page 4-124
- “Methods to Copy, Convert, and Compare MWStructArrays” on page 4-132

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

```java
MWStructArray()```
To construct an `MWStructArray` object with the specified dimensions and field names, use

```java
MWStructArray(int[] dims, java.lang.String[] fieldnames)
```

To construct an `MWStructArray` object with the specified number of rows and columns, and field names, use

```java
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:

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;dispose&quot; on page 4-120</td>
<td>Frees the native MATLAB array contained by this array.</td>
</tr>
<tr>
<td>&quot;disposeArray&quot; on page 4-121</td>
<td>Frees all native MATLAB arrays contained in the input object.</td>
</tr>
</tbody>
</table>

**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
Using Class Methods

Example — Disposing of a Structure Array Object

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“classID” on page 4-122</td>
<td>Returns the MATLAB type of this array.</td>
</tr>
<tr>
<td>“fieldNames” on page 4-122</td>
<td>Returns the field names in this array.</td>
</tr>
<tr>
<td>“getDimensions” on page 4-123</td>
<td>Returns an array containing the size of each dimension of this array.</td>
</tr>
<tr>
<td>“isEmpty” on page 4-123</td>
<td>Tests whether the array has no elements.</td>
</tr>
</tbody>
</table>
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“numberOfDimensions” on page 4-123</td>
<td>Returns the number of dimensions of this array.</td>
</tr>
<tr>
<td>“numberOfElements” on page 4-123</td>
<td>Returns the total number of elements in this array.</td>
</tr>
<tr>
<td>“numberOfFields” on page 4-123</td>
<td>Returns the number of fields in this array.</td>
</tr>
</tbody>
</table>

**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

```java
public MWClassID classID()
```

**Input Parameters**

None

**Example — Getting the Class ID of a Structure Array**

Create an `MWStructArray` object and display the class ID:

```java
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:

```java
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:

```java
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`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“get” on page 4-125</td>
<td>Returns the element at the specified offset as an <code>Object</code>.</td>
</tr>
<tr>
<td>“getData” on page 4-127</td>
<td>Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.</td>
</tr>
</tbody>
</table>
Using Class Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“getField”</td>
<td>Returns a shared copy of the element at the specified one-based offset and field name in this array as an MWArray instance.</td>
</tr>
<tr>
<td>on page 4-128</td>
<td></td>
</tr>
<tr>
<td>“set”</td>
<td>Replaces the element at the specified one-based offset in this array with the specified element.</td>
</tr>
<tr>
<td>on page 4-129</td>
<td></td>
</tr>
<tr>
<td>“toArray”</td>
<td>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.</td>
</tr>
<tr>
<td>on page 4-131</td>
<td></td>
</tr>
</tbody>
</table>

**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().

```java
public Object get(int index)
pubic 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().

```java
public Object get(String fieldname, int index)
pubic 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
Using MWArray Classes

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 ith dimension.

Exceptions

The get method throws the following exception:

IndexOutOfBoundsException

The specified index parameter is invalid.

Example — Getting Structure Array Data with get

Get the data stored in field f2 at index (1,1) of MWStructArray object S:

```java
int[] sdims = {1, 2};
String[] sfields = {"f1", "f2", "f3"};
Integer val = new Integer(555);

MWStructArray S = new MWStructArray(sdims, sfields);

int[] index = {1, 1};
S.set(sfields[2], index, val);

Object d_out = S.get(sfields[2], index);
System.out.println("Structure data S(1,1).f2 is " + d_out.toString());
```

When run, the example displays this output:
Structure data S(1,1).f2 is 555

**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

```java
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:

```java
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: 
");
```
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 $1 \leq \text{index} \leq N$, 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 $1 \leq \text{index}[i] \leq N[i]$, 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

```java
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

```java
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();
```

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 4-133 | Creates and returns a deep copy of this array.
“compareTo” on page 4-134 | Compares this array with the specified array for order.
“equals” on page 4-134 | Indicates whether some other array is equal to this one.
“hashCode” on page 4-134 | Returns a hash code value for the array.
“sharedCopy” on page 4-134 | Creates and returns a shared copy of this array.
“toString” on page 4-135 | 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

```java
public Object clone()
```

**Input Parameters**

None

**Exceptions**

The clone method throws the following exception:

```java
IndexOutOfBoundsException
```

The specified index parameter is invalid.
Example — Cloning a Structure Array Object

Create an MWStructArray object and then a clone of that object:

```java
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

```java
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:

```java
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 4-136
- “Methods to Destroy an MWCellArray” on page 4-137
- “Methods to Return Information About an MWCellArray” on page 4-138
- “Methods to Get and Set Data in the MWCellArray” on page 4-140
- “Methods to Copy, Convert, and Compare MWCellArrays” on page 4-147
**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

```java
MWCellArray();
```

To construct an MWCellArray object with the specified dimensions, use

```java
MWCellArray(int[] dims);
```

To construct an MWCellArray object with the specified number of rows and columns, use

```java
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:

```java
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:

```java
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.
Using MWArray Classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“dispose” on page 4-138</td>
<td>Frees the native MATLAB array contained by this array.</td>
</tr>
<tr>
<td>“disposeArray” on page 4-138</td>
<td>Frees all native MATLAB arrays contained in the input object.</td>
</tr>
</tbody>
</table>

**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:

```java
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.

```java
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 4-139 | Returns the MATLAB type of this array.  
“getDimensions” on page 4-139 | Returns an array containing the size of each dimension of this array.  
“isEmpty” on page 4-140 | Tests whether the array has no elements.  
“numberOfDimensions” on page 4-140 | Returns the number of dimensions of this array.  
“numberOfElements” on page 4-140 | 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

```java
public MWClassID classID()
```

**Input Parameters**

None

**Example — Getting the Class ID of a Cell Array**

Create an MWCellArray object and display its class:

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“get” on page 4-140</td>
<td>Returns the element at the specified offset as an Object.</td>
</tr>
<tr>
<td>“getCell” on page 4-142</td>
<td>Returns a shared copy of the element at the specified one-based offset in this array as an MWArray instance.</td>
</tr>
<tr>
<td>“getData” on page 4-143</td>
<td>Returns a one-dimensional array containing a copy of the data in the underlying MATLAB array.</td>
</tr>
<tr>
<td>“set” on page 4-144</td>
<td>Replaces the element at the specified one-based offset in this array with the specified element.</td>
</tr>
<tr>
<td>“toArray” on page 4-146</td>
<td>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.</td>
</tr>
</tbody>
</table>

** 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:

```java
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

Use get to read index (1,3) of MWCellArray object C:

```java
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

**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:

```java
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 \(1 \leq index \leq N\), 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 \(1 \leq index[i] \leq N[i]\), 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()
```

gedata 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`:

```java
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
target 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 of the requested element in the MWCellArray

In the case where \texttt{index} is of type \texttt{int}, the valid range for \texttt{index} is $1 \leq \texttt{index} \leq N$, where \texttt{N} is the total number of elements in the array.

In the case where \texttt{index} is of type \texttt{int[]}, each element of the \texttt{index} vector is an index along one dimension of the MWCellArray object. The valid range for any index is $1 \leq \texttt{index}[i] \leq N[i]$, where \texttt{N[i]} is the size of the \texttt{i}th dimension.

Exceptions

The \texttt{set} method throws the following exception:

\texttt{IndexOutOfBoundsException}

The specified \texttt{index} parameter is invalid.

Example — Setting Values in a Cell Array

Set the value of the MWCellArray object \texttt{C} at index \texttt{(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:

```java
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`**

```java
int[] cdims = {1, 3};
MWCellArray C = new MWCellArray(cdims);

System.out.println("Data read from cell array C 
");
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“clone” on page 4-147</td>
<td>Creates and returns a deep copy of this array.</td>
</tr>
<tr>
<td>“compareTo” on page 4-148</td>
<td>Compares this array with the specified array for order.</td>
</tr>
<tr>
<td>“equals” on page 4-148</td>
<td>Indicates whether some other array is equal to this one.</td>
</tr>
<tr>
<td>“hashCode” on page 4-148</td>
<td>Returns a hash code value for the array.</td>
</tr>
<tr>
<td>“sharedCopy” on page 4-148</td>
<td>Creates and returns a shared copy of this array.</td>
</tr>
<tr>
<td>“toString” on page 4-149</td>
<td>Returns a string representation of the array.</td>
</tr>
</tbody>
</table>

**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:

```java
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:

```java
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.

---

**Note** MATLAB opaque arrays are not supported in the current release.

**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:

```java
double AReal = 24;

MWNumericArray A = new MWNumericArray(AReal, MWClassID.INT16);
System.out.println("Array A of type "+ A.classID() + " = 
" + 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:

```java
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:

```java
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:

```java
public final int hashCode()
```

**isNumeric.** This method tests if this type is numeric.

The prototype for `isNumeric` is as follows:

```java
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:

```java
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:

```java
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:

```java
public static final MWComplexity COMPLEX
```
Example – Determining the Complexity of an Array

Determine whether matrix $A$ is real or complex. The complexity method of $\text{MWNumericArray}$ returns an enumeration of type $\text{MWComplexity}$.

```java
double AReal = 24;
double AIImag = 5;

$\text{MWNumericArray A} = \text{new MWNumericArray}(\text{AReal}, \text{AIImag});$
System.out.println("A is a " + A.complexity() + " matrix");
```

When run, the example displays this output:

A is a complex matrix

Methods of $\text{MWComplexity}$

**toString.** This method returns a string representation of the property. The `toString` method of $\text{MWComplexity}$ overrides the `toString` method of class $\text{java.lang.Object}$.

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

```java
public java.lang.String toString()
```
Sample Applications (Java)

Note The examples for MATLAB Builder for Java are in \matlabroot\toolbox\javabuilder\Examples.

In addition to these examples, see “Example: Magic Square” on page 1-15 for a simple example that gets you started using MATLAB Builder for Java.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plot Example (p. 5-2)</td>
<td>How to encapsulate a MATLAB function that draws a plot given two input arguments</td>
</tr>
<tr>
<td>Spectral Analysis Example (p. 5-8)</td>
<td>How to create a class that has two methods</td>
</tr>
<tr>
<td>Matrix Math Example (p. 5-16)</td>
<td>How to create and use a class with three methods that encapsulate MATLAB functions</td>
</tr>
</tbody>
</table>
**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:

   \[ \text{matlabroot}\backslash\text{toolbox}\backslash\text{javabuilder}\backslash\text{Examples}\backslash\text{PlotExample} \]

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

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

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

   The following code defines the drawplot function:

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

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

4 Specify a Java component as follows:
   a. While in MATLAB, issue the following command to open the
      Deployment Tool dialog box:
         deploytool
   b. Create a new project with these settings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component name</td>
<td>plotdemo</td>
</tr>
<tr>
<td>Class name</td>
<td>plotter</td>
</tr>
<tr>
<td>Show verbose output</td>
<td>Selected</td>
</tr>
</tbody>
</table>

c. Add the computefft.m and plotfft.m M-files to the project.
   d. Save the project.

5 Build the component.

6 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**

```java
/* createplot.java
* This file is used as an example for the MATLAB
* Builder for Java product.
*
* Copyright 2001-2006 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++)
            {
```
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 the `plotter` object, as shown:
  ```java
  thePlot = new plotter();
  ```
- Calls the `drawplot` method to plot the equation using the MATLAB `plot` function, as shown:
  ```java
  thePlot.drawplot(x, y);
  ```
- Uses a try-catch block to catch and handle any exceptions.

7 Compile the createplot application using javac.
   a. On Windows, execute the following command:
      
      ```
javac -classpath
   .;matlabroot\java\jar\toolbox\javabuilder.jar;
   .\distrib\plotdemo.jar createplot.java
      
   b. On UNIX, execute this command:
      
      ```javac -classpath
   .;matlabroot/java/jar/toolbox/javabuilder.jar:
   ./distrib/plotdemo.jar createplot.java```

8 Run the application.

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

On Windows, type

```java -classpath
   .;matlabroot\java\jar\toolbox\javabuilder.jar;
   .\distrib\plotdemo.jar
   -Djava.library.path=matlabroot\bin\win32;..\distrib
   createplot```

On UNIX, type

```java -classpath
   .;matlabroot/java/jar/toolbox/javabuilder.jar:
   ./distrib/plotdemo.jar
   -Djava.library.path=matlabroot/bin/<Arch>/distrib
   createplot
   % where <Arch> = glux86 gluxa64 sol2```
Note The supported JRE version is 1.5.0. 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**

```matlab
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 2001-2006 The MathWorks, Inc.
if (isempty(data))
    fftdata = [];
    freq = [];
```
spectralAnalysisExample:

```plaintext
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`

```plaintext
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 2001-2006 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 6-2.

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 Specify a Java component as follows:

   a. While in MATLAB, issue the following command to open the Deployment Tool dialog box:

      ```
      deploytool
      ```

   b. Create a new project with these settings:

      | Field            | Value       |
      |------------------|-------------|
      | Component name   | spectralanalysis |
      | Class name       | fourier     |
      | Show verbose output | Selected   |

   c. 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.

   d. 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.

5 Build the component.
6 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.

```java
/* powerspect.java
 * This file is used as an example for the MATLAB
 * Builder for Java product.
 *
 * Copyright 2001-2006 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
            */
```
```java
* Sampling interval = 0.01
*/
int[] dims = {1, nSamples};
data = MWNumericArray.newInstance(dims, MWClassID.DOUBLE,
    MWComplexity.REAL);
for (int i = 1; i <= nSamples; i++)
{
    double t = (i-1)*interval;
    double x = Math.sin(2.0*Math.PI*15.0*t) +
        Math.sin(2.0*Math.PI*40.0*t) +
        Math.random();
data.set(i, x);
}

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

/* Compute power spectral density and plot result */
result = theFourier.plotfft(3, data,
    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:

  ```java
  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

7 Compile the **powerspect.java** application using **javac**.
   
   a. Open a Command Prompt window and **cd** to the
      ```bash
      matlabroot\work\spectralanalysis
      ```
   
   b. On Windows, execute the following command:

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

   c. On UNIX, execute the following command:

      ```bash
      javac -classpath
      .:matlabroot/java/jar/toolbox/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.

8 Run the application

   - On Windows, execute the **powerspect** class file as follows:

      ```bash
      java -classpath
On UNIX, execute the powerspect class file as follows:

```
java -classpath
   .:matlabroot/java/jar/toolbox/javabuilder.jar:.
distrib/spectralanalysis.jar
   -Djava.library.path=matlabroot/bin/<Arch>.
distrib
powerspect
% where <Arch> = glux86 gluxa64 sol2
```

**Note** The supported JRE version is 1.5.0. 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:
Matrix Math Example

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.mathworkstoolbox.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:

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

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

MATLAB Functions to Be Encapsulated

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

cholesky.m

```matlab
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 2001-2006 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 2001-2006 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 2001-2006 The MathWorks, Inc.

[Q,R] = qr(A);

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

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

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 Specify a Java component as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component name</td>
<td>factormatrix</td>
</tr>
<tr>
<td>Class name</td>
<td>factor</td>
</tr>
<tr>
<td>Show verbose output</td>
<td>Selected</td>
</tr>
</tbody>
</table>

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

7 Save the project.

8 Build the component.

9 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 2001-2006 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]++)
{
    {
        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:

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

10 Compile the getfactor application using javac.

cd to the `matlabroot\work\factormatrix` directory.

- On Windows, execute the following command:
  ```
javac -classpath
  .;\matlabroot\\java\\jar\\toolbox\\javabuilder.jar;
  .\distrib\factormatrix.jar getfactor.java
  ```

- On UNIX, execute the following command:
  ```
javac -classpath
  .:\matlabroot\\java\\jar\\toolbox\\javabuilder.jar:
  ./distrib/factormatrix.jar getfactor.java
  ```

11 Run the application.

Run getfactor using a nonsparse matrix

- On Windows, execute the getfactor class file as follows:
  ```
  java -classpath
  .;\matlabroot\\java\\jar\\toolbox\\javabuilder.jar;
  .\distrib\factormatrix.jar
  -Djava.library.path=matlabroot\bin\win32;.\distrib
  getfactor 4
  ```
• On UNIX, execute the getfactor class file as follows:

```
java -classpath
    ./matlabroot/java/jar/toolbox/javabuilder.jar:
    ./distrib/factormatrix.jar
    -Djava.library.path=matlabroot/bin/<Arch>/distrib
getfactor 4
% where <Arch> = glux86 gluxa64 sol2
```

**Note**  The supported JRE version is 1.5.0. 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
```
QR factorization:

**Q matrix:**

<table>
<thead>
<tr>
<th></th>
<th>-0.8944</th>
<th>-0.3586</th>
<th>-0.1952</th>
<th>0.1826</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4472</td>
<td></td>
<td>-0.7171</td>
<td>-0.3904</td>
<td>0.3651</td>
</tr>
<tr>
<td>0</td>
<td>0.5976</td>
<td>-0.5855</td>
<td>0.5477</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.6831</td>
<td>0.7303</td>
<td></td>
</tr>
</tbody>
</table>

**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:
Matrix Math Example

\[(\begin{array}{cccc}
(1,1) & 1.4142 \\
(1,2) & -0.7071 \\
(2,2) & 1.2247 \\
(2,3) & -0.8165 \\
(3,3) & 1.1547 \\
(3,4) & -0.8660 \\
(4,4) & 1.1180 \\
\end{array})\]

LU factorization:
L matrix:
\[(\begin{array}{cccc}
(1,1) & 1.0000 \\
(2,1) & -0.5000 \\
(2,2) & 1.0000 \\
(3,2) & -0.6667 \\
(3,3) & 1.0000 \\
(4,3) & -0.7500 \\
(4,4) & 1.0000 \\
\end{array})\]

U matrix:
\[(\begin{array}{cccc}
(1,1) & 2.0000 \\
(1,2) & -1.0000 \\
(2,2) & 1.5000 \\
(2,3) & -1.0000 \\
(3,3) & 1.3333 \\
(3,4) & -1.0000 \\
(4,4) & 1.2500 \\
\end{array})\]

QR factorization:
Q matrix:
\[(\begin{array}{cccc}
(1,1) & 0.8944 \\
(2,1) & -0.4472 \\
(1,2) & 0.3586 \\
(2,2) & 0.7171 \\
(3,2) & -0.5976 \\
(1,3) & 0.1952 \\
(2,3) & 0.3904 \\
(3,3) & 0.5855 \\
\end{array})\]
(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.
Reference Information for Java

Requirements for MATLAB Builder for Java (p. 6-2)
Software requirements for using MATLAB Builder for Java

MATLAB Builder for Java Graphical User Interface (p. 6-7)
Details about the windows, dialog boxes, menus, and buttons

Data Conversion Rules (p. 6-10)
Details about the way that MATLAB Builder for Java handles data

Programming Interfaces Generated by Java Builder (p. 6-14)
Details about the function signatures for methods that MATLAB Builder for Java creates

MWArray Class Specification (p. 6-19)
Link to class information
Requirements for MATLAB Builder for Java

- “System Requirements” on page 6-2
- “Limitations and Restrictions” on page 6-2
- “Settings for Environment Variables (Development Machine)” on page 6-2

System Requirements
System requirements and restrictions on use for MATLAB Builder for Java are as follows:

- All requirements for the 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 1.5.0. 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 the 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 6-3
• “Java CLASSPATH Variable” on page 6-4
• “Native Library Path Variables” on page 6-6

**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\j2sdk1.5.0.)

```
set JAVA_HOME=C:\java\jdk\j2sdk1.5.0
```

Alternatively, you can add jdk_directory/bin to the system path. For example:

```
set PATH=%PATH%;c:\java\jdk\j2sdk1.5.0\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 1.5.0. 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/j2sdk1.5.0.)
setenv JAVA_HOME /java/jdk/j2sdk1.5.0

Alternatively, you can add jdk_directory\bin to the system path.

**Java CLASSPATH Variable**

To build and run a Java application that encapsulates MATLAB 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.

```
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 `sol2`, 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:

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>PATH</td>
</tr>
<tr>
<td>Linux</td>
<td>LD_LIBRARY_PATH</td>
</tr>
<tr>
<td>Solaris</td>
<td>LD_LIBRARY_PATH</td>
</tr>
<tr>
<td>Macintosh</td>
<td>DYLD_LIBRARY_PATH</td>
</tr>
</tbody>
</table>

For information on how to set these path variables, see the following topics in the “Stand-Alone 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.
MATLAB Builder for Java Graphical User Interface

Use the MATLAB Builder for Java graphical user interface (GUI) to create Java classes that encapsulate your M-file functions. You can build and package Java components that support the classes you create.

To start the application, run the following command at the MATLAB prompt:

deploytool

Use the GUI to perform the following tasks:

- Create projects that define your Java components
- Name the classes to contain the M-file methods you want to use and associate the classes with a component in the project.
  
  Associate the appropriate methods with each class

- Build Java class files and methods from the M-files you specify.
- Create a component installer, a package you can use to install the components on another machine for development.

The Deployment Tool opens as shown.
Note that the Deployment Tool window can be docked or undocked.

The Deployment Tool toolbar has the following icons.

**Toolbar Icons**

<table>
<thead>
<tr>
<th>Name of Toolbar Icon</th>
<th>Icon</th>
<th>Click to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Project</td>
<td>![New Project Icon]</td>
<td>Create a new deployment project.</td>
</tr>
<tr>
<td>Open Project</td>
<td>![Open Project Icon]</td>
<td>View projects and select one to open.</td>
</tr>
<tr>
<td>Save Project</td>
<td>![Save Project Icon]</td>
<td>Save the current project, including all files and settings.</td>
</tr>
<tr>
<td>Add Class (only for components that contain classes)</td>
<td>![Add Class Icon]</td>
<td>Open the Add Class dialog box, where you can specify the name of a new class to be created as part of the current project.</td>
</tr>
</tbody>
</table>
### Toolbar Icons (Continued)

<table>
<thead>
<tr>
<th>Name of Toolbar Icon</th>
<th>Icon</th>
<th>Click to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
<td><img src="image" alt="Remove Icon" /></td>
<td>Remove the selected class folder or the selected files from the project.</td>
</tr>
<tr>
<td>Build</td>
<td><img src="image" alt="Build Icon" /></td>
<td>Build the components specified by the project, displaying the build process in the <strong>Deployment Tool Output</strong> pane.</td>
</tr>
<tr>
<td>Package</td>
<td><img src="image" alt="Package Icon" /></td>
<td>Create a self-extracting executable (Windows) or .zip file (UNIX) that contains the files needed to use the component in an application.</td>
</tr>
<tr>
<td>Settings</td>
<td><img src="image" alt="Settings Icon" /></td>
<td>Change settings for the project. You can improve performance by changing settings, such as whether to exclude the MCR or change the search path when building.</td>
</tr>
<tr>
<td>Help</td>
<td><img src="image" alt="Help Icon" /></td>
<td>View a quick start and steps for using the Deployment Tool.</td>
</tr>
</tbody>
</table>
Data Conversion Rules

- “Java to MATLAB Conversion” on page 6-10
- “MATLAB to Java Conversion” on page 6-12
- “Unsupported MATLAB Array Types” on page 6-13

Java to MATLAB Conversion

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

<table>
<thead>
<tr>
<th>Java Type</th>
<th>MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>float</td>
<td>single</td>
</tr>
<tr>
<td>byte</td>
<td>int8</td>
</tr>
<tr>
<td>int</td>
<td>int32</td>
</tr>
<tr>
<td>short</td>
<td>int16</td>
</tr>
<tr>
<td>long</td>
<td>int64</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>boolean</td>
<td>logical</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>double</td>
</tr>
</tbody>
</table>
### Java to MATLAB Conversion Rules (Continued)

<table>
<thead>
<tr>
<th>Java Type</th>
<th>MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Float</td>
<td>single</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>int8</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>int32</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>int64</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>int16</td>
</tr>
<tr>
<td>java.lang.Number</td>
<td>double</td>
</tr>
</tbody>
</table>

**Note** Subclasses of `java.lang.Number` not listed above are converted to `double`.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Boolean</td>
<td>logical</td>
</tr>
<tr>
<td>java.lang.Character</td>
<td>char</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>char</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>MATLAB Type</th>
<th>Java Type (Primitive)</th>
<th>Java Type (Object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell</td>
<td>N/A</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> Cell arrays are constructed and accessed as arrays of MWArray.</td>
<td></td>
</tr>
<tr>
<td>structure</td>
<td>N/A</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> Structure arrays are constructed and accessed as arrays of MWArray.</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>java.lang.Character</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>single</td>
<td>float</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>int8</td>
<td>byte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>int16</td>
<td>short</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>int32</td>
<td>int</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>int64</td>
<td>long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>uint8</td>
<td>byte</td>
<td>java.lang.ByteJava has no unsigned type to represent the uint8 used in MATLAB. Construction of and access to MATLAB arrays of an unsigned type requires conversion.</td>
</tr>
</tbody>
</table>
MATLAB to Java Conversion Rules (Continued)

<table>
<thead>
<tr>
<th>MATLAB Type</th>
<th>Java Type (Primitive)</th>
<th>Java Type (Object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16</td>
<td>short</td>
<td>java.lang.short</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>uint32</td>
<td>int</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>uint64</td>
<td>long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>logical</td>
<td>boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>Function handle</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>Java class</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>User class</td>
<td>Not supported</td>
<td></td>
</tr>
</tbody>
</table>

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

- “APIs Based on MATLAB Function Signatures” on page 6-14
- “Standard API” on page 6-15
- “mlx API” on page 6-17
- “Code Fragment: Signatures Generated for myprimes Example” on page 6-17

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 pre-allocated 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
  java.util.List outputs = new ArrayList(3);
  myclass.myfunction(outputs, inputs); // outputs contains 0 elements!
  ```

  And this would be the correct usage:

  ```java
  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.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>API to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic MATLAB function</td>
<td>function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)</td>
</tr>
<tr>
<td>API if there are no input</td>
<td>public Object[] foo( int numArgsOut</td>
</tr>
<tr>
<td>arguments</td>
<td></td>
</tr>
<tr>
<td>API if there is one input</td>
<td>public Object[] foo( int numArgsOut, Object In1</td>
</tr>
<tr>
<td>argument</td>
<td></td>
</tr>
<tr>
<td>API if there are two to N input</td>
<td>public Object[] foo( int numArgsOut, Object In1, Object In2,</td>
</tr>
<tr>
<td>arguments</td>
<td></td>
</tr>
<tr>
<td>API if there are optional</td>
<td>public Object[] foo( int numArgsOut, Object in1,</td>
</tr>
<tr>
<td>arguments, represented</td>
<td></td>
</tr>
</tbody>
</table>
by the `varargin` argument

Object \( \text{in2}, \ldots, \text{inN} \), \( \text{Object \ varargin} \)

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

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

For a function with the following structure:

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

Java Builder generates the following API, as the mlx interface:

```java
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.

```java
/* 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 6-14 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.
Functions — Alphabetical List
**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.

See “Creating a Java Component” on page 1-5 to get started using the Deployment Tool to create Java components, and see “Using the GUI to Create and Package a Deployable Component” in the MATLAB Compiler documentation for information about using the Deployment Tool to create standalone applications and libraries.
Examples

Use this list to find examples in the documentation.
Quick Start

“Example: Magic Square” on page 1-15

Handling Data

“Code Fragment: Passing an MWArray” on page 3-7
“Code Fragment: Passing a Java Double Object” on page 3-8
“Code Fragment: Passing an MWArray” on page 3-8
“Code Fragment: Passing Variable Numbers of Inputs” on page 3-10
“Code Fragment: Passing Array Inputs” on page 3-11
“Code Fragment: Passing a Variable Number of Outputs” on page 3-12
“Code Fragment: Passing Optional Arguments with the Standard Interface” on page 3-13
“Code Fragment: Using MWArray Query” on page 3-16
“Handling Data Conversion Between Java and MATLAB” on page 3-28
“Examples of Using set” on page 4-19
“Examples of Using get” on page 4-20
“Examples of Using set and get Methods” on page 4-26
“Code Fragment: Signatures Generated for myprimes Example” on page 6-17

Handling Errors

“Code Fragment: Handling an Exception in the Called Function” on page 3-20
“Code Fragment: Handling an Exception in the Calling Function” on page 3-21
“Code Fragment: Catching General Exceptions” on page 3-22
“Code Fragment: Catching Multiple Exception Types” on page 3-23
Handling Memory

“Code Fragment: Use try-finally to Ensure Resources Are Freed” on page 3-27

Sample Applications (Java)

“Plot Example” on page 5-2
“Spectral Analysis Example” on page 5-8
“Matrix Math Example” on page 5-16
A

API
  data conversion classes 3-6
  MATLAB Builder for Java 4-1
arguments
  optional 3-9
    standard interface 3-13
  optional inputs 3-10
  optional outputs 3-12
  passing 3-6
array API
  overview 4-2
array inputs
  passing 3-11
arrays
  cell 4-31
    constructing 4-32
  character 4-26
    constructing 4-27
  logical 4-22
    constructing 4-22
  numeric 4-7
    constructing 4-7

B

build output
  componentLibInfo.java 2-9

C

calling interface
  standard 6-15

calling methods 1-23

cell arrays 4-31
  constructing 4-32

calling methods of a 1-23

calling methods of a

checking

checked exceptions

checking

check exceptions

checked exceptions

checked exceptions

A

API
  data conversion classes 3-6
  MATLAB Builder for Java 4-1
arguments
  optional 3-9
    standard interface 3-13
  optional inputs 3-10
  optional outputs 3-12
  passing 3-6
array API
  overview 4-2
array inputs
  passing 3-11
arrays
  cell 4-31
    constructing 4-32
  character 4-26
    constructing 4-27
  logical 4-22
    constructing 4-22
  numeric 4-7
    constructing 4-7

B

build output
  componentLibInfo.java 2-9

C

calling interface
  standard 6-15

calling methods 1-23

cell arrays 4-31
  constructing 4-32

calling methods of a 1-23

calling methods of a

checking

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

A

API
  data conversion classes 3-6
  MATLAB Builder for Java 4-1
arguments
  optional 3-9
    standard interface 3-13
  optional inputs 3-10
  optional outputs 3-12
  passing 3-6
array API
  overview 4-2
array inputs
  passing 3-11
arrays
  cell 4-31
    constructing 4-32
  character 4-26
    constructing 4-27
  logical 4-22
    constructing 4-22
  numeric 4-7
    constructing 4-7

B

build output
  componentLibInfo.java 2-9

C

calling interface
  standard 6-15

calling methods 1-23

cell arrays 4-31
  constructing 4-32

calling methods of a 1-23

calling methods of a

checking

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

A

API
  data conversion classes 3-6
  MATLAB Builder for Java 4-1
arguments
  optional 3-9
    standard interface 3-13
  optional inputs 3-10
  optional outputs 3-12
  passing 3-6
array API
  overview 4-2
array inputs
  passing 3-11
arrays
  cell 4-31
    constructing 4-32
  character 4-26
    constructing 4-27
  logical 4-22
    constructing 4-22
  numeric 4-7
    constructing 4-7

B

build output
  componentLibInfo.java 2-9

C

calling interface
  standard 6-15

calling methods 1-23

cell arrays 4-31
  constructing 4-32

calling methods of a 1-23

calling methods of a

checking

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

A

API
  data conversion classes 3-6
  MATLAB Builder for Java 4-1
arguments
  optional 3-9
    standard interface 3-13
  optional inputs 3-10
  optional outputs 3-12
  passing 3-6
array API
  overview 4-2
array inputs
  passing 3-11
arrays
  cell 4-31
    constructing 4-32
  character 4-26
    constructing 4-27
  logical 4-22
    constructing 4-22
  numeric 4-7
    constructing 4-7

B

build output
  componentLibInfo.java 2-9

C

calling interface
  standard 6-15

calling methods 1-23

cell arrays 4-31
  constructing 4-32

calling methods of a 1-23

calling methods of a

checking

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions

checked exceptions
constructing
  cell arrays 4-32
  character arrays 4-27
  logical arrays 4-22
  mwarrays 4-38
  mwcellarrays 4-136
  mwchararrays 4-108
  mwlogicalarrays 4-92
  mwnumericarrays 4-59
  mwstructarrays 4-118
  numeric arrays 4-7
  sparse arrays 4-15
converting characters to MATLAB char array 6-11
converting data 3-7
  Java to MATLAB 6-10
  MATLAB to Java 6-12
converting strings to MATLAB char array 6-11
create plot example 5-2
creating objects 1-23 3-3

D
data conversion 3-7
  characters, strings 6-11
  Java to MATLAB 6-10
  MATLAB to Java 6-12
  rules for Java components 6-10
  unsigned integers 6-13
data conversion classes 4-1
mwarray 4-38
  comparing 4-49
  constructors 4-38
  converting 4-49
  copying 4-49
  disposing 4-39
  get information on 4-40
  get, set 4-44
  sparse 4-54
mwcellarray 4-135
  comparing 4-147
  constructors 4-136
  converting 4-147
  copying 4-147
  disposing 4-137
  get information on 4-138
  get, set 4-140
mwcellarray 4-107
  comparing 4-116
  constructors 4-108
  converting 4-116
  copying 4-116
  creating 4-109
  disposing 4-109
  get information on 4-111
  get, set 4-112
mwclassid 4-149
  fields 4-149
  methods 4-151
mwcomplexity 4-152
  fields 4-152
  methods 4-153
mwlogicalarray 4-92
  comparing 4-104
  constructors 4-92
  converting 4-104
  copying 4-104
  creating 4-93
  disposing 4-93
  get information on 4-98
  get, set 4-100
  sparse 4-107
mwnumericarray 4-58
  comparing 4-87
  constants 4-90
  constructors 4-59
  converting 4-87
  copying 4-87
  creating 4-63
  disposing 4-63
  get information on 4-71
  get, set: imaginary 4-79
  get, set: real 4-75
  sparse 4-90
mwstructarray 4-118
  comparing 4-132
  constructors 4-118
  converting 4-132
  copying 4-132
  disposing 4-120
  get information on 4-121
  get, set 4-124
data conversion rules 3-28
deploytool function 7-2
development machine
  running the application 1-21
dispose 3-26 4-5
  mwareay 4-39
  mwcellarray 4-138
  mwchararray 4-110
  mwlogicalarray 4-98
  mwnumericarray 4-71
  mwstructarray 4-120
disposearray 4-5
  mwareay 4-40
  mwcellarray 4-138
  mwchararray 4-111
  mwlogicalarray 4-98
  mwnumericarray 4-71
  mwstructarray 4-121
disposing mwarrays 4-39
  disposing of mwnumericarrays 4-63

E
  environment variables
    classpath 6-4
    java_home 6-3
    ld_library_path 6-6
    path 6-6
    setting 6-2
equals
  mwareay 4-52
  mwcellarray 4-148
  mwchararray 4-117
  mwclassid 4-151
  mwlogicalarray 4-105
  mwnumericarray 4-89
  mwstructarray 4-134
error handling 3-19
example applications
  Java 5-1
examples
  Java create plot 5-2
  magic square in C# 1-15
exceptions 3-19
  catching 3-22 to 3-23
  checked
    in called function 3-20
    in calling function 3-21
  general 3-22
  unchecked 3-22

F
  factory methods
    of mwcellarray 4-34
    of mwchararray 4-30
    of mwlogicalarray 4-25
    of mwnumericarray 4-17
  fieldnames
    mwstructarray 4-122
finalization 3-27
freeing native resources
try-finally 3-27

g
  garbage collection 3-25
get 4-5
    mwarray 4-45
    mwcellarray 4-34 4-140
    mwchararray 4-30 4-112
    mwlogicalarray 4-25 4-100
    mwnumericarray 4-17 4-76
    example 4-20
    mwstructarray 4-125
getboolean
    mwlogicalarray 4-101
getbyte
    mwnumericarray 4-78
getchar
    mwchararray 4-113
getdata 4-5
    mwarray 4-46
    mwcellarray 4-37 4-143
    mwchararray 4-112
    mwlogicalarray 4-100
    mwnumericarray 4-76
    mwstructarray 4-127
getdimensions 4-5
    mwarray 4-42
    mwcellarray 4-139
    mwchararray 4-112
    mwlogicalarray 4-100
    mwnumericarray 4-72
    mwstructarray 4-123
getdouble
    mwnumericarray 4-76
geteps
    mwnumericarray 4-91
getfield

  mwstructarray 4-128
getfloat
    mwnumericarray 4-77
getimagbyte
    mwnumericarray 4-85
getimag
    mwnumericarray 4-81
getimagdata
    mwnumericarray 4-82
getimagdouble
    mwnumericarray 4-83
getimagfloat
    mwnumericarray 4-84
getimagint
    mwnumericarray 4-84
getimaglong
    mwnumericarray 4-84
getimagshort
    mwnumericarray 4-84
getinf
    mwnumericarray 4-91
getint
    mwnumericarray 4-77
getlong
    mwnumericarray 4-77
genan
    mwnumericarray 4-92
getshort
    mwnumericarray 4-77
getsize
    mwclassid 4-151
GUI
    icons 6-8

h
hashcode
  mwarray 4-52
  mwcellarray 4-148
  mwchararray 4-117
Index

mwclassid 4-151
mwlogicalarray 4-106
mwnumericarray 4-89
mwstructarray 4-134

I
importing classes 1-23
isempty 4-5
    mwarray 4-42
    mwcellarray 4-140
    mwchararray 4-112
    mwlogicalarray 4-100
    mwnumericarray 4-72
    mwstructarray 4-123
isfinite
    mwnumericarray 4-72
isinf
    mwnumericarray 4-73
isnan
    mwnumericarray 4-74
isnumeric
    mwclassid 4-152
issparse 4-5
    mwarray 4-55

J
jagged arrays
    constructing 4-11
Java application
    running on the development machine 1-21
    sample application
        usemyclass.java 3-5
    writing 5-1

java builder api
    mwarray
        comparing 4-49
        constructors 4-38
        converting 4-49
        copying 4-49
        disposing 4-39
        get information on 4-40
        get, set 4-44
        sparse 4-54
    mwcellarray 4-135
        comparing 4-147
        constructors 4-136
        converting 4-147
        copying 4-147
        disposing 4-137
        get information on 4-138
        get, set 4-140
    mwchararray 4-107
        comparing 4-116
        constructors 4-108
        converting 4-116
        copying 4-116
        creating 4-109
        disposing 4-109
        get information on 4-111
        get, set 4-112
    mwclassid 4-149
        fields 4-149
        methods 4-151
    mwcomplexity 4-152
        fields 4-152
        methods 4-153
mwlogicalarray 4-92
  comparing 4-104
  constructors 4-92
  converting 4-104
  copying 4-104
  creating 4-93
  disposing 4-93
  get information on 4-98
  get, set 4-100
  sparse 4-107
mwnumericarray 4-58
  comparing 4-87
  constants 4-90
  constructors 4-59
  converting 4-87
  copying 4-87
  creating 4-63
  disposing 4-63
  get information on 4-71
  get, set: imaginary 4-79
  get, set: real 4-75
  sparse 4-90
mwstructarray 4-118
  comparing 4-132
  constructors 4-118
  converting 4-132
  copying 4-132
  disposing 4-120
  get information on 4-121
  get, set 4-124
Java Builder API
  mwarray 4-38
Java classes 2-1
Java component
  instantiating classes 3-3
  Java examples 5-1
  overview of creating 1-5
  specifying 3-2
Java interfaces
  mwarray 4-4
Java reflection 3-14
Java to MATLAB data conversion 6-10
java_home variable 6-3
JVM 3-25

L
  ld_library_path variable 6-6
  LibInfo.java 2-9
  limitations 6-2
    platform-specific 2-10 3-2
  logical arrays 4-22
    constructing 4-22

M
  M-file method
    myprimes.m 3-5
  MATLAB Builder for Java
    introduction 1-2
    system requirements 6-2
  MATLAB to Java data conversion 6-12
  matrix math example
    Java 5-16
  maximumnonzeros 4-6
    mwarray 4-57
memory
  preserving 3-25
memory management
  native resources 3-25
method overrides
  mwarray 4-3
method signatures
  standard interface
    method signatures 3-8 6-14
methods
  adding 5-8
  calling 1-23
  error handling 3-19
  mwarray 4-4
mwcellarray 4-34
mwchararray 4-30
mwlogicalarray 4-25
mwnumericarray 4-17
of MWArray 3-8 3-28

multidimensional numeric arrays
constructing 4-10

mwarray 4-2 4-38
comparing 4-49
constructors 4-38
converting 4-49
copying 4-49
disposing 4-49
get, set 4-44
Java interfaces 4-3
method overrides 4-3
methods of 4-4
sparse 4-54

MWArray 3-6 4-1
MWArray class library
See also Data conversion 1-15

MWarray methods
get 4-30
set 4-30

mwclassid

cell 4-149
char 4-150
double 4-150
fields 4-149
function 4-150
int16 4-150
int32 4-150
int64 4-150
int8 4-150
logical 4-150
methods 4-151
object 4-150
opaque 4-151
single 4-150
struct 4-150
uint16 4-150
uint32 4-151
uint64 4-151
uint8 4-150
unknown 4-151

mwcomplexity 4-152
complex 4-152
fields 4-152
methods 4-153
real 4-152

mwlogicalarray 4-22 4-92
comparing 4-104
constructors 4-92
Index

converting 4-104
copying 4-104
creating 4-93
disposing 4-93
get information on 4-98
get, set 4-100
newinstance 4-23
newsparse 4-23
sparse 4-107
mwlogicalarray methods
get 4-25
set 4-25
mwnumericarray 4-7 4-58
comparing 4-87
constants 4-90
constructors 4-59
converting 4-87
copying 4-87
creating 4-63
disposing 4-63
get information on 4-71
get, set
imaginary 4-79
real 4-75
newinstance 4-12
newsparse 4-12
sparse 4-90
mwnumericarray methods
get 4-17
set 4-17
mwstructarray 4-118
comparing 4-132
constructors 4-118
converting 4-132
copying 4-132
disposing 4-120
get information on 4-121
get, set 4-124
myprimes.m 3-5

N
native resources
dispose 3-26
finalizing 3-27
newinstance
mwchararray 4-29 4-109
mwlogicalarray 4-23 4-94
mwnumericarray 4-12 4-63
newsparse
mwlogicalarray 4-23 4-95
mwnumericarray 4-12 4-66
numberofdimensions 4-6
mwarray 4-43
mwcellarray 4-140
mwchararray 4-112
mwlogicalarray 4-100
mwnumericarray 4-75
mwstructarray 4-123
numberofelements 4-6
mwarray 4-43
mwcellarray 4-140
mwchararray 4-112
mwlogicalarray 4-100
mwnumericarray 4-75
mwstructarray 4-123
numberoffields
mwstructarray 4-123
numberofnonzeros 4-6
mwarrays 4-58
numeric arrays 4-7
constructing 4-7
numeric matrices
constructing 4-10

O
objects
creating 1-23 3-3
operating system issues 2-10 3-2
optional arguments 3-9
input 3-10
output 3-12
standard interface 3-13

P
passing arguments 3-6
passing array inputs 3-11
passing data
matlab to java 2-7
path variable 6-6
platform issues 2-10 3-2
portability 2-10 3-2
programming
overview 1-12
project
elements of 2-3

R
requirements
system 6-2
resource management 3-25
restrictions 6-2
return values
handling 3-14
java reflection 3-14
mwarray query 3-16
rowindex 4-6
mwarray 4-57

S
set 4-6
mwarray 4-47
mwcellarray 4-34 4-144
mwchararray 4-30 4-114
mwlogicalarray 4-25 4-102
mwnumericarray 4-17 4-79
example 4-19
mwstructarray 4-129
setimag
mwnumericarray 4-85
setting environment variables 6-2
sharedcopy 4-7
mwarray 4-53
mwcellarray 4-148
mwchararray 4-117
mwlogicalarray 4-106
mwnumericarray 4-89
mwstructarray 4-134
sparse
mwlogicalarray 4-107
mwnumericarray 4-90
sparse arrays 4-54
constructing 4-15
standard interface 6-15
passing optional arguments 3-13
system requirements 6-2

toarray 4-7
mwarray 4-48
mwcellarray 4-37 4-146
mwchararray 4-116
mwlogicalarray 4-104
mwnumericarray 4-79
mwstructarray 4-131
toimagarray
mwnumericarray 4-86
toolbar
icons 6-8
tostring
mwarray 4-54
mwcellarray 4-149
mwchararray 4-118
mwclassid 4-152
mwcomplexity 4-153
mwlogicalarray 4-106
mwnumericarray 4-89
mwstructarray 4-135
try-finally 3-27

U
unchecked exceptions 3-22
usage information
  data conversion classes 4-1

getting started 1-1
sample Java applications 5-1
usemyclass.java application 3-5
utility classes
  base class 4-2
  overview 4-2