This routine multiplies an input scalar times an input scalar or matrix.

```
C=================================================================
C     xtimesy.f
C
C     Multiply the first input by the second input.
C
C     This is a MEX file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.12 $
C=================================================================

C     Computational subroutine
      subroutine xtimesy(x, y, z, m, n)
      real*8  x, y(3,3), z(3,3)
      integer m, n
      do 20 i=1,m
         do 10 j=1,n
            z(i,j) = x*y(i,j)
 10      continue
 20   continue
      return
      end


C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform

      integer plhs(*), prhs(*)
      integer mxCreateDoubleMatrix
      integer x_pr, y_pr, z_pr
C-----------------------------------------------------------------

      integer nlhs, nrhs
      integer m, n, size
      integer mxGetM, mxGetN, mxIsNumeric
      real*8  x, y(3,3), z(3,3)
```

```
C     Check for proper number of arguments.
      if (nrhs .ne. 2) then
         call mexErrMsgTxt('Two inputs required.')
      elseif (nlhs .ne. 1) then
         call mexErrMsgTxt('One output required.')
      endif

C     Check to see both inputs are numeric.
      if (mxIsNumeric(prhs(1)) .ne. 1) then
         call mexErrMsgTxt('Input #1 is not a numeric.')
      elseif (mxIsNumeric(prhs(2)) .ne. 1) then
         call mexErrMsgTxt('Input #2 is not a numeric array.')
      endif

C     Check that input #1 is a scalar.
      m = mxGetM(prhs(1))
      n = mxGetN(prhs(1))
      if(n .ne. 1 .or. m .ne. 1) then
         call mexErrMsgTxt('Input #1 is not a scalar.')
      endif

C     Get the size of the input matrix.
      m = mxGetM(prhs(2))
      n = mxGetN(prhs(2))
      size = m*n

C     Create matrix for the return argument.
      plhs(1) = mxCreateDoubleMatrix(m, n, 0)
      x_pr = mxGetPr(prhs(1))
      y_pr = mxGetPr(prhs(2))
      z_pr = mxGetPr(plhs(1))

C     Load the data into Fortran arrays.
      call mxCopyPtrToReal8(x_pr, x, 1)
      call mxCopyPtrToReal8(y_pr, y, size)

C     Call the computational subroutine.
      call xtimesy(x, y, z, m, n)
```


Ex.1049 / Page 438 of 748
Intel Corporation et al

```
C     Load the output into a MATLAB array.
      call mxCopyReal8ToPtr(z, z_pr, size)

      return
      end
```

As this example shows, creating MEX-file gateways that handle multiple inputs and outputs is straightforward. All you need to do is keep track of which indices of the vectors `prhs` and `plhs` correspond to which input and output arguments of your function. In this example, the input variable `x` corresponds to `prhs(1)` and the input variable `y` to `prhs(2)`.

For an input scalar `x` and a real 3-by-3 matrix,

```
x = 3; y = ones(3);
```

typing

```
z = xtimesy(x, y)
```

yields this result

```
z =
     3     3     3
     3     3     3
     3     3     3
```

Handling Complex Data

MATLAB stores complex double-precision data as two vectors of numbers — one contains the real data and one contains the imaginary data. The API provides two functions, `mxCopyPtrToComplex16` and `mxCopyComplex16ToPtr`, which allow you to copy the MATLAB data to a native `complex*16` Fortran array.

This example takes two complex vectors (of length 3) and convolves them.

```
C=================================================================
C     convec.f
C     Example for illustrating how to pass complex data from
C     MATLAB to FORTRAN (using COMPLEX data type) and back
C     again.
C
C     Convolves two complex input vectors.
```

```
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.15 $
C=================================================================

C     Computational subroutine
      subroutine convec(x, y, z, nx, ny)
      complex*16 x(*), y(*), z(*)
      integer nx, ny

C     Initialize the output array
      do 10 i=1,nx+ny-1
         z(i) = (0.0,0.0)
 10   continue

      do 30 i=1,nx
         do 20 j=1,ny
            z(i+j-1) = z(i+j-1) + x(i) * y(j)
 20      continue
 30   continue
      return
      end


C     The gateway routine.
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform.

      integer plhs(*), prhs(*)
      integer mxGetPr, mxGetPi, mxCreateDoubleMatrix
C-----------------------------------------------------------------

      integer nlhs, nrhs
      integer mx, nx, my, ny, nz
      integer mxGetM, mxGetN, mxIsComplex
      complex*16 x(100), y(100), z(199)
```

```
C     Check for proper number of arguments.
      if (nrhs .ne. 2) then
         call mexErrMsgTxt('Two inputs required.')
      elseif (nlhs .gt. 1) then
         call mexErrMsgTxt('Too many output arguments.')
      endif

C     Check that inputs are both row vectors.
      mx = mxGetM(prhs(1))
      nx = mxGetN(prhs(1))
      my = mxGetM(prhs(2))
      ny = mxGetN(prhs(2))
      nz = nx+ny-1

C     Only handle row vector input.
      if(mx .ne. 1 .or. my .ne. 1) then
         call mexErrMsgTxt('Both inputs must be row vector.')

C     Check sizes of the two inputs.
      elseif(nx .gt. 100 .or. ny .gt. 100) then
         call mexErrMsgTxt('Inputs must have less than 100
                           elements.')

C     Check to see both inputs are complex.
      elseif ((mxIsComplex(prhs(1)) .ne. 1) .or.
     +        (mxIsComplex(prhs(2)) .ne. 1)) then
         call mexErrMsgTxt('Inputs must be complex.')
      endif

C     Create the output array.
      plhs(1) = mxCreateDoubleMatrix(1, nz, 1)

C     Load the data into Fortran arrays(native COMPLEX data).
      call mxCopyPtrToComplex16(mxGetPr(prhs(1)),
                                mxGetPi(prhs(1)), x, nx)
      call mxCopyPtrToComplex16(mxGetPr(prhs(2)),
                                mxGetPi(prhs(2)), y, ny)

C     Call the computational subroutine.
      call convec(x, y, z, nx, ny)
```


Ex.1049 / Page 441 of 748
Intel Corporation et al

```
C     Load the output into a MATLAB array.
      call mxCopyComplex16ToPtr(z, mxGetPr(plhs(1)),
                                   mxGetPi(plhs(1)), nz)
      return
      end
```

Entering these numbers at the MATLAB prompt

```
x = [3 - 1i, 4 + 2i, 7 - 3i]

x =

   3.0000 - 1.0000i   4.0000 + 2.0000i   7.0000 - 3.0000i

y = [8 - 6i, 12 + 16i, 40 - 42i]

y =

   8.0000 - 6.0000i  12.0000 +16.0000i  40.0000 -42.0000i
```

and invoking the new MEX-file

```
z = convec(x, y)
```

results in

```
z =

   1.0e+02 *

  Columns 1 through 4

   0.1800 - 0.2600i   0.9600 + 0.2800i   1.3200 - 1.4400i
   3.7600 - 0.1200i

  Column 5

   1.5400 - 4.1400i
```

which agrees with the results the built-in MATLAB function `conv.m` produces.

Ex.1049 / Page 442 of 748
Intel Corporation et al

Dynamically Allocating Memory

It is possible to allocate memory dynamically in a Fortran MEX-file, but you must use `%val` to do it. This example takes an input matrix of real data and doubles each of its elements.

```
C=================================================================
C     dblmat.f
C     Example for illustrating how to use %val.
C     Doubles the input matrix. The demo only handles real part
C     of input.
C
C     NOTE: If your Fortran compiler does not support %val,
C     use mxCopy_routine.
C     NOTE: The subroutine compute() is in the file called
C     compute.f.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.13 $
C=================================================================

C     The gateway subroutine
      subroutine mexfunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform

      integer plhs(*), prhs(*)
      integer pr_in, pr_out
      integer mxGetPr, mxCreateDoubleMatrix
C-----------------------------------------------------------------

      integer nlhs, nrhs, mxGetM, mxGetN
      integer m_in, n_in, size

      if(nrhs .ne. 1) then
         call mexErrMsgTxt('One input required.')
      endif
      if(nlhs .gt. 1) then
         call mexErrMsgTxt('Less than one output required.')
```

```
      endif

      m_in = mxGetM(prhs(1))
      n_in = mxGetN(prhs(1))
      size = m_in * n_in
      pr_in = mxGetPr(prhs(1))
      plhs(1) = mxCreateDoubleMatrix(m_in, n_in, 0)
      pr_out = mxGetPr(plhs(1))

C     Call the computational routine.
      call compute(%val(pr_out), %val(pr_in), size)

      return
      end
```

This is the subroutine that `dblmat` calls to double the input matrix.

```
C=================================================================
C     compute.f
C
C     This subroutine doubles the input matrix. Your version of
C     compute() may do whaveter you would like it to do.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.3 $
C=================================================================

C     Computational subroutine
      subroutine compute(out_mat, in_mat, size)
      integer size, i
      real*8  out_mat(*), in_mat(*)

      do 10 i=1,size
         out_mat(i) = 2*in_mat(i)
 10   continue

      return
      end
```

For an input 2-by-3 matrix

```
x = [1 2 3; 4 5 6];
```

typing

```
y = dblmat(x)
```

yields

```
y =
     2     4     6
     8    10    12
```

Note The `dblmat.f` example, as well as `fulltosparse.f` and `sincall.f`, are split into two parts, the gateway and the computational subroutine, because of restrictions in some compilers.

Handling Sparse Matrices

The MATLAB API provides a set of functions that allow you to create and manipulate sparse matrices from within your MEX-files. There are special parameters associated with sparse matrices, namely `ir`, `jc`, and `nzmax`. For information on how to use these parameters and how MATLAB stores sparse matrices in general, refer to the section on "The MATLAB Array" on page 3-5.

Note Sparse array indexing is zero-based, not one-based.

This example illustrates how to populate a sparse matrix.

```
C=================================================================
C     fulltosparse.f
C     Example for illustrating how to populate a sparse matrix.
C     For the purpose of this example, you must pass in a
C     non-sparse 2-dimensional argument of type real double.
C
C     NOTE: The subroutine loadsparse() is in the file called
C     loadsparse.f.
```

```
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.6 $
C=================================================================

C     The gateway routine.
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform

      integer plhs(*), prhs(*)
      integer mxGetPr, mxCreateSparse, mxGetIr, mxGetJc
      integer pr, sr, irs, jcs
C-----------------------------------------------------------------

      integer nlhs, nrhs
      integer m, n, nzmax
      integer mxGetM, mxGetN, mxIsComplex, mxIsDouble
      integer loadsparse

C     Check for proper number of arguments.
      if (nrhs .ne. 1) then
         call mexErrMsgTxt('One input argument required.')
      endif
      if (nlhs .gt. 1) then
         call mexErrMsgTxt('Too many output arguments.')
      endif

C     Check data type of input argument.
      if (mxIsDouble(prhs(1)) .eq. 0) then
       call mexErrMsgTxt('Input argument must be of type double.')
      endif
      if (mxIsComplex(prhs(1)) .eq. 1) then
         call mexErrMsgTxt('Input argument must be real only')
      endif

C     Get the size and pointers to input data.
      m = mxGetM(prhs(1))
```

```
      n = mxGetN(prhs(1))
      pr = mxGetPr(prhs(1))

C     Allocate space.
C     NOTE: Assume at most 20% of the data is sparse.
      nzmax = dble(m*n) *.20 + .5

C    NOTE: The maximum number of non-zero elements cannot be less
C     than the number of columns in the matrix.
      if (n .gt. nzmax) then
         nzmax = n
      endif

      plhs(1) = mxCreateSparse(m,n,nzmax,0)
      sr = mxGetPr(plhs(1))
      irs = mxGetIr(plhs(1))
      jcs = mxGetJc(plhs(1))

C     Load the sparse data.
      if (loadsparse(%val(pr), %val(sr), %val(irs), %val(jcs),
     +m,n,nzmax) .eq. 1) then
       call mexPrintf('Truncating output, input is > 20%% sparse')
      endif
      return
      end
```

This is the subroutine that `fulltosparse` calls to fill the `mxArray` with the sparse data.

```
C=================================================================
C     loadsparse.f
C    This is the subfunction called by fulltosparse that fills the
C     mxArray with the sparse data.  Your version of
C     loadsparse can operate however you would like it to on the
C     data.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.4 $
C=================================================================
```

```
C     Load sparse data subroutine.
      function loadsparse(a, b, ir, jc, m, n, nzmax)
      integer nzmax, m, n
      integer ir(*), jc(*)
      real*8 a(*), b(*)
      integer i, j, k

C     Copy nonzeros.
      k = 1
      do 100 j=1,n
C        NOTE: Sparse indexing is zero based.
         jc(j) = k-1
         do 200 i=1,m
            if (a((j-1)*m+i).ne. 0.0) then
               if (k .gt. nzmax) then
                  jc(n+1) = nzmax
                  loadsparse = 1
                  goto 300
               endif
               b(k) = a((j-1)*m+i)
C              NOTE: Sparse indexing is zero based.
               ir(k) = i-1
               k = k+1
            endif
 200     continue
 100  continue
C     NOTE: Sparse indexing is zero based.
      jc(n+1) = k-1
      loadsparse = 0
 300  return
      end
```

At the MATLAB prompt, entering

```
full = eye(5)
full =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

creates a full, 5-by-5 identity matrix. Using `fulltosparse` on the full matrix produces the corresponding sparse matrix.

```
spar = fulltosparse(full)
spar =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Calling Functions from Fortran MEX-Files

It's possible to call MATLAB functions, operators, M-files, and even other MEX-files from within your Fortran source code by using the API function `mexCallMATLAB`. This example creates an `mxArray`, passes various pointers to a subfunction to acquire data, and calls `mexCallMATLAB` to calculate the sine function and plot the results.

```
C=================================================================
C     sincall.f
C
C     Example for illustrating how to use mexCallMATLAB.
C
C     Creates an mxArray and passes its associated pointers (in
C     this demo, only pointer to its real part, pointer to number
C     of rows, pointer to number of columns) to subfunction fill()
C     to get data filled up, then calls mexCallMATLAB to calculate
C     sin function and plot the result.
C
C     NOTE: The subfunction fill() is in the file called fill.f.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.10 $
C ================================================================
```

```
C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform

      integer plhs(*), prhs(*)
      integer rhs(1), lhs(1)
      integer mxGetPr, mxCreateDoubleMatrix
C-----------------------------------------------------------------

      integer nlhs, nrhs
      integer m, n, max

C     initializition
      m = 1
      n = 1
      max = 1000

      rhs(1) = mxCreateDoubleMatrix(max, 1, 0)

C     Pass the pointer and variable and let fill() fill up data.
      call fill(%val(mxGetPr(rhs(1))), m, n, max)
      call mxSetM(rhs(1), m)
      call mxSetN(rhs(1), n)

      call mexCallMATLAB(1, lhs, 1, rhs, 'sin')
      call mexCallMATLAB(0, NULL, 1, lhs, 'plot')

C     Clean up the unfreed memory after calling mexCallMATLAB.
      call mxDestroyArray(rhs(1))
      call mxDestroyArray(lhs(1))

      return
      end
```

This is the subroutine that `sincall` calls to fill the mxArray with data.

```
C=============================================================
C     fill.f
C     This is the subfunction called by sincall that fills the
C     mxArray with data. Your version of fill can load your data
C     however you would like.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.3 $
C=============================================================

C     Subroutine for filling up data.
      subroutine fill(pr, m, n, max)
      real*8  pr(*)
      integer i, m, n, max

      m = max/2
      n = 1
      do 10 i=1,m
 10      pr(i) = i*(4*3.1415926/max)

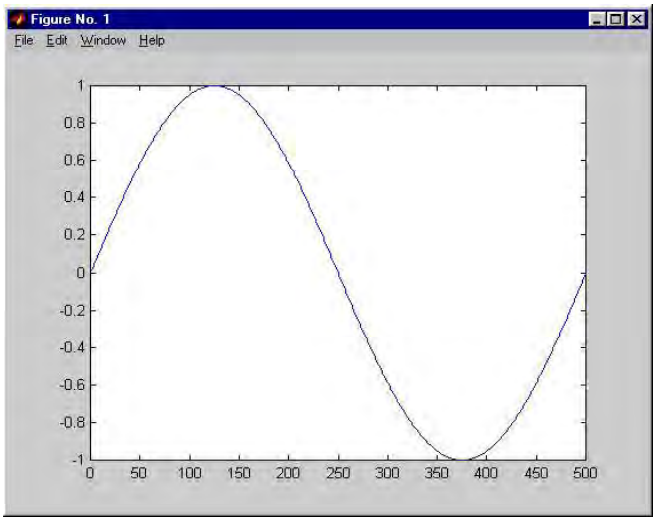
      return
      end
```

It is possible to use `mexCallMATLAB` (or any other API routine) from within your computational Fortran subroutine. Note that you can only call most MATLAB functions with double-precision data. M-functions that perform computations, like `eig`, will not work correctly with data that is not double precision.

Running this example

```
sincall
```

displays the results



Note It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. See the example below.

The following example creates an M-file that returns two variables but only assigns one of them a value.

```
function [a,b]=foo[c]
a=2*c;
```

MATLAB displays the following warning message.

```
Warning: One or more output arguments not assigned during call to
'foo'.
```

If you then call `foo` using `mexCallMATLAB`, the unassigned output variable will now be of type `mxUNKNOWN_CLASS`.

Advanced Topics

These sections cover advanced features of MEX-files that you can use when your applications require sophisticated MEX-files.

Help Files

Because the MATLAB interpreter chooses the MEX-file when both an M-file and a MEX-file with the same name are encountered in the same directory, it is possible to use M-files for documenting the behavior of your MEX-files. The MATLAB `help` command will automatically find and display the appropriate M-file when help is requested and the interpreter will find and execute the corresponding MEX-file when the function is actually invoked.

Linking Multiple Files

You can combine several source files when building MEX-files. For example,

```
mex circle.f square.o rectangle.f shapes.o
```

is a legal command that operates on the `.f` and `.o` files to create a MEX-file called `circle.`*`ext`*, where *`ext`* is the extension corresponding to the MEX-file type. The name of the resulting MEX-file is taken from the first file in the list.

You may find it useful to use a software development tool like `MAKE` to manage MEX-file projects involving multiple source files. Simply create a `MAKEFILE` that contains a rule for producing object files from each of your source files and then invoke `mex` to combine your object files into a MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Note On UNIX, you must use the `-fortran` switch to the `mex` script if you are linking Fortran objects.

Workspace for MEX-File Functions

Unlike M-file functions, MEX-file functions do not have their own variable workspace. `mexEvalString` evaluates the string in the caller's workspace. In addition, you can use the `mexGetMatrix` and `mexPutMatrix` routines to get and put variables into the caller's workspace.

Memory Management

MATLAB now implicitly destroys (by calling `mxDestroyArray`) any arrays created by a MEX-file that are not returned in the left-hand side list (`plhs()`). Consequently, any misconstructed arrays left over at the end of a MEX-file's execution have the potential to cause memory errors.

In general, we recommend that MEX-files destroy their own temporary arrays and clean up their own temporary memory. For additional information on memory management techniques, see the sections "Memory Management" on page 4-37 and "Memory Management Compatibility Issues" on page 3-36.

Ex.1049 / Page 454 of 748
Intel Corporation et al

Debugging Fortran Language MEX-Files

On most platforms, it is now possible to debug MEX-files while they are running within MATLAB. Complete source code debugging, including setting breakpoints, examining variables, and stepping through the source code line-by-line, is now available.

Note The section on "Troubleshooting" on page 3-31 provides additional information on isolating problems with MEX-files.

To debug a MEX-file from within MATLAB, you must first compile the MEX-file with the `-g` option to `mex`.

```
mex -g filename.f
```

Debugging on UNIX

You must start MATLAB from within a debugger. To do this, specify the name of the debugger you want to use with the `-D` option when starting MATLAB. For example, to use `dbx`, the UNIX debugger, type

```
matlab -Ddbx
```

Once the debugger loads MATLAB into memory, you can start it by issuing a `run` command. Now, from within MATLAB, enable MEX-file debugging by typing

```
dbmex on
```

at the MATLAB prompt. Then run the MEX-file you want to debug as you would ordinarily (either directly or by means of some other function or script). Before executing the MEX-file, you will be returned to the debugger.

You may need to tell the debugger where the MEX-file was loaded or the name of the MEX-file, in which case MATLAB will display the appropriate command for you to use. At this point, you are ready to start debugging. You can list the source code for your MEX-file and set break points in it. It is often convenient to set one at mexFunction so that you stop at the beginning of the gateway routine.

Note The name `mexFunction` may be slightly altered by the compiler (i.e., it may have an underscore appended). To determine how this symbol appears in a given MEX-file, use the UNIX command

```
nm <MEX-file> | grep -i mexfunction
```

To proceed from the breakpoint, issue a `continue` command to the debugger.

Once you hit one of your breakpoints, you can make full use of any facilities your debugger provides to examine variables, display memory, or inspect registers. Refer to the documentation provided with your debugger for information on its use.

If you are at the MATLAB prompt and want to return control to the debugger, you can issue the command

```
dbmex stop
```

which allows you to gain access to the debugger so you can set additional breakpoints or examine source code. To resume execution, issue a `continue` command to the debugger.

Debugging on Windows

Compaq Visual Fortran

If you are using the Compaq (or DIGITAL) Visual Fortran compiler, you use the Microsoft debugging environment to debug your program.

1. Start the Microsoft Visual Studio by typing at the DOS prompt

   ```
   msdev filename.dll
   ```

2. In the Microsoft environment, from the **Project** menu, select **Settings**. In the window that opens, select the **Debug** tab. This options window contains edit boxes. In the edit box labeled **Executable for debug session**, enter the full path where MATLAB resides. All other edit boxes should be empty.

3. Open the source files and set a break point on the desired line of code by right-clicking with your mouse on the line of code.

4 From the **Build** menu, select **Debug**, and click **Go**.

5 You will now be able to run your MEX-file in MATLAB and use the Microsoft debugging environment. For more information on how to debug in the Microsoft environment, see the Microsoft Development Studio documentation.

Ex.1049 / Page 457 of 748
Intel Corporation et al

Ex.1049 / Page 458 of 748
Intel Corporation et al

6

Calling MATLAB from C and Fortran Programs

The MATLAB engine library is a set of routines that allows you to call MATLAB from your own programs, thereby employing MATLAB as a computation engine. MATLAB engine programs are C or Fortran programs that communicate with a separate MATLAB process via pipes, on UNIX, and through a Component Object Model (COM) interface, on Windows. There is a library of functions provided with MATLAB that allows you to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB.

Using the MATLAB Engine (p. 6-2)	What type of applications is the MATLAB engine useful for, and what functions are available to use with it
Examples of Calling Engine Functions (p. 6-5)	Example programs that call MATLAB from C or Fortran, and that attach to an existing MATLAB session
Compiling and Linking Engine Programs (p. 6-16)	Compiling and linking on UNIX and Windows

Using the MATLAB Engine

Some of the things you can do with the MATLAB engine are:

- Call a math routine, for example, to invert an array or to compute an FFT from your own program. When employed in this manner, MATLAB is a powerful and programmable mathematical subroutine library.
- Build an entire system for a specific task, for example, radar signature analysis or gas chromatography, where the front end (GUI) is programmed in C and the back end (analysis) is programmed in MATLAB, thereby shortening development time.

The MATLAB engine operates by running in the background as a separate process from your own program. This offers several advantages:

- On UNIX, the MATLAB engine can run on your machine, or on any other UNIX machine on your network, including machines of a different architecture. Thus you could implement a user interface on your workstation and perform the computations on a faster machine located elsewhere on your network. The description of the `engOpen` function offers further information.
- Instead of requiring that all of MATLAB be linked to your program (a substantial amount of code), only a small engine communication library is needed.

The Engine Library

The engine library contains the following routines for controlling the MATLAB computation engine. Their names all begin with the three-letter prefix `eng`. These tables list all the available engine functions and their purposes.

C Engine Routines

Function	Purpose
`engOpen`	Start up MATLAB engine
`engClose`	Shut down MATLAB engine
`engGetVariable`	Get a MATLAB array from the MATLAB engine

Ex.1049 / Page 460 of 748
Intel Corporation et al

C Engine Routines

Function	Purpose
`engPutVariable`	Send a MATLAB array to the MATLAB engine
`engEvalString`	Execute a MATLAB command
`engOutputBuffer`	Create a buffer to store MATLAB text output
`engOpenSingleUse`	Start a MATLAB engine session for single, nonshared use
`engGetVisible`	Determine visibility of MATLAB engine session
`engSetVisible`	Show or hide MATLAB engine session

Fortran Engine Routines

Function	Purpose
`engOpen`	Start up MATLAB engine
`engClose`	Shut down MATLAB engine
`engGetVariable`	Get a MATLAB array from the MATLAB engine
`engPutVariable`	Send a MATLAB array to the MATLAB engine
`engEvalString`	Execute a MATLAB command
`engOutputBuffer`	Create a buffer to store MATLAB text output

The MATLAB engine also uses the `mx` prefixed API routines discussed in Chapter 4, “Creating C Language MEX-Files” and Chapter 5, “Creating Fortran MEX-Files.”

Ex.1049 / Page 461 of 748
Intel Corporation et al

Communicating with MATLAB

On UNIX, the engine library communicates with the MATLAB engine using pipes, and, if needed, `rsh` for remote execution. On Microsoft Windows, the engine library communicates with MATLAB using a Component Object Model () interface. Chapter 8, “COM and DDE Support” contains a detailed description of COM.

GUI-Intensive Applications

If you have graphical user interface (GUI) intensive applications that execute a lot of callbacks through the MATLAB engine, you should force these callbacks to be evaluated in the context of the base workspace. Use `evalin` to specify that the base workspace is to be used in evaluating the callback expression, as follows.

```
engEvalString(ep, "evalin('base', expression)")
```

Specifying the base workspace in this manner ensures that MATLAB will process the callback correctly and return results for that call.

This does not apply to computational applications that do not execute callbacks.

Ex.1049 / Page 462 of 748
Intel Corporation et al

Examples of Calling Engine Functions

This section contains examples that illustrate how to call engine functions from C and Fortran programs. The examples cover the following topics:

- "Calling MATLAB From a C Application"
- "Calling MATLAB From a Fortran Application"
- "Attaching to an Existing MATLAB Session"

It is important to understand the sequence of steps you must follow when using the engine functions. For example, before using `engPutVariable`, you must create the matrix and populate it.

After reviewing these examples, follow the instructions in "Compiling and Linking Engine Programs" on page 6-16 to build the application and test it. By building and running the application, you will ensure that your system is properly configured for engine applications.

Calling MATLAB From a C Application

This program, `engdemo.c`, illustrates how to call the engine functions from a stand-alone C program. For the Windows version of this program, see `engwindemo.c` in the `<matlab>\extern\examples\eng_mat` directory. Engine examples, like the MAT-file examples, are located in the `eng_mat` directory.

```
/*
 * engdemo.c
 *
 * This is a simple program that illustrates how to call the
 * MATLAB engine functions from a C program.
 *
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * Revision: 1.8 $
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "engine.h"
#define  BUFSIZE 256
```

```
int main()

{
   Engine *ep;
   mxArray *T = NULL, *result = NULL;
   char buffer[BUFSIZE];
   double time[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
                      8.0, 9.0};
   /*
    * Start the MATLAB engine locally by executing the string
    * "matlab".
    *
    * To start the session on a remote host, use the name of
    * the host as the string rather than \0.
    *
    * For more complicated cases, use any string with whitespace,
    * and that string will be executed literally to start MATLAB.
    */
   if (!(ep = engOpen("\0"))) {
      fprintf(stderr, "\nCan't start MATLAB engine\n");
      return EXIT_FAILURE;
   }

   /*
    * PART I
    *
    * For the first half of this demonstration, we will send data
    * to MATLAB, analyze the data, and plot the result.
    */

   /*
    * Create a variable for our data.
    */
   T = mxCreateDoubleMatrix(1, 10, mxREAL);
   memcpy((void *)mxGetPr(T), (void *)time, sizeof(time));

   /*
    * Place the variable T into the MATLAB workspace.
    */
   engPutVariable(ep, "T", T);
```

```
/*
 * Evaluate a function of time, distance = (1/2)g.*t.^2
 * (g is the acceleration due to gravity).
 */
engEvalString(ep, "D = .5.*(-9.8).*T.^2;");

/*
 * Plot the result.
 */
engEvalString(ep, "plot(T,D);");
engEvalString(ep, "title('Position vs. Time for a falling
               object');");
engEvalString(ep, "xlabel('Time (seconds)');");
engEvalString(ep, "ylabel('Position (meters)');");

/*
 * Use fgetc() to make sure that we pause long enough to be
 * able to see the plot.
 */
printf("Hit return to continue\n\n");
fgetc(stdin);

/*
 * We're done for Part I! Free memory, close MATLAB engine.
 */
printf("Done for Part I.\n");
mxDestroyArray(T);
engEvalString(ep, "close;");

/*
 * PART II
 *
 * For the second half of this demonstration, we will request
 * a MATLAB string, which should define a variable X.  MATLAB
 * will evaluate the string and create the variable.  We
 * will then recover the variable, and determine its type.
 */

/*
```

```
 * Use engOutputBuffer to capture MATLAB output, so we can
 * echo it back.
 */

engOutputBuffer(ep, buffer, BUFSIZE);
while (result == NULL) {
   char str[BUFSIZE];

   /*
    * Get a string input from the user.
    */
   printf("Enter a MATLAB command to evaluate.  This command
           should\n");
   printf("create a variable X.  This program will then
           determine\n");
   printf("what kind of variable you created.\n");
   printf("For example: X = 1:5\n");
   printf(">> ");

   fgets(str, BUFSIZE-1, stdin);

   /*
    * Evaluate input with engEvalString.
    */
   engEvalString(ep, str);

   /*
    * Echo the output from the command.  First two characters
    * are always the double prompt (>>).
    */
   printf("%s", buffer+2);

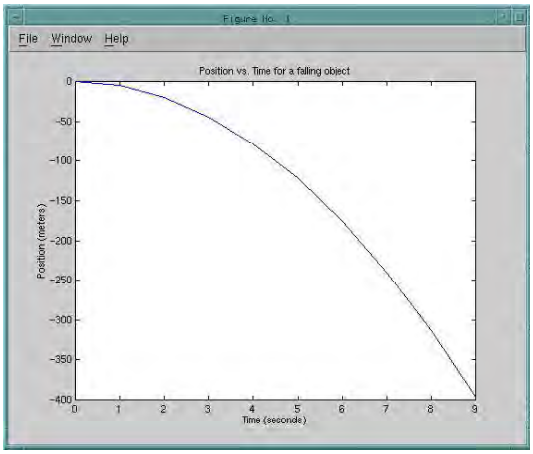
   /*
    * Get result of computation.
    */
   printf("\nRetrieving X...\n");
   if ((result = engGetVariable(ep,"X")) == NULL)
      printf("Oops! You didn't create a variable X.\n\n");
   else {
      printf("X is class %s\t\n", mxGetClassName(result));
```

```
        }
    }

    /*
     * We're done! Free memory, close MATLAB engine and exit.
     */
    printf("Done!\n");
    mxDestroyArray(result);
    engClose(ep);

    return EXIT_SUCCESS;
}
```

The first part of this program launches MATLAB and sends it data. MATLAB then analyzes the data and plots the results.



The program then continues with

```
Press Return to continue
```

Pressing **Return** continues the program.

```
Done for Part I.
Enter a MATLAB command to evaluate.  This command should
create a variable X.  This program will then determine
what kind of variable you created.
For example: X = 1:5
```

Entering `X = 17.5` continues the program execution.

```
X = 17.5

X =

   17.5000


Retrieving X...
X is class double
Done!
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

Calling MATLAB From a Fortran Application

This program, `fengdemo.f`, illustrates how to call the engine functions from a stand-alone Fortran program.

```
C=================================================================
C     fengdemo.f
C
C     This program illustrates how to call the MATLAB
C     Engine functions from a Fortran program.
C
C Copyright (c) 1984-2000 by The MathWorks, Inc.
C $Revision: 1.9 $
C=================================================================

      program main
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform

      integer engOpen, engGetVariable, mxCreateDoubleMatrix
      integer mxGetPr
      integer ep, T, D
C-----------------------------------------------------------------
```

```
C     Other variable declarations here
      double precision time(10), dist(10)
      integer engPutVariable, engEvalString, engClose
      integer temp, status
      data time / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
                  9.0, 10.0 /

      ep = engOpen('matlab ')

      if (ep .eq. 0) then
         write(6,*) 'Can''t start MATLAB engine'
         stop
      endif

      T = mxCreateDoubleMatrix(1, 10, 0)
      call mxCopyReal8ToPtr(time, mxGetPr(T), 10)
C
C     Place the variable T into the MATLAB workspace.
C
      status = engPutVariable(ep, 'T', T)

      if (status .ne. 0) then
         write(6,*) 'engPutVariable failed'
         stop
      endif
C
C     Evaluate a function of time, distance = (1/2)g.*t.^2
C     (g is the acceleration due to gravity).
C
      if (engEvalString(ep, 'D = .5.*(-9.8).*T.^2;') .ne. 0) then
         write(6,*) 'engEvalString failed'
         stop
      endif
C
C     Plot the result.
C
      if (engEvalString(ep, 'plot(T,D);') .ne. 0) then
         write(6,*) 'engEvalString failed'
         stop
      endif
```

```
      if (engEvalString(ep, 'title(''Position vs. Time'')')
      .ne. 0) then
         write(6,*) 'engEvalString failed'
         stop
      endif

      if (engEvalString(ep, 'xlabel(''Time (seconds)'')')
      .ne. 0) then
         write(6,*) 'engEvalString failed'
         stop
      endif
      if (engEvalString(ep, 'ylabel(''Position (meters)'')')
      .ne. 0) then
         write(6,*) 'engEvalString failed'
         stop
      endif
C
C     Read from console to make sure that we pause long enough to
C     be able to see the plot.
C
      print *, 'Type 0 <return> to Exit'
      print *, 'Type 1 <return> to continue'

      read(*,*) temp

      if (temp.eq.0) then
         print *, 'EXIT!'
         stop
      end if

      if (engEvalString(ep, 'close;') .ne. 0) then
         write(6,*) 'engEvalString failed'
         stop
      endif

      D = engGetVariable(ep, 'D')
      call mxCopyPtrToReal8(mxGetPr(D), dist, 10)
      print *, 'MATLAB computed the following distances:'
      print *, '  time(s)  distance(m)'
```

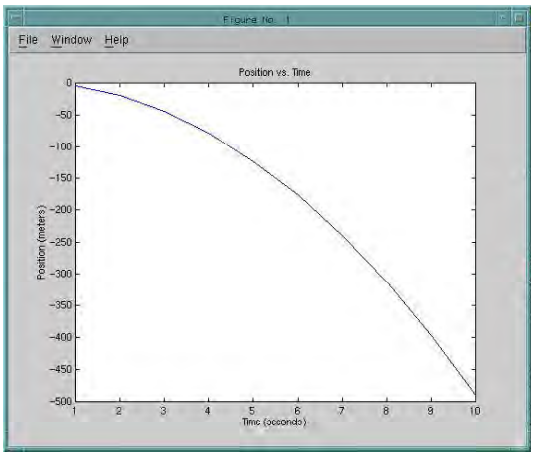
```
      do 10 i=1,10
         print 20, time(i), dist(i)
 20      format(' ', G10.3, G10.3)
 10   continue

      call mxDestroyArray(T)
      call mxDestroyArray(D)
      status = engClose(ep)

      if (status .ne. 0) then
         write(6,*) 'engClose failed'
         stop
      endif

      stop
      end
```

Executing this program launches MATLAB, sends it data, and plots the results.



The program continues with

```
Type 0 <return> to Exit
Type 1 <return> to continue
```

Entering `1` at the prompt continues the program execution.

```
1
 MATLAB computed the following distances:
   time(s)  distance(m)
   1.00     -4.90
   2.00     -19.6
   3.00     -44.1
   4.00     -78.4
   5.00     -123.
   6.00     -176.
   7.00     -240.
   8.00     -314.
   9.00     -397.
   10.0     -490.
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

Attaching to an Existing MATLAB Session

You can make a MATLAB engine program attach to a MATLAB session that is already running by starting the MATLAB session with `/Automation` in the command line. When you make a call to `engOpen`, it will then connect to this existing session. You should only call `engOpen` once, as any `engOpen` calls will now connect to this one MATLAB session.

The `/Automation` option also causes the command window to be minimized. You must open it manually.

Note For more information on the `/Automation` command line argument, see "Additional Automation Server Information" on page 8-40. For information about the Component Object Model interfaces used by MATLAB, see "Introducing MATLAB COM Integration" on page 8-2.

For example:

1. Shut down any MATLAB sessions.
2. From the **Start** button on the Windows menu bar, click **Run**.

Ex.1049 / Page 472 of 748
Intel Corporation et al

3 In the **Open** field, type

```
d:\matlab\bin\win32\matlab.exe /Automation
```

where `d:\matlab\bin\win32` represents the path to the MATLAB executable.

4 Click **OK**. This starts MATLAB.

5 In MATLAB, change directories to `$MATLAB/extern/examples/eng_mat`, where `$MATLAB` is the MATLAB root directory.

6 Compile the `engwindemo.c` example.

7 Run the `engwindemo` program by typing at the MATLAB prompt

```
!engwindemo
```

This does not start another MATLAB session, but rather uses the MATLAB session that is already open.

Note On the UNIX platform, you cannot make a MATLAB engine program use a MATLAB session that is already running.

Ex.1049 / Page 473 of 748
Intel Corporation et al

Compiling and Linking Engine Programs

To produce an executable version of an engine program, you must compile it and link it with the appropriate library. This section describes the steps required to compile and link engine programs on UNIX and Windows systems. It begins by looking at a special consideration for compilers that do not mask floating-point exceptions. Topics covered are:

- "Masking Floating-Point Exceptions"
- "Compiling and Linking on UNIX"
- "Compiling and Linking on Windows"

Masking Floating-Point Exceptions

Certain mathematical operations can result in nonfinite values. For example, division by zero results in the nonfinite IEEE value, `inf`. A floating-point exception occurs when such an operation is performed. Because MATLAB uses an IEEE model that supports nonfinite values such as `inf` and `NaN`, MATLAB disables, or *masks*, floating-point exceptions.

Some compilers do not mask floating-point exceptions by default. This causes engine programs built with such compilers to terminate when a floating-point exception occurs. Consequently, you need to take special precautions when using these compilers to mask floating-point exceptions so that your engine application will perform properly.

Note MATLAB based applications should never get floating-point exceptions. If you do get a floating-point exception, verify that any third party libraries that you link against do not enable floating-point exception handling.

The only compiler and platform on which you need to mask floating-point exceptions is the Borland C++ compiler on Windows.

Borland C++ Compiler on Windows

To mask floating-point exceptions when using the Borland C++ compiler on the Windows platform, you must add some code to your program. Include the following at the beginning of your `main()` or `WinMain()` function, before any calls to MATLAB API functions.

Ex.1049 / Page 474 of 748
Intel Corporation et al

```
#include <float.h>
    .
    .
    .
_control87(MCW_EM,MCW_EM);
    .
    .
    .
```

Compiling and Linking on UNIX

Under UNIX at runtime, you must tell the system where the API shared libraries reside. These sections provide the necessary UNIX commands depending on your shell and system architecture.

Setting Runtime Library Path

In C shell, the command to set the library path is

```
setenv LD_LIBRARY_PATH
<matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
```

In Bourne shell, the commands to set the library path are

```
LD_LIBRARY_PATH=<matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

where `<matlab>` is the MATLAB root directory and `$Arch` is your system architecture (`alpha`, `glnx86`, `sgi`, `sol2`, `hp700`, or `ibm_rs`).

Note that the environment variable (LD_LIBRARY_PATH in this example) varies on several platforms. This table lists the different environment variable names you should use on these systems.

Environment Variable Names

Architecture	Environment Variable
HP700	SHLIB_PATH
IBM RS/6000	LIBPATH

It is convenient to place these commands in a startup script such as `~/.cshrc` for C shell or `~/.profile` for Bourne shell.

Compiling and Linking

MATLAB provides an options file, `engopts.sh`, that lets you use the `mex` script to easily compile and link engine applications. For example, to compile and link the `engdemo.c` example, you can use

```
mex -f <matlab>/bin/engopts.sh <pathname>/engdemo.c
```

where `<pathname>` specifies the complete path to the specified file.

If you need to modify the options file for your particular compiler or platform, use the `-v` switch to view the current compiler and linker settings and then make the appropriate changes in a local copy of the `engopts.sh` file.

Compiling and Linking on Windows

To compile and link engine programs, use the `mex` script with an engine options file. Table 3-2, Options Files, on page 3-16 lists the engine options files included in this release. All of these options files are located in `<matlab>\bin\win32\mexopts`.

As an example, to compile and link the stand-alone engine application `engwindemo.c` on Windows using MSVC (Version 5.0), use

```
mex -f <matlab>\bin\win32\mexopts\msvc50engmatopts.bat ...
<pathname>\engwindemo.c
```

where `<pathname>` specifies the complete path to the specified file. If you need to modify the options file for your particular compiler, use the `-v` switch to view the current compiler and linker settings and then make the appropriate changes in the options file.

Ex.1049 / Page 476 of 748
Intel Corporation et al

7

Calling Java from MATLAB

This chapter describes how to use the MATLAB interface to Java classes and objects. This MATLAB capability enables you to conveniently bring Java classes into the MATLAB environment, to construct objects from those classes, to call methods on the Java objects, and to save Java objects for later reloading — all accomplished with MATLAB functions and commands.

Using Java from MATLAB: An Overview (p. 7-2)	How you can benefit from using the MATLAB Java interface
Bringing Java Classes into MATLAB (p. 7-4)	Using Java built-in, third-party, or your own classes
Creating and Using Java Objects (p. 7-9)	Constructing and working with Java objects
Invoking Methods on Java Objects (p. 7-18)	Calling syntax, static methods, querying MATLAB about methods
Working with Java Arrays (p. 7-28)	How MATLAB represents Java arrays and how to work with them
Passing Data to a Java Method (p. 7-46)	How to pass MATLAB data types into Java
Handling Data Returned from a Java Method (p. 7-56)	How to handle data types returned by Java
Introduction to Programming Examples (p. 7-62)	Introduction and links to sample programs that use the MATLAB interface to Java
Example – Reading a URL (p. 7-63)	Open a connection to a Web site and read text from the site using a buffered stream reader
Example – Finding an Internet Protocol Address (p. 7-66)	Call methods on an `InetAddress` object to get hostname and IP address information
Example – Communicating Through a Serial Port (p. 7-68)	Create a `SerialPort` object and configure the port using methods provided by that class
Example – Creating and Using a Phone Book (p. 7-73)	Create a phonebook using a data dictionary that operates using key/value pairs in a hash table

Using Java from MATLAB: An Overview

Java Interface Is Integral to MATLAB

Every installation of MATLAB includes a Java Virtual Machine (JVM), so that you can use the Java interpreter via MATLAB commands, and you can create and run programs that create and access Java objects. For information on MATLAB installation, see the MATLAB installation documentation for your platform.

Benefits of the MATLAB Java Interface

The MATLAB Java interface enables you to:

- Access Java API (application programming interface) class packages that support essential activities such as I/O and networking. For example, the `URL` class provides convenient access to resources on the internet
- Access third-party Java classes
- Easily construct Java objects in MATLAB
- Call Java object methods, using either Java or MATLAB syntax
- Pass data between MATLAB variables and Java objects

Who Should Use the MATLAB Java Interface

The MATLAB Java interface is intended for all MATLAB users who want to take advantage of the special capabilities of the Java programming language. For example:

- You need to access, from MATLAB, the capabilities of available Java classes.
- You are familiar with object-oriented programming in Java or in another language, such as C++.
- You are familiar with MATLAB object-oriented classes, or with MATLAB MEX-files.

To Learn More About Java Programming

For a complete description of the Java language and for guidance in object-oriented software design and programming, you'll need to consult outside resources. For example, these recently published books may be helpful:

Ex.1049 / Page 478 of 748
Intel Corporation et al

- *Java in a Nutshell* (Second Edition), by David Flanagan
- *Teach Yourself Java in 21 Days*, by Lemay and Perkins

Another place to find information is the JavaSoft Web site.

```
http://www.javasoft.com
```

For other suggestions on object-oriented programming resources, see:

- *Object Oriented Software Construction*, by Bertrand Meyer
- *Object Oriented Analysis and Design with Applications*, by Grady Booch

Platform Support for the Java Virtual Machine

To find out which version of the Java Virtual Machine (JVM) is being used by MATLAB on your platform, type the following at the MATLAB prompt.

```
version -java
```


Ex.1049 / Page 479 of 748
Intel Corporation et al

Bringing Java Classes into MATLAB

You can draw from an extensive collection of existing Java classes or create your own class definitions to use with MATLAB. This section explains how to go about finding the class definitions that you need or how to create classes of your own design. Once you have the classes you need, defined in either individual `.class` files, packages, or Java Archive files, you will see how to make them available within the MATLAB environment.

This section addresses the following topics:

- "Sources of Java Classes"
- "Defining New Java Classes" on page 7-4
- "Making Java Classes Available to MATLAB" on page 7-5
- "Loading Java Class Definitions" on page 7-6
- "Simplifying Java Class Names" on page 7-7

Sources of Java Classes

There are three main sources of Java classes that you can use in MATLAB:

- **Java built-in classes**

  The Java language includes general-purpose class packages, such as `java.awt`. See your Java language documentation for descriptions of these packages.
- **Third-party classes**

  There are a number of packages of special-purpose Java classes that you can use.
- **User-defined classes**

  You can define new Java classes or subclasses of existing classes. You need to use a Java development environment to do this, as explained in the following section.

Defining New Java Classes

To define new Java classes and subclasses of existing classes, you must use a Java development environment, external to MATLAB, that supports Java version 1.1. You can download the development kit that supports the version

Ex.1049 / Page 480 of 748
Intel Corporation et al

1.1.8 Java Virtual Machine from the web site at Sun Microsystems, (`http://www.java.sun.com/jdk/`). The Sun site also provides documentation for the Java language and classes that you will need for development.

After you create class definitions in `.java` files, use your Java compiler to produce `.class` files from them. The next step is to make the class definitions in those `.class` files available for you to use in MATLAB.

Making Java Classes Available to MATLAB

To make your third-party and user-defined Java classes available in MATLAB, place them on your MATLAB path by adding their locations to the file `classpath.txt`. For more information on `classpath.txt`, see "Finding and Editing classpath.txt" on page 7-6.

Making Individual (Unpackaged) Classes Available

To make individual classes (classes not part of a package) available in MATLAB, add to `classpath.txt` an entry containing the full path to the directory you want to use for the `.class` file(s).

For example, to make available your compiled Java classes in the file `d:\work\javaclasses\test.class`, add the following entry to your `classpath.txt` file.

```
d:\work\javaclasses
```

Making Entire Packages Available

To access one or more classes belonging to a package, you need to make the entire package available to MATLAB. Do this by adding to `classpath.txt` the full path to the *parent directory of the highest-level directory* of the package path. This directory is the first component in the package name.

For example, if your Java class package `com.mw.tbx.ini has` its classes in directory `d:\work\com\mw\tbx\ini`, add the following entry to your `classpath.txt`.

```
d:\work
```

Making Classes in a JAR File Available

You can use the `jar` (Java Archive) tool to create a JAR file, containing multiple Java classes and packages in a compressed ZIP format. For information on `jar`

and JAR files, consult your Java development documentation or the JavaSoft web site. See also “To Learn More About Java Programming” on page 7-2.

To make the contents of a JAR file available for use in MATLAB, add to `classpath.txt` the full path, *including full filename*, for the JAR file.

Note Note that the `classpath.txt` requirement for JAR files is different than that for `.class` files and packages, for which you do not specify any filename.

For example, to make available the JAR file `e:\java\classes\utilpkg.jar`, add the following to your `classpath.txt`.

```
e:\java\classes\utilpkg.jar
```

Finding and Editing classpath.txt

To make your Java classes available, you can edit the default `classpath.txt`, which is in the directory `toolbox/local`. Or, you can copy the default `classpath.txt` from this directory to your own startup directory where the changes you make to it do not affect anyone else.

To find the `classpath.txt` that is used by your MATLAB environment, use the MATLAB `which` function.

```
which classpath.txt
```

To edit either the default file or the copy you have made in your own directory, enter the following command in MATLAB.

```
edit classpath.txt
```

MATLAB reads `classpath.txt` only upon startup. So, if you edit `classpath.txt` or change your `.class` files while MATLAB is running, you must restart MATLAB to put those changes into effect.

Loading Java Class Definitions

Normally, MATLAB loads a Java class automatically when your code first uses it, (for example, when you call its constructor). However, there is one exception that you should be aware of.

When you use the `which` function on methods defined by Java classes, the function only acts on the classes currently loaded into the MATLAB working environment. In contrast, `which` always operates on MATLAB classes, whether or not they are loaded.

Determining Which Classes Are Loaded

At any time during a MATLAB session, you can obtain a listing of all the Java classes that are currently loaded. To do so, you use the `inmem` function, in the following form.

```
[M,X,J] = inmem
```

This function returns the list of Java classes in output argument `J`. (It also returns in `M` the names of all currently loaded M-files, and in `X` the names of all currently loaded MEX-files.)

Here's a sample of output from the `inmem` function.

```
[m,x,j] = inmem
m =
    'isequal'
    'isunix'
    'fullfile'
    'filesep'
    .
    .
    .
    'matlabrc'
x =
    'getprofl'
j =
    'java.awt.Frame'
    'com.mathworks.ide.desktop.MLDesktop'
```

Simplifying Java Class Names

Your MATLAB commands can refer to any Java class by its fully qualified name, which includes its package name. For example, the following are fully qualified names:

- `java.lang.String`
- `java.util.Enumeration`

A fully qualified name can be rather long, making commands and functions, such as constructors, cumbersome to edit and to read. You can refer to classes by the class name alone (without a package name) if you, first, import the fully qualified name into MATLAB.

The `import` command has the following forms.

```
import pkg_name.*               % Import all classes in package
import pkg_name1.* pkg_name2.*  % Import multiple packages
import class_name               % Import one class
import                          % Display current import list
L = import                      % Return current import list
```

MATLAB adds all classes that you `import` to a list called the *import list*. You can see what classes are on that list by typing `import`, without any arguments. Your code can refer to any class on the list by class name alone.

When called from a function, `import` adds the specified classes to the import list in effect for that function. When invoked at the command prompt, `import` uses the base import list for your MATLAB environment.

For example, suppose a function contains the following statements.

```
import java.lang.String
import java.util.* java.awt.*
import java.util.Enumeration
```

Code that follows the `import` statements above can now refer to the `String`, `Frame`, and `Enumeration` classes without using the package names.

```
str = String('hello');     % Create java.lang.String object
frm = Frame;               % Create java.awt.Frame object
methods Enumeration        % List java.util.Enumeration methods
```

To `clear` the list of imported Java classes, invoke the command

```
clear import
```

Creating and Using Java Objects

In MATLAB, you create a Java object by calling one of the constructors of that class. You then use commands and programming statements to perform operations on these objects. You can also save your Java objects to a MAT-file and, in subsequent sessions, reload them into MATLAB.

This section addresses the following topics:

- "Constructing Java Objects"
- "Concatenating Java Objects" on page 7-11
- "Saving and Loading Java Objects to MAT-Files" on page 7-13
- "Finding the Public Data Fields of an Object" on page 7-14
- "Accessing Private and Public Data" on page 7-15
- "Determining the Class of an Object" on page 7-17

Constructing Java Objects

You construct Java objects in MATLAB by calling the Java class constructor, which has the same name as the class. For example, the following constructor creates a `Frame` object with the title `'Frame A'` and the other properties with their default values.

```
frame = java.awt.Frame('Frame A');
```

Displaying the new object `frame` shows the following.

```
frame =
java.awt.Frame[frame0,0,0,0x0,invalid,hidden,layout=
java.awt.BorderLayout,resizable,title=Frame A]
```

All of the programming examples in this chapter contain Java object constructors. For example, the sample code for Reading a URL creates a `java.net.URL` object with the constructor

```
url = java.net.URL(...
     'http://archive.ncsa.uiuc.edu/demoweb/')
```


Ex.1049 / Page 485 of 748
Intel Corporation et al

Using the javaObject Function

Under certain circumstances, you may need to use the `javaObject` function to construct a Java object. The following syntax invokes the Java constructor for class, `class_name`, with the argument list that matches `x1,...,xn`, and returns a new object, `J`.

```
J = javaObject('class_name',x1,...,xn);
```

For example, to construct and return a Java object of class `java.lang.String`, you use

```
strObj = javaObject('java.lang.String','hello');
```

Using the `javaObject` function enables you to:

- Use classes that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify the class for an object at run-time, for example, as input from an application user

The default MATLAB constructor syntax requires that no segment of the input class name be longer than `namelengthmax` characters. (A *class name segment* is any portion of the class name before, between, or after a dot. For example, there are three segments in class, `java.lang.String`.) Any class name segment that exceeds `namelengthmax` characters is truncated by MATLAB. In the rare case where you need to use a class name of this length, you must use `javaObject` to instantiate the class.

The `javaObject` function also allows you to specify the Java class for the object being constructed at run-time. In this situation, you call `javaObject` with a string variable in place of the class name argument.

```
class  = 'java.lang.String';
text = 'hello';
strObj = javaObject(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, you would use

```
strObj = java.lang.String('hello');
```


Ex.1049 / Page 486 of 748
Intel Corporation et al

Note Typically, you will not need to use `javaObject`. The default MATLAB syntax for instantiating a Java class is somewhat simpler and is preferable for most applications. Use `javaObject` primarily for the two cases described above.

Java Objects Are References in MATLAB

In MATLAB, Java objects are *references* and do not adhere to MATLAB copy-on-assignment and pass-by-value rules. For example,

```
origFrame = java.awt.Frame;
setSize(origFrame, 800, 400);
newFrameRef = origFrame;
```

In the second statement above, the variable `newFrameRef` is a second reference to `origFrame`, not a copy of the object. In any code following the example above, any change to the object at `newFrameRef` also changes the object at `origFrame`. This effect occurs whether the object is changed by MATLAB code, or by Java code.

The following example shows that `origFrame` and `newFrameRef` are both references to the same entity. When the size of the frame is changed via one reference (`newFrameRef`), the change is reflected through the other reference (`origFrame`), as well.

```
setSize(newFrameRef, 1000, 800);

getSize(origFrame)
ans =
java.awt.Dimension[width=1000,height=800]
```

Concatenating Java Objects

You can concatenate Java objects in the same way that you concatenate native MATLAB data types. You use either the `cat` function or the square bracket operators to tell MATLAB to assemble the enclosed objects into a single object.

Ex.1049 / Page 487 of 748
Intel Corporation et al

Concatenating Objects of the Same Class

If all of the objects being operated on are of the same Java class, then the concatenation of those objects produces an array of objects from the same class.

In the following example, the `cat` function concatenates two objects of the class `java.awt.Point`. The class of the result is also `java.awt.Point`.

```
point1 = java.awt.Point(24,127);
point2 = java.awt.Point(114,29);

cat(1, point1, point2)
ans =
java.awt.Point[]:
    [1x1 java.awt.Point]
    [1x1 java.awt.Point]
```

Concatenating Objects of Unlike Classes

When you concatenate objects of unlike classes, MATLAB finds one class from which all of the input objects inherit, and makes the output an instance of this class. MATLAB selects the lowest common parent in the Java class hierarchy as the output class.

For example, concatenating objects of `java.lang.Byte`, `java.lang.Integer`, and `java.lang.Double` yields an object of `java.lang.Number`, since this is the common parent to the three input classes.

```
byte = java.lang.Byte(127);
integer = java.lang.Integer(52);
double = java.lang.Double(7.8);

[byte; integer; double]

ans =
java.lang.Number[]:
    [   127]
    [    52]
    [7.8000]
```

If there is no common, lower level parent, then the resultant class is `java.lang.Object`, which is the root of the entire Java class hierarchy.

```
byte = java.lang.Byte(127);
point = java.awt.Point(24,127);

[byte; point]

ans =
java.lang.Object[]:
    [               127]
    [1x1 java.awt.Point]
```

Saving and Loading Java Objects to MAT-Files

Use the MATLAB `save` function to save a Java object to a MAT-file. Use the `load` function to load it back into MATLAB from that MAT-file. To save a Java object to a MAT-file, and to load the object from the MAT-file, make sure that the object and its class meet all of the following criteria:

- The class implements the Serializable interface (part of the Java API), either directly or by inheriting it from a parent class. Any embedded or otherwise referenced objects must also implement Serializable.
- The definition of the class is not changed between saving and loading the object. Any change to the data fields or methods of a class prevents the loading (deserialization) of an object that was constructed with the old class definition.
- Either the class does not have any transient data fields, or the values in transient data fields of the object to be saved are not significant. Values in transient data fields are never saved with the object.

If you define your own Java classes, or subclasses of existing classes, you can follow the criteria above to enable objects of the class to be saved and loaded in MATLAB. For details on defining classes to support serialization, consult your Java development documentation. (See also “To Learn More About Java Programming” on page 7-2).

Ex.1049 / Page 489 of 748
Intel Corporation et al

Finding the Public Data Fields of an Object

To list the public fields that belong to a Java object, use the `fieldnames` function, which takes either of these forms.

```
names = fieldnames(obj)
names = fieldnames(obj,'-full')
```

Calling `fieldnames` without `'-full'` returns the names of all the data fields (including inherited) on the object. With the `'-full'` qualifier, `fieldnames` returns the full description of the data fields defined for the object, including type, attributes, and inheritance information.

Suppose, for example, that you constructed a `Frame` object with

```
frame = java.awt.Frame;
```

To obtain the full description of the data fields on `frame`, you could use the command

```
fieldnames(frame,'-full')
```

Sample output from this command follows.

```
ans =
'static final int WIDTH
    % Inherited from java.awt.image.ImageObserver'
'static final int HEIGHT
    % Inherited from java.awt.image.ImageObserver'
[1x74 char]
'static final int SOMEBITS
    % Inherited from java.awt.image.ImageObserver'
'static final int FRAMEBITS
    % Inherited from java.awt.image.ImageObserver'
'static final int ALLBITS
    % Inherited from java.awt.image.ImageObserver'
'static final int ERROR
    % Inherited from java.awt.image.ImageObserver'
'static final int ABORT
    % Inherited from java.awt.image.ImageObserver'
'static final float TOP_ALIGNMENT
    % Inherited from java.awt.Component'
'static final float CENTER_ALIGNMENT
    % Inherited from java.awt.Component'
```


Ex.1049 / Page 490 of 748
Intel Corporation et al

```
'static final float BOTTOM_ALIGNMENT
    % Inherited from java.awt.Component'
'static final float LEFT_ALIGNMENT
    % Inherited from java.awt.Component'
'static final float RIGHT_ALIGNMENT
    % Inherited from java.awt.Component'
.
.
.
```

Accessing Private and Public Data

Java API classes provide accessor methods you can use to read from and, where allowed, to modify *private* data fields. These are sometimes referred to as *get* and *set* methods, respectively.

Some Java classes have *public* data fields, which your code can read or modify directly. To access these fields, use the syntax `object.field`.

Examples

The `java.awt.Frame` class provides an example of access to both private and public data fields. This class has the read accessor method `getSize`, which returns a `java.awt.Dimension` object. The `Dimension` object has data fields `height` and `width`, which are public and therefore directly accessible. The following example shows MATLAB commands accessing this data.

```
frame = java.awt.Frame;
frameDim = getSize(frame);
height = frameDim.height;
frameDim.width = 42;
```

The programming examples in this chapter also contain calls to data field accessors. For instance, the sample code for “Example – Finding an Internet Protocol Address” on page 7-66 uses calls to accessors on a `java.net.InetAddress` object.

```
hostname = address.getHostName;
ipaddress = address.getHostAddress;
```

Accessing Data from a Static Field

In Java, a static data field is a field that applies to an entire class of objects. Static fields are most commonly accessed in relation to the class name itself in Java. For example, the code below accesses the `WIDTH` field of the `Frame` class by referring to it in relation to the package and class names, `java.awt.Frame`, rather than an object instance.

```
width = java.awt.Frame.WIDTH;
```

In MATLAB, you can use that same syntax. Or you can refer to the `WIDTH` field in relation to an instance of the class. The example shown here creates an instance of `java.awt.Frame` called `frameObj`, and then accesses the `WIDTH` field using the name `frameObj` rather than the package and class names.

```
frame = java.awt.Frame('Frame A');

width = frame.WIDTH
width =
    1
```

Assigning to a Static Field

You can assign values to static Java fields by using a static `set` method of the class, or by making the assignment in reference to an instance of the class. For more information, see the previous section, “Accessing Data from a Static Field”. You can assign `value` to the field `staticFieldName` in the example below by referring to this field in reference to an instance of the class.

```
objectName = java.className;
objectName.staticFieldName = value;
```

Note MATLAB does not allow assignment to static fields using the class name itself.

Ex.1049 / Page 492 of 748
Intel Corporation et al

Determining the Class of an Object

To find the class of a Java object, use the query form of the MATLAB function, `class`. After execution of the following example, `frameClass` contains the name of the package and class that Java object `frame` instantiates.

```
frameClass = class(frame)
frameClass =
java.awt.Frame
```

Because this form of `class` also works on MATLAB objects, it does not, in itself, tell you whether it is a Java class. To determine the type of class, use the `isjava` function, which has the form

```
x = isjava(obj)
```

`isjava` returns `1` if `obj` is Java, and `0` if it is not.

```
isjava(frame)
ans =
    1
```

To find out whether or not an object is an instance of a specified class, use the `isa` function, which has the form

```
x = isa(obj, 'class_name')
```

`isa` returns `1` if `obj` is an instance of the class named '*`class_name`*', and `0` if it is not. Note that '*`class_name`*' can be a MATLAB built-in or user-defined class, as well as a Java class.

```
isa(frame, 'java.awt.Frame')
ans =
    1
```

Invoking Methods on Java Objects

This section explains how to invoke an object's methods in MATLAB. It also covers how to obtain information related to the methods that you're using and how MATLAB handles certain types of nonstandard situations.

This section addresses the following topics:

- "Using Java and MATLAB Calling Syntax"
- "Invoking Static Methods on Java Classes" on page 7-20
- "Obtaining Information About Methods" on page 7-21
- "Java Methods That Affect MATLAB Commands" on page 7-25
- "How MATLAB Handles Undefined Methods" on page 7-26
- "How MATLAB Handles Java Exceptions" on page 7-27

Using Java and MATLAB Calling Syntax

To call methods on Java objects, you can use the Java syntax

```
object.method(arg1,...,argn)
```

In the following example, `frame` is the `java.awt.Frame` object created above, and `getTitle` and `setTitle` are methods of that object.

```
frame.setTitle('Sample Frame')

title = frame.getTitle
title =
Sample Frame
```

Alternatively, you can call Java object (nonstatic) methods with the MATLAB syntax

```
method(object, arg1,...,argn)
```

With MATLAB syntax, the `java.awt.Frame` example above becomes

```
setTitle(frame, 'Sample Frame')

title = getTitle(frame)
title =
Sample Frame
```

All of the programming examples in this chapter contain invocations of Java object methods. For example, the code for Reading a URL contains a call, using MATLAB syntax, to the `openStream` method on a `java.net.URL` object, `url`.

```
is = openStream(url)
```

In another example, the code for "Example – Creating and Using a Phone Book" on page 7-73 contains a call, using Java syntax, to the `load` method on a `java.utils.Properties` object, `pb_htable`.

```
pb_htable.load(FIS);
```

Using the javaMethod Function on Nonstatic Methods

Under certain circumstances, you may need to use the `javaMethod` function to call a Java method. The following syntax invokes the method, `method_name`, on Java object `J` with the argument list that matches `x1,...,xn`. This returns the value `X`.

```
X = javaMethod('method_name',J,x1,...,xn);
```

For example, to call the `startsWith` method on a `java.lang.String` object passing one argument, use

```
gAddress = java.lang.String('Four score and seven years ago');
str = java.lang.String('Four score');

javaMethod('startsWith', gAddress, str)
ans =
     1
```

Using the `javaMethod` function enables you to

- Use methods that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify the method you want to invoke at run-time, for example, as input from an application user.

The only way to invoke a method whose name is longer than `namelengthmax` characters is to use `javaMethod`. The Java and MATLAB calling syntax does not accept method names of this length.

With `javaMethod`, you can also specify the method to be invoked at run-time. In this situation, your code calls `javaMethod` with a string variable in place of the `method_name` argument. When you use `javaMethod` to invoke a static method, you can also use a string variable in place of the class name argument.

Note Typically, you will not need to use `javaMethod`. The default MATLAB syntax for invoking a Java method is somewhat simpler and is preferable for most applications. Use `javaMethod` primarily for the two cases described above.

Invoking Static Methods on Java Classes

To invoke a `static` method on a Java class, use the Java invocation syntax

```
class.method(arg1,...,argn)
```

For example, call the `isNaN` static method on the `java.lang.Double` class.

```
java.lang.Double.isNaN(2.2)
```

Alternatively, you can apply static method names to instances of a class. In this example, the `isNaN` static method is referenced in relation to the `dblObject` instance of the `java.lang.Double` class.

```
dblObject = java.lang.Double(2.2);

dblObject.isNaN
ans =
     0
```

Several of the programming examples in this chapter contain examples of static method invocation. For example, the code for Communicating Through a Serial Port contains a call to static method `getPortIdentifier` on Java class `javax.comm.CommPortIdentifier`.

```
commPort =
javax.comm.CommPortIdentifier.getPortIdentifier('COM1');
```


Ex.1049 / Page 496 of 748
Intel Corporation et al

Using the javaMethod Function on Static Methods

The `javaMethod` function was introduced in section "Using the javaMethod Function on Nonstatic Methods" on page 7-19. You can also use this function to call static methods.

The following syntax invokes the static method, `method_name`, in class, `class_name`, with the argument list that matches `x1,...,xn`. This returns the value `X`.

```
X = javaMethod('method_name','class_name',x1,...,xn);
```

For example, to call the static `isNaN` method of the `java.lang.Double` class on a double value of 2.2, you use

```
javaMethod('isNaN','java.lang.Double',2.2);
```

Using the `javaMethod` function to call static methods enables you to:

- Use methods that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify method and class names at run-time, for example, as input from an application user

Obtaining Information About Methods

MATLAB offers several functions to help obtain information related to the Java methods you are working with. You can request a list of all of the methods that are implemented by any class. The list may be accompanied by other method information such as argument types and exceptions. You can also request a listing of every Java class that you loaded into MATLAB that implements a specified method.

Methodsview: Displaying a Listing of Java Methods

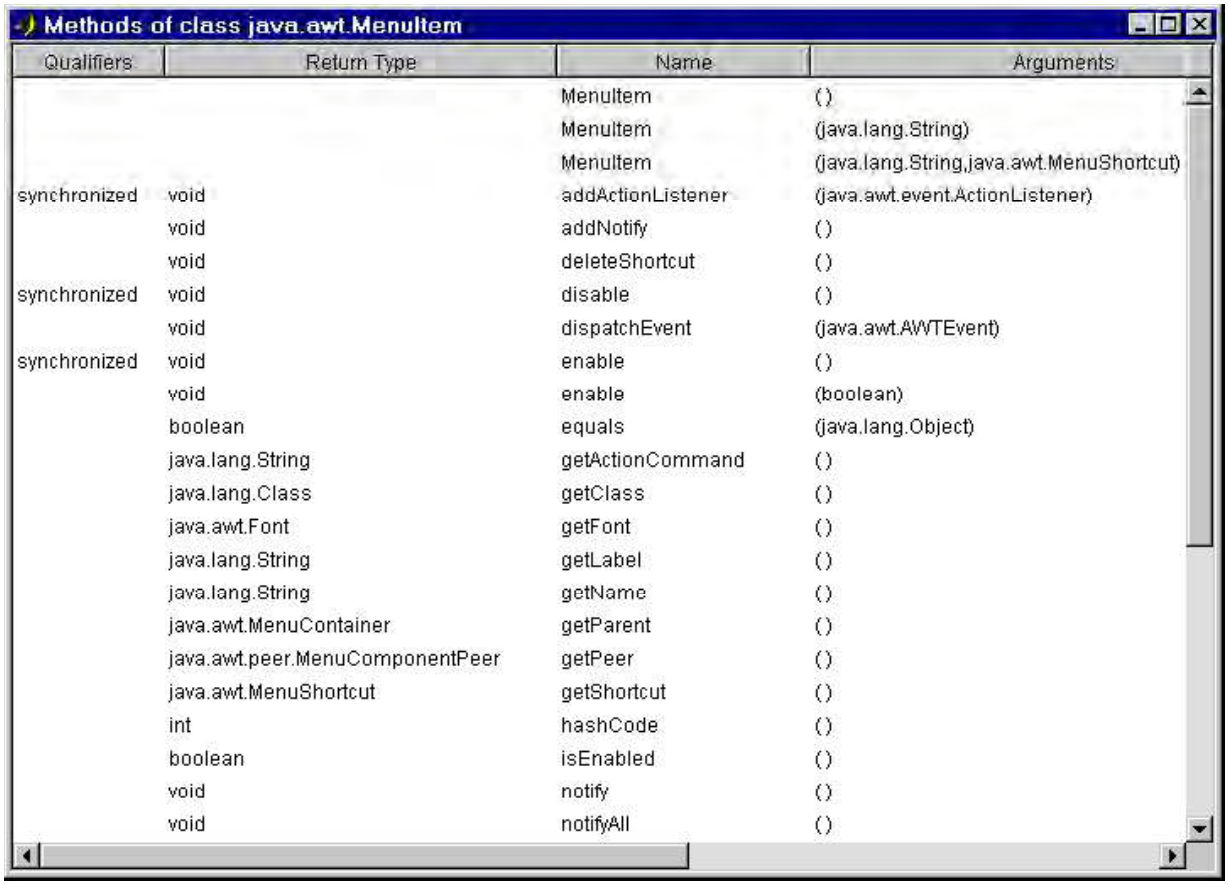
If you want to know what methods are implemented by a particular Java (or MATLAB) class, use the `methodsview` function in MATLAB. Specify the class name (along with its package name, for Java classes) in the command line. If you have imported the package that defines this class, then the class name alone will suffice.

Ex.1049 / Page 497 of 748
Intel Corporation et al

The following command lists information on all methods in the `java.awt.MenuItem` class.

```
methodsview java.awt.MenuItem
```

A new window appears, listing one row of information for each method in the class. This is what the `methodsview` display looks like. The fieldnames shown at the top of the window are described following the figure.



Qualifiers	Return Type	Name	Arguments
		MenuItem	()
		MenuItem	(java.lang.String)
		MenuItem	(java.lang.String,java.awt.MenuShortcut)
synchronized	void	addActionListener	(java.awt.event.ActionListener)
	void	addNotify	()
	void	deleteShortcut	()
synchronized	void	disable	()
	void	dispatchEvent	(java.awt.AWTEvent)
synchronized	void	enable	()
	void	enable	(boolean)
	boolean	equals	(java.lang.Object)
	java.lang.String	getActionCommand	()
	java.lang.Class	getClass	()
	java.awt.Font	getFont	()
	java.lang.String	getLabel	()
	java.lang.String	getName	()
	java.awt.MenuContainer	getParent	()
	java.awt.peer.MenuComponentPeer	getPeer	()
	java.awt.MenuShortcut	getShortcut	()
	int	hashCode	()
	boolean	isEnabled	()
	void	notify	()
	void	notifyAll	()

Each row in the window displays up to six fields of information describing the method. The table below lists the fields displayed in the `methodsview` window along with a description and examples of each field type.

Table 7-1: Fields Displayed in the Methodsview Window

Field Name	Description	Examples
Qualifiers	Method type qualifiers	abstract, synchronized
Return Type	Data type returned by the method	`void`, `java.lang.String`
Name	Method name	`addActionListener`, `dispatchEvent`
Arguments	Arguments passed to method	`boolean`, `java.lang.Object`
Other	Other relevant information	throws `java.io.IOException`
Parent	Parent of the specified class	`java.awt.MenuComponent`

Using the Methods Function on Java Classes

In addition to `methodsview`, the MATLAB `methods` function, that returns information on methods of MATLAB classes, will also work on Java classes. You can use any of the following forms of this command.

```
methods class_name
methods class_name -full
n = methods('class_name')
n = methods('class_name','-full')
```

Use `methods` without the `'-full'` qualifier to return the names of all the methods (including inherited methods) of the class. Names of overloaded methods are listed only once.

With the `'-full'` qualifier, `methods` returns a listing of the method names (including inherited methods) along with attributes, argument lists, and inheritance information on each. Each overloaded method is listed separately.

Ex.1049 / Page 499 of 748
Intel Corporation et al

For example, display a full description of all methods of the `java.awt.Dimension` object.

```
methods java.awt.Dimension -full

Methods for class java.awt.Dimension:
Dimension()
Dimension(java.awt.Dimension)
Dimension(int,int)
java.lang.Class getClass()  % Inherited from java.lang.Object
int hashCode()  % Inherited from java.lang.Object
boolean equals(java.lang.Object)
java.lang.String toString()
void notify()  % Inherited from java.lang.Object
void notifyAll()  % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait() throws java.lang.InterruptedException
   % Inherited from java.lang.Object
java.awt.Dimension getSize()
void setSize(java.awt.Dimension)
void setSize(int,int)
```

Determining What Classes Define a Method

You can use the `which` function to display the fully qualified name (package and class name) of a method implemented by a *loaded* Java class. With the `-all` qualifier, the `which` function finds all classes with a method of the name specified.

Suppose, for example, that you want to find the package and class name for the `concat` method, with the `String` class currently loaded. Use the command

```
which concat
java.lang.String.concat  % String method
```

If the `java.lang.String` class has not been loaded, the same `which` command would give the output

```
which concat
concat not found.
```

If you use `which -all` for the method `equals`, with the `String` and `java.awt.Frame` classes loaded, you see the following display.

```
which -all equals
java.lang.String.equals                      % String method
java.awt.Frame.equals                        % Frame method
com.mathworks.ide.desktop.MLDesktop.equals  % MLDesktop method
```

The `which` function operates differently on Java classes than it does on MATLAB classes. MATLAB classes are always displayed by `which`, whether or not they are loaded. This is not true for Java classes. You can find out which Java classes are currently loaded by using the command `[m,x,j]=inmem`, described in "Determining Which Classes Are Loaded" on page 7-7.

For a description of how Java classes are loaded, see "Making Java Classes Available to MATLAB" on page 7-5.

Java Methods That Affect MATLAB Commands

MATLAB commands that operate on Java objects and arrays make use of the methods that are implemented within, or inherited by, these objects' classes. There are some MATLAB commands that you can alter somewhat in behavior by changing the Java methods that they rely on.

Changing the Effect of disp and display

You can use the `disp` function to display the value of a variable or an expression in MATLAB. Terminating a command line without a semicolon also calls the `disp` function. You can also use `disp` to display a Java object in MATLAB.

When `disp` operates on a Java object, MATLAB formats the output using the `toString` method of the class to which the object belongs. If the class does not implement this method, then an inherited `toString` method is used. If no intermediate ancestor classes define this method, it uses the `toString` method defined by the `java.lang.Object` class. You can override inherited `toString` methods in classes that you create by implementing such a method within your class definition. In this way, you can change the way MATLAB displays information regarding the objects of the class.

Changing the Effect of isequal

The MATLAB `isequal` function compares two or more arrays for equality in type, size, and contents. This function can also be used to test Java objects for equality.

When you compare two Java objects using `isequal`, MATLAB performs the comparison using the Java method, `equals`. MATLAB first determines the class of the objects specified in the command, and then uses the `equals` method implemented by that class. If it is not implemented in this class, then an inherited `equals` method is used. This will be the `equals` method defined by the `java.lang.Object` class if no intermediate ancestor classes define this method.

You can override inherited `equals` methods in classes that you create by implementing such a method within your class definition. In this way, you can change the way MATLAB performs comparison of the members of this class.

Changing the Effect of double and char

You can also define your own Java methods `toDouble` and `toChar` to change the output of the MATLAB `double` and `char` functions. For more information, see the sections entitled “Converting to the MATLAB double Data Type” and “Converting to the MATLAB char Data Type” on page 7-58.

How MATLAB Handles Undefined Methods

If your MATLAB command invokes a nonexistent method on a Java object, MATLAB looks for a built-in function with the same name. If MATLAB finds a built-in function of that name, it attempts to invoke it. If MATLAB does not find a function with that name, it displays a message stating that it cannot find a method by that name for the class.

For example, MATLAB has a built-in method named `size`, and the Java API `java.awt.Frame` class also has a `size` method. If you call `size` on a `Frame` object, the `size` method defined by `java.awt.Frame` is executed. However, if you call `size` on an object of `java.lang.String`, MATLAB does not find a `size` method for this class. It executes the MATLAB `size` built-in instead.

```
string = java.lang.String('hello');

size(string)
ans =
     1     1
```

Note When you define a Java class for use in MATLAB, avoid giving any of its methods the same name as a MATLAB built-in function.

How MATLAB Handles Java Exceptions

If invoking a Java method or constructor throws an exception, MATLAB catches the exception and transforms it into a MATLAB error. MATLAB puts the text of the Java error message into its own error message. Receiving an error from a Java method or constructor has the same appearance as receiving an error from an M-file.

Working with Java Arrays

You can pass singular Java objects to and from methods or you may pass them in an array, providing the method expects them in that form. This array must either be a Java array (returned from another method call or created within MATLAB) or, under certain circumstances, a MATLAB cell array. This section describes how to create and manipulate Java arrays in MATLAB. Later sections will describe how to use MATLAB cell arrays in calls to Java methods.

Note The term *dimension* here refers more to the number of subscripts required to address the elements of an array than to its length, width, and height characteristics. For example, a 5-by-1 array is referred to as being one-dimensional, as its individual elements can be indexed into using only one array subscript.

This section addresses the following topics:

- "How MATLAB Represents the Java Array"
- "Creating an Array of Objects Within MATLAB"
- "Accessing Elements of a Java Array"
- "Assigning to a Java Array"
- "Concatenating Java Arrays"
- "Creating a New Array Reference"
- "Creating a Copy of a Java Array"

Ex.1049 / Page 504 of 748
Intel Corporation et al

How MATLAB Represents the Java Array

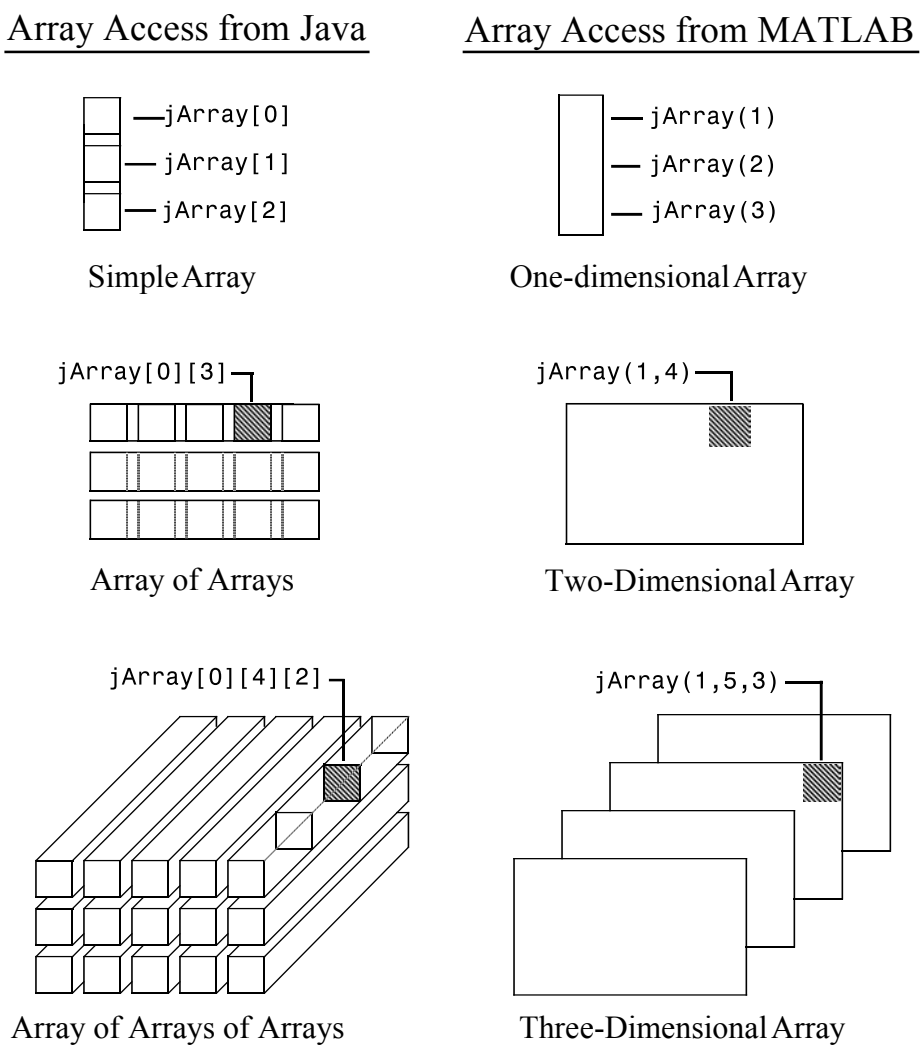
The term *java array* refers to any array of Java objects returned from a call to a Java class constructor or method. You may also construct a Java array within MATLAB using the `javaArray` function. The structure of a Java array is significantly different from that of a MATLAB matrix or array. MATLAB *hides* these differences whenever possible, allowing you to operate on the arrays using the usual MATLAB command syntax. Just the same, it may be helpful to keep the following differences in mind as you work with Java arrays.

Representing More Than One Dimension

An array in the Java language is strictly a one-dimensional structure because it is measured only in length. If you want to work with a two-dimensional array, you can create an equivalent structure using an array of arrays. To add further dimensions, you add more levels to the array, making it an array of arrays of arrays, and so on. You may want to use such multilevel arrays when working in MATLAB as it is a matrix and array-based programming language.

MATLAB makes it easy for you to work with multilevel Java arrays by treating them like the matrices and multidimensional arrays that are a part of the language itself. You access elements of an array of arrays using the same MATLAB syntax that you would use if you were handling a matrix. If you were to add more levels to the array, MATLAB would be able to access and operate on the structure as if it were a multidimensional MATLAB array.

The left side of the following figure shows Java arrays of one, two, and three dimensions. To the right of each is the way the same array is represented to you in MATLAB. Note that single-dimension arrays are represented as a column vector.



Ex.1049 / Page 506 of 748
Intel Corporation et al

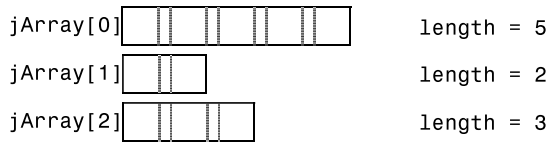
Array Indexing

Java array indexing is different than MATLAB array indexing. Java array indices are zero-based, MATLAB array indices are one-based. In Java programming, you access the elements of array `y` of length `N` using `y[0]` through `y[N-1]`. When working with this array in MATLAB, you access these same elements using the MATLAB indexing style of `y(1)` through `y(N)`. Thus, if you have a Java array of 10 elements, the seventh element is obtained using `y(7)`, and not `y[6]` as you would have used when writing a program in Java.

The Shape of the Java Array

A Java array can be different from a MATLAB array in its overall *shape*. A two-dimensional MATLAB array maintains a rectangular shape, as each row is of equal length and each column of equal height. The Java counterpart of this, an array of arrays, does not necessarily hold to this rectangular form. Each individual lower level array may have a different length.

Such an array structure is pictured below. This is an array of three underlying arrays of different lengths. The term *ragged* is commonly used to describe this arrangement of array elements as the array ends do not match up evenly. When a Java method returns an array with this type of structure, it is stored in a cell array by MATLAB.



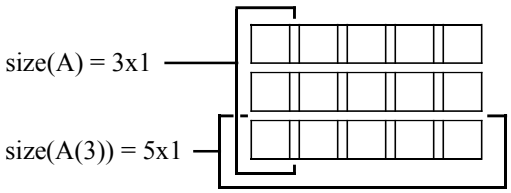
Interpreting the Size of a Java Array

When the MATLAB `size` function is applied to a simple Java array, the number of rows returned is the length of the Java array and the number of columns is always 1.

Determining the size of a Java array of arrays is not so simple. The potentially ragged shape of an array returned from Java makes it impossible to size the array in the same way as for a rectangular matrix. In a ragged Java array, there is no one value that represents the size of the lower level arrays.

Ex.1049 / Page 507 of 748
Intel Corporation et al

When the `size` function is applied to a Java array of arrays, the resulting value describes the top level of the specified array. For the Java array shown here



`size(A)` returns the dimensions of the highest array level of A. The highest level of the array has a size of 3-by-1.

```
size(A)
ans =
     3     1
```

To find the size of a lower level array, say the five-element array in row 3, refer to the row explicitly.

```
size(A(3))
ans =
     5     1
```

You can specify a dimension in the `size` command using the following syntax. However, you will probably find this useful only for sizing the first dimension, `dim=1`, as this will be the only non-unary dimension.

```
m = size(X,dim)

size(A, 1)
ans =
     3
```

Interpreting the Number of Dimensions of a Java Arrays

For Java arrays, whether they are simple one-level arrays or multilevel, the MATLAB `ndims` function always returns a value of 2 to indicate the number of dimensions in the array. This is a measure of the number of dimensions in the top-level array which will always be equal to 2.

Creating an Array of Objects Within MATLAB

To call a Java method that has one or more arguments defined as an array of Java objects, you must, under most circumstances, pass your objects in a Java array. You can construct an array of objects in a call to a Java method or constructor. Or you can create the array within MATLAB.

The MATLAB `javaArray` function lets you create a Java array structure that can be handled in MATLAB as a single multidimensional array. You specify the number and size of the array dimensions along with the class of objects you intend to store in it. Using the one-dimensional Java array as its primary building block, MATLAB then builds an array structure that satisfies the dimensions requested in the `javaArray` command.

Using the javaArray Function

To create a Java object array, use the MATLAB `javaArray` function, which has the following syntax.

```
A = javaArray('element_class', m, n, p, ...)
```

The first argument is the `'element_class'` string, which names the class of the elements in the array. You must specify the fully qualified name (package and class name). The remaining arguments (`m, n, p, ...`) are the number of elements in each dimension of the array.

An array that you create with `javaArray` is equivalent to the array that you would create with the Java code.

```
A = new element_class[m][n][p]...;
```

The following command builds a Java array of four lower level arrays, each capable of holding five objects of the `java.lang.Double` class. (You will probably be more likely to use primitive types of `double` than instances of the `java.lang.Double` class, but in this context, it affords us a simple example.)

```
dblArray = javaArray('java.lang.Double', 4, 5);
```

The `javaArray` function does not deposit any values into the array elements that it creates. You must do this separately. The following MATLAB code stores objects of the `java.lang.Double` type in the Java array `dblArray` that was just created.

```
for m = 1:4
    for n = 1:5
    dblArray(m,n) = java.lang.Double((m*10) + n);
    end
end

dblArray
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Another Way to Create a Java Array

You can also create an array of Java objects using syntax that is more typical to MATLAB. For example, the following syntax creates a 4-by-5 MATLAB array of type double and assigns zero to each element of the array.

```
matlabArray(4,5) = 0;
```

You use similar syntax to create a Java array in MATLAB, except that you must specify the Java class name. The value being assigned, `0` in this example, is stored in the final element of the array, javaArray`(4,5)`. All other elements of the array receive the empty matrix.

```
javaArray(4,5) = java.lang.Double(0)
javaArray =
java.lang.Double[][]:
     []     []     []     []     []
     []     []     []     []     []
     []     []     []     []     []
     []     []     []     []    [0]
```

Note You cannot change the dimensions of an existing Java array as you can with a MATLAB array. The same restriction exists when working with Java arrays in the Java language. See the example below.

This example first creates a scalar MATLAB array, and then successfully modifies it to be two-dimensional.

```
matlabArray = 0;
matlabArray(4,5) = 0

matlabArray =

     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

When you try this with a Java array, you get an error. Similarly, you cannot create an array of Java arrays from a Java array, and so forth.

```
javaArray = java.lang.Double(0);
javaArray(4,5) = java.lang.Double(0);
??? Index exceeds Java array dimensions.
```

Accessing Elements of a Java Array

You can access elements of a Java object array by using the MATLAB array indexing syntax, `A(row,col)`. For example, to access the element of array `dblArray` located at row 3, column 4, use

```
row3_col4 = dblArray(3,4)
row3_col4 =
34.0
```

In Java, this would be `dblArray[2][3]`.

You can also use MATLAB array indexing syntax to access an element in an object's data field. Suppose that `myMenuObj` is an instance of a window menu class. This user-supplied class has a data field, `menuItemArray`, which is a Java array of `java.awt.menuItem`. To get element `3` of this array, use the following command.

```
currentItem = myMenuObj.menuItemArray(3)
```

Using Single Subscript Indexing to Access Arrays

Elements of a MATLAB matrix are most commonly referenced using both row and column subscripts. For example, you use `x(3,4)` to reference the array

element at the intersection of row 3 and column 4. Sometimes it is more advantageous to use just a single subscript. MATLAB provides this capability (see the section on "Advanced Indexing" in the *Using MATLAB* manual).

Indexing into a MATLAB matrix using a single subscript references one element of the matrix. Using the MATLAB matrix shown here, `matlabArray(3)` returns a single element of the matrix.

```
matlabArray = [11 12 13 14 15; 21 22 23 24 25; ...
               31 32 33 34 35; 41 42 43 44 45]
matlabArray =
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
    41    42    43    44    45

matlabArray(3)
ans =
    31
```

Indexing this way into a Java array of arrays references an entire subarray of the overall structure. Using the `dblArray` Java array, that looks the same as `matlabArray` shown above, `dblArray(3)` returns the 5-by-1 array that makes up the entire third row.

```
row3 = dblArray(3)
row3 =
java.lang.Double[]:
    [31]
    [32]
    [33]
    [34]
    [35]
```

This is a useful feature of MATLAB as it allows you to specify an entire array from a larger array structure, and then manipulate it as an object.

Using the Colon Operator

Use of the MATLAB `colon` operator (`:`) is supported in subscripting Java array references. This operator works just the same as when referencing the contents of a MATLAB array. Using the Java array of `java.lang.Double` objects shown

here, the statement `dblArray(2,2:4)` refers to a portion of the lower level array, `dblArray(2)`. A new array, `row2Array`, is created from the elements in columns 2 through 4.

```
dblArray
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]

row2Array = dblArray(2,2:4)
row2Array =
java.lang.Double[]:
    [22]
    [23]
    [24]
```

You also can use the colon operator in single-subscript indexing, as covered in “Using Single Subscript Indexing to Access Arrays” on page 7-35. By making your subscript a colon rather than a number, you can convert an array of arrays into one linear array. The following example converts the 4-by-5 array `dblArray` into a 20-by-1 linear array.

```
linearArray = dblArray(:)
linearArray =
java.lang.Double[]:
    [11]
    [12]
    [13]
    [14]
    [15]
    [21]
    [22]
     .
     .
     .
```

This works the same way on an N-dimensional Java array structure. Using the colon operator as a single subscripted index into the array produces a linear array composed of all of the elements of the original array.

Note Java and MATLAB arrays are stored differently in memory. This is reflected in the order they are given in a linear array. Java array elements are stored in an order that matches the *rows* of the matrix, (`11`, `12`, `13`, `...` in the array shown above). MATLAB array elements are stored in an order that matches the *columns*, (`11`, `21`, `31`, `...`).

Using END in a Subscript

You can use the `end` keyword in the first subscript of an access statement. The first subscript references the top-level array in a multilevel Java array structure.

Note Using `end` on lower level arrays is not valid due to the potentially ragged nature of these arrays (see "The Shape of the Java Array" on page 7-31). In this case, there is no consistent end value to be derived.

The following example displays data from the third to the last row of Java array `dblArray`.

```
last2rows = dblArray(3:end, :)
last2rows =
java.lang.Double[][]:
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Assigning to a Java Array

You assign values to objects in a Java array in essentially the same way as you do in a MATLAB array. Although Java and MATLAB arrays are structured quite differently, you use the same command syntax to specify which elements you want to assign to. See "How MATLAB Represents the Java Array" on page 7-29 for more information on Java and MATLAB array differences.

The following example deposits the value 300 in the dblArray element at row 3, column 2. In Java, this would be dblArray[2][1].

```
dblArray(3,2) = java.lang.Double(300)
dblArray =
java.lang.Double[][]:
    [11]    [ 12]    [13]    [14]    [15]
    [21]    [ 22]    [23]    [24]    [25]
    [31]    [300]    [33]    [34]    [35]
    [41]    [ 42]    [43]    [44]    [45]
```

You use the same syntax to assign to an element in an object's data field. Continuing with the myMenuObj example shown in "Accessing Elements of a Java Array" on page 7-35, you assign to the third menu item in menuItemArray as follows.

```
myMenuObj.menuItemArray(3) = java.lang.String('Save As...');
```

Using Single Subscript Indexing for Array Assignment

You can use a single-array subscript to index into a Java array structure that has more than one dimension. Refer to "Using Single Subscript Indexing to Access Arrays" on page 7-35 for a description of this feature as used with Java arrays.

You can use single-subscript indexing to assign values to an array as well. The example below assigns a one-dimensional Java array, onedimArray, to a row of a two-dimensional Java array, dblArray. Start out by creating the one-dimensional array.

```
onedimArray = javaArray('java.lang.Double', 5);
for k = 1:5
    onedimArray(k) = java.lang.Double(100 * k);
    end
```

Since dblArray(3) refers to the 5-by-1 array displayed in the third row of dblArray, you can assign the entire, similarly dimensioned, 5-by-1 onedimArray to it.

Ex.1049 / Page 515 of 748
Intel Corporation et al

```
dblArray(3) = onedimArray
dblArray =
java.lang.Double[][]:
    [ 11]    [ 12]    [ 13]    [ 14]    [ 15]
    [ 21]    [ 22]    [ 23]    [ 24]    [ 25]
    [100]    [200]    [300]    [400]    [500]
    [ 41]    [ 42]    [ 43]    [ 44]    [ 45]
```

Assigning to a Linear Array

You can assign a value to *every* element of a multidimensional Java array by treating the array structure as if it were a single linear array. This entails replacing the single, numerical subscript with the MATLAB colon operator. If you start with the `dblArray` array, you can initialize the contents of every object in the two-dimensional array with the following statement.

```
dblArray(:) = java.lang.Double(0)
dblArray =
java.lang.Double[][]:
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
```

Using the Colon Operator

You can use the MATLAB colon operator as you would when working with MATLAB arrays. The statements below assign given values to each of the four rows in the Java array, `dblArray`. Remember that each row actually represents a separate Java array in itself.

```
dblArray(1,:) = java.lang.Double(125);
dblArray(2,:) = java.lang.Double(250);
dblArray(3,:) = java.lang.Double(375);
dblArray(4,:) = java.lang.Double(500)
dblArray =
java.lang.Double[][]:
    [125]    [125]    [125]    [125]    [125]
    [250]    [250]    [250]    [250]    [250]
    [375]    [375]    [375]    [375]    [375]
    [500]    [500]    [500]    [500]    [500]
```

Assigning the Empty Matrix

When working with MATLAB arrays, you can assign the empty matrix, (i.e., the 0-by-0 array denoted by []) to an element of the array. For Java arrays, you can also assign [] to array elements. This stores the `NULL` value, rather than a 0-by-0 array, in the Java array element.

Subscripted Deletion

When you assign the empty matrix value to an entire row or column of a MATLAB array, you find that MATLAB actually removes the affected row or column from the array. In the example below, the empty matrix is assigned to all elements of the fourth column in the MATLAB matrix, `matlabArray`. Thus, the fourth column is completely eliminated from the matrix. This changes its dimensions from 4-by-5 to 4-by-4.

```
matlabArray = [11 12 13 14 15; 21 22 23 24 25; ...
               31 32 33 34 35; 41 42 43 44 45]
matlabArray =
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
    41    42    43    44    45

matlabArray(:,4) = []
matlabArray =
    11    12    13    15
    21    22    23    25
    31    32    33    35
    41    42    43    45
```

You can assign the empty matrix to a Java array, but the effect will be different. The next example shows that, when the same operation is performed on a Java array, the structure is not collapsed; it maintains its 4-by-5 dimensions.

```
dblArray(:,4) = []
dblArray =
java.lang.Double[][]:
    [125]    [125]    [125]     []    [125]
    [250]    [250]    [250]     []    [250]
    [375]    [375]    [375]     []    [375]
    [500]    [500]    [500]     []    [500]
```

The `dblArray` data structure s actually an array of five-element arrays of `java.lang.Double` objects. The empty array assignment placed the `NULL` value in the fourth element of each of the lower level arrays.

Concatenating Java Arrays

You can concatenate arrays of Java objects in the same way as arrays of other data types. (To understand how scalar Java objects are concatenated by MATLAB see "Concatenating Java Objects" on page 7-11.)

Use either the `cat` function or the square bracket (`[]`) operators. This example horizontally concatenates two Java arrays: `d1`, and `d2`.

```
% Construct a 2-by-3 array of java.lang.Double.
d1 = javaArray('java.lang.Double',2,3);
for m = 1:3     for n = 1:3
d1(m,n) = java.lang.Double(n*2 + m-1);
end;            end;

d1
d1 =
java.lang.Double[][]:
    [2]    [4]    [6]
    [3]    [5]    [7]
    [4]    [6]    [8]

% Construct a 2-by-2 array of java.lang.Double.
d2 = javaArray('java.lang.Double',2,2);
for m = 1:3     for n = 1:2
d2(m,n) = java.lang.Double((n+3)*2 + m-1);
end;            end;
```

```
d2
d2 =
java.lang.Double[][]:
    [ 8]    [10]
    [ 9]    [11]
    [10]    [12]

% Concatenate the two along the second dimension.
d3 = cat(2,d1,d2)
d3 =
java.lang.Double[][]:
    [2]    [4]    [6]    [ 8]    [10]
    [3]    [5]    [7]    [ 9]    [11]
    [4]    [6]    [8]    [10]    [12]
```

Creating a New Array Reference

Because Java arrays in MATLAB are *references*, assigning an array variable to another variable results in a second reference to the array.

Consider the following example where two separate array variables reference a common array. The original array, `origArray`, is created and initialized. The statement `newArrayRef = origArray` creates a copy of this array variable. Changes made to the array referred to by `newArrayRef` also show up in the original array.

```
origArray = javaArray('java.lang.Double', 3, 4);
for m = 1:3
   for n = 1:4
      origArray(m,n) = java.lang.Double((m * 10) + n);
   end
end

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```

```
%  ----- Make a copy of the array reference -----
newArrayRef = origArray;
newArrayRef(3,:) = java.lang.Double(0);

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [ 0]    [ 0]    [ 0]    [ 0]
```

Creating a Copy of a Java Array

You can create an entirely new array from an existing Java array by indexing into the array to describe a block of elements, (or subarray), and assigning this subarray to a variable. The assignment copies the values in the original array to the corresponding cells of the new array.

As with the example in section "Creating a New Array Reference" on page 7-43, an original array is created and initialized. But, this time, a copy is made of the array contents rather than copying the array reference. Changes made using the reference to the new array do not affect the original.

```
origArray = javaArray('java.lang.Double', 3, 4);
for m = 1:3
   for n = 1:4
      origArray(m,n) = java.lang.Double((m * 10) + n);
   end
end

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```

```
%  ----- Make a copy of the array contents -----
newArray = origArray(:,:);
newArray(3,:) = java.lang.Double(0);

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```


Ex.1049 / Page 521 of 748
Intel Corporation et al