Session 3: Object-oriented programming and external interfaces

Version 1.0.2

Aaron Ponti

Contents

1 Object-oriented programmingr

1.1 Introduction to OOP

Creating software applications typically involves designing how to represent the application data and determining how to implement operations performed on that data. **Procedural programs** pass data to functions, which perform the necessary operations on the data. **Object-oriented software** encapsulates data and operations in objects that interact with each other via the object's interface.

The MATLAB language enables you to create programs using both procedural and object-oriented techniques and to use objects and ordinary functions in your programs.

Procedural Program Design In procedural programming, your design focuses on steps that must be executed to achieve a desired state. You typically represent data as individual variables or fields of a structure and implement operations as functions that take the variables as arguments. Programs usually call a sequence of functions, each one of which is passed data, and then returns modified data. Each function performs an operation or perhaps many operations on the data.

Object-Oriented Program Design The object-oriented program design involves:

- Identifying the components of the system or application that you want to build
- Analyzing and identifying patterns to determine what components are used repeatedly or share characteristics
- Classifying components based on similarities and differences

After performing this analysis, you define classes that describe the objects your application uses.

Classes and Objects A class describes a set of objects with common characteristics. Objects are specific instances of a class. The values contained in an object's properties are what make an object different from other objects of the same class (an object of class *double* might have a value of 5). The functions defined by the class (called *methods*) are what implement object behaviors that are common to all objects of a class (you can add two doubles regardless of their values).

1.2 Classes in MATLAB

In the MATLAB language, every value is assigned to a **class**. For example, creating a variable with an assignment statement constructs a variable of the appropriate class:

```
>> a = 7;
>> b = 'some string';
>> whos
  Name    Size        Bytes    Class

  a       1x1         8        double
  b       1x11        22       char
```

Basic commands like *whos* display the class of each value in the workspace. This information helps MATLAB users recognize that some values are characters and display as text while other values might be double, single, or other types of numbers. Some variables can contain different classes of values like *cells*.

1.3 User-defined classes

You can create your own MATLAB classes. For example, you could define a class to represent polynomials. This class could define the operations typically associated with MATLAB classes, like addition, subtraction, indexing, displaying in the command window, and so on. However, these operations would need to perform the equivalent of polynomial addition, polynomial subtraction, and so on. For example, when you add two polynomial objects:

```
p1 + p2
```

the *plus* operation would know how to add polynomial objects because the polynomial class defines this operation. When you define a class, you **overload** special MATLAB functions (*plus.m* for the addition operator) that are called by the MATLAB runtime when those operations are applied to an object of your class.

1.4 MATLAB classes - key terms

MATLAB classes use the following words to describe different parts of a class definition and related concepts.

- Class definition — Description of what is common to every instance of a class.
- Properties — Data storage for class instances
- Methods — Special functions that implement operations that are usually performed only on instances of the class
- Events — Messages that are defined by classes and broadcast by class instances when some specific action occurs
- Attributes — Values that modify the behavior of properties, methods, events, and classes
- Listeners — Objects that respond to a specific event by executing a callback function when the event notice is broadcast
- Objects — Instances of classes, which contain actual data values stored in the objects' properties
- Subclasses — Classes that are derived from other classes and that inherit the methods, properties, and events from those classes (subclasses facilitate the reuse of code defined in the superclass from which they are derived).
- Superclasses — Classes that are used as a basis for the creation of more specifically defined classes (i.e., subclasses).
- Packages — Folders that define a scope for class and function naming

1.5 Handle vs. value classes

There are two kinds of MATLAB classes—handle classes and value classes.

- *Handle classes* create objects that reference the data contained. When you copy a handle object, MATLAB copies the handle, but not the data stored in the object properties. The copy refers to the same data as the original handle. If you change a property value on the original object, the copied object reflects the same change.
- *Value classes* make copies of the data whenever the object is copied or passed to a function. MATLAB numeric types are value classes.

The kind of class you use depends on the desired behavior of the class instances and what features you want to use. Use a handle class when you want to create a reference to the data contained in an object of the class, and do not want copies of the object to make copies of the object data. For example, use a handle class to implement an object that contains information for a phone book entry. Multiple application programs can access a particular phone book entry, but there can be only one set of underlying data. The reference behavior of handles enables these classes to support features like events, listeners, and dynamic properties. Use value classes to represent entities that do not need to be unique, like numeric values. For example, use a value class to implement a polynomial data type. You can copy a polynomial object and then modify its coefficients to make a different polynomial without affecting the original polynomial. In section 1.8 we will see an example comparison between a value and a handle class.

1.6 Class folders

There are two basic ways to specify classes with respect to folders:

- Creating a **single, self-contained class definition file** in a folder on the MATLAB path. The name of the file must match the class (and constructor) name and must have the .m extension. The class is defined entirely in this file.
- Distributing a class definition to multiple files in an **@ folder** inside a path folder. Only one class can be defined in a @ folder.

In addition, **package folders** (which always begin with a "+" character) can contain multiple class definitions, package-scoped functions, and other packages. A package folder defines a new name space in which you can reuse class names. Use the package name to refer to classes and functions defined in package folders (for example, *packagefld1.ClassNameA()*, *packagefld2.packageFunction()*).

1.7 Class building blocks

The basic components in the class definition are blocks describing the whole class and specific aspects of its definition.

1.7.1 The classdef block

The classdef block contains the class definition within a file that starts with the classdef keyword and terminates with the end keyword.

```
classdef className
  ...
end
```

The classdef block contains the class definition. The classdef line is where you specify:

- Class attributes

  Class attributes modify class behavior in some way. Assign values to class attributes only when you want to change their default value. No change to default attribute values:

  ```
  classdef className
    ...
  end
  ```

  One or more attribute values assigned:

  ```
  classdef (attribute1 = value,...) className
    ...
  end
  ```

- Superclasses

  To define a class in terms of one (*simple inheritance*) or more other classes (*multiple inheritance*) by specifying the superclasses on the classdef line:

  ```
  classdef className < superClass1Name & superClass2Name
    ...
  end
  ```

  A handle class inherits from the class *handle* (i.e. it has the class *handle* as superclass):

  ```
  classdef handleClassName < handle
    ...
  end
  ```

  In section 1.9 we will see and example of class inheritance.

The classdef block contains the *properties*, *methods*, and *events* subblocks.

Class attributes Possible attributes for classes are:

- *Hidden* (logical, default is false): if true, the class does not appear in the output of MATLAB commands or tools that display class names;
- *InferiorClasses* (cell, default is {}): use this attribute to establish a precedence relationship among classes;
- *ConstructOnLoad* (logical, default is false): if true, MATLAB calls the class constructor when loading an object from a MAT-file. Therefore, you must implement the constructor so it can be called with no arguments without producing an error;
- *Sealed* (logical, default is false): if true, this class can not be subclassed.

1.7.2 The properties block

The properties block (one for each unique set of attribute specifications) contains property definitions, including optional initial values.

Properties encapsulate the data that belongs to instances of classes. Data contained in properties can be *public*, *protected*, or *private*. This data can be a fixed set of constant values, or it can be *dependent* on other values and calculated only when queried. You control these aspects of property behaviors by setting property attributes and by defining property-specific access methods.

The properties block starts with the properties keyword and terminates with the end keyword.

```
classdef className
  properties
    ...
  end
  ...
end
```

You can control aspects of property definitions in the following ways:

- Specifying a default value for each property individually
- Assigning attribute values on a per block basis
- Defining methods that execute when the property is set or queried

There are two basic approaches to initializing property values:

- In the property definition — MATLAB evaluates the expression only once and assigns the same value to the property of every instance.

```
classdef className
  properties
    PropertyName % No default value assigned
    PropertyName = 'some text';
    PropertyName = sin(pi/12); % Expression returns default value
  end
end
```

Evaluation of property default values occurs only when the value is first needed, and only once when MATLAB first initializes the class. MATLAB does not reevaluate the expression each time you create a class instance.

- In the class constructor (see section 1.7.3) — MATLAB evaluates the assignment expression for each instance, which ensures that each instance has a unique value.

  To assign values to a property from within the class constructor, reference the object that the constructor returns (the output variable *obj*):

  ```
  classdef MyClass
    properties
      PropertyOne
    end
    methods
      function obj = MyClass(intval)
        obj.PropertyOne = intval;
      end
    end
  end
  ```

All properties have **attributes** that modify certain aspects of the property's behavior. Specified attributes apply to all properties in a particular properties block. For example:

```
classdef className
  properties
    PropertyName % No default value assigned
    PropertyName = sin(pi/12); % Expression returns default value
  end
  properties (SetAccess = private, GetAccess = private)
    Stress
    Strain
  end
end
```

In this case, only methods in the same class definition can modify and query the *Stress* and *Strain* properties. This restriction exists because the class defines these properties in a properties block with *SetAccess* and *GetAccess* attributes set to private.

You can define methods that MATLAB calls whenever setting or querying a property value (see next section). Define property **set** access or **get** access methods in *methods* blocks that specify no attributes and have the following syntax:

```
methods
  function value = get.PropertyName(object)
    ...
  end
  function obj = set.PropertyName(obj,value)
    ...
  end
end
```

MATLAB does not call the property set access method when assigning the default value specified in the property's definition block. Moreover, if a handle class defines the property, the set access method does not need to return the modified object (see also section 1.8).

Property attributes Possible attributes for properties are:

- *AbortSet* (logical, default is false): if true, and this property belongs to a handle class, then MATLAB does not set the property value if the new value is the same as the current value. This approach prevents the triggering of property *PreSet* and *PostSet* events.
- *Abstract* (logical, default is false): if true, the property has no implementation, but a concrete subclass must redefine this property without Abstract being set to true.
- *Access* (enumeration, default is 'public'):
  - *public*: unrestricted access;
  - *protected:* access from class or derived classes;
  - *private:* access by class members only.
  - use *Access* to set both *SetAccess* and *GetAccess* to the same value. Query the values of *SetAccess* and *GetAccess* directly (not *Access*).
- *Constant* (logical, default is false): set to true if you want only one value for this property in all instances of the class:
  - subclasses inherit constant properties, but cannot change them;
  - constant properties cannot be *Dependent*;
  - *setAccess* is ignored.
- *Dependent* (logical, default is false): if false, property value is stored in object. If true, property value is not stored in object. The set and get functions cannot access the property by indexing into the object using the property name. MATLAB does not display in the command window the names and values of *Dependent* properties that do not define a get method (scalar object display only).
- *GetAccess*: see Access
- *GetObservable* (logical, default is false): if true, and it is a handle class property, then you can create listeners for access to this property. The listeners are called whenever property values are queried.
- *Hidden* (logical, default is false): determines whether the property should be shown in a property list (e.g., Property Inspector, call to set or get, etc.). MATLAB does not display in the command window the names and values of properties whose *Hidden* attribute is true or properties having protected or private *GetAccess*.

- *SetAccess*: see Access
- *SetObservable* (logical, default is false): if true, and it is a handle class property, then you can create listeners for access to this property. The listeners are called whenever property values are modified.
- *Transient* (logical, default is false): if true, property value is not saved when object is saved to a file.

1.7.3 The methods block

The methods block (one for each unique set of attribute specifications) contains function definitions for the class methods. The methods block starts with the methods keyword and terminates with the end keyword.

```
classdef className
  ...
  methods
    ...
  end
  ...
end
```

Methods are functions that implement the operations performed on objects of a class. Methods, along with other class members support the concept of **encapsulation**—class instances contain data in properties and class methods operate on that data. This allows the internal workings of classes to be hidden from code outside of the class, and thereby enabling the class implementation to change without affecting code that is external to the class. Methods have access to private members of their class including other methods and properties. This enables you to hide data and create special interfaces that must be used to access the data stored in objects.

There are specialized kinds of methods that perform certain functions or behave in particular ways:

- *Ordinary* methods are functions that act on one or more objects and return some new object or some computed value. These methods are like ordinary MATLAB functions that cannot modify input arguments. Ordinary methods enable classes to implement arithmetic operators and computational functions. These methods require an object of the class on which to operate.
- *Constructor* methods are specialized methods that create objects of the class. A constructor method must have the same name as the class and typically initializes property values with data obtained from input arguments. The class constructor method must return the object it creates.
- *Destructor* methods are called automatically when the object is destroyed, for example if you call delete(object) or there are no longer any references to the object.

- *Property access* methods enable a class to define code to execute whenever a property value is queried or set.
- *Static* methods are functions that are associated with a class, but do not necessarily operate on class objects. These methods do not require an instance of the class to be referenced during invocation of the method, but typically perform operations in a way specific to the class.
- *Conversion* methods are overloaded constructor methods from other classes that enable your class to convert its own objects to the class of the overloaded constructor. For example, if your class implements a *double* method, then this method is called instead of the *double* class constructor to convert your class object to a MATLAB *double* object.
- *Abstract* methods serve to define a class that cannot be instantiated itself, but serves as a way to define a common interface used by a number of subclasses. Classes that contain abstract methods are often referred to as interfaces.

The **constructor** method has the same name as the class and returns an object. You can assign values to properties in the class constructor. Terminate all method functions with an end statement.

```
classdef ClassName
  ...
  methods
    function obj = ClassName(arg1,arg2,...) % Constructor
      obj.Prop1 = arg1;
      ...
    end
    function normalMethod(obj,arg1,...)
      ...
    end
  end
  methods (Static = true)                   % A static
    function staticMethod(arg1,...)         % method is a
      ...                                   % class method
    end
  end
end
```

MATLAB differs from languages like C++ and Java in that there is no special hidden class instance (e.g. the *this* object) passed to all methods. You must pass an object of the class explicitly to the method. The left most argument does not need to be the class instance, and the argument list can have multiple objects.

You can define class methods in files that are separate from the class definition file. To use multiple files for a class definition, put the class files in a folder having a name beginning with the @ character followed by the name of the class. Ensure that the parent folder of the @-folder is on the MATLAB path. To define a method in a separate file in the class @-folder, create the function in a separate file, but do not use a method block in that file. Name the file with the function name, as with any function.

You must put the following methods in the classdef file, not in separate files:

- Class constructor
- Delete method
- All functions that use dots in their names, including:
  - Converter methods that convert to classes contained in packages, which must use the package name as part of the class name
  - Property *set* and *get* access methods

If you specify method attributes for a method that you define in a separate file, include the method signature in a methods block in the classdef block. For example, the following code shows a method with *Access* set to private in the methods block. The method implementation resides in a separate file. Do not include the function or end keywords in the methods block, just the function signature showing input and output arguments.

```
classdef ClassName
  ...
  % In a methods block, set the method attributes
  % and add the function signature
  methods (Access = private)
    output = myFunc(obj,arg1,arg2)
  end
end
```

In a file named *myFunc.m*, in the @ClassName folder, define the function:

```
function output = myFunc(obj,arg1,arg2)
  ...
end
```

Method attributes Possible attributes for methods are:

- *Abstract* (logical, default is false): if true, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method:
  - subclasses are not required to define the same number of input and output arguments. However, subclasses generally use the same signature when implementing their version of the method;
  - the method can have comments after the function line;
  - the method does not contain function or end keywords, only the function syntax (e.g., *[a,b] = myMethod(x,y)*)
- *Access* (enumeration, default is 'public'): determines what code can call this method

  - *public*: unrestricted access;
  - *protected:* access from methods in class or derived classes;
  - *private:* access by class methods members only (not from subclasses).

- *Hidden* (logical, default is false): when false, the method name shows in the list of methods displayed using the *methods* or *methodsview* commands. If set to true, the method name is not included in these listings.
- *Sealed* (logical, default is false): tf true, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.
- Static (logical, default is false): set to true to define a method that does not depend on an object of the class and does not require an object argument. You must use the class name to call the method: *classname.methodname*

1.7.4 The events block

The events block (one for each unique set of attribute specifications) contains the names of events that this class declares. **Please mind that only handle classes can define and use events.**

The events blocks starts with the events keyword and terminates with the end keyword.

```
classdef className
  ...
  events
    ...
  end
  ...
end
```

To define an event, you declare a name for the event in the events block. Then one of the class methods triggers the event using the notify method, which is method inherited from the *handle* class. Only classes derived from the *handle* class can define events. For example, the following class:

- Defines an event named *StateChange*
- Triggers the event using the *notify* method (inherited from handle).

```
classdef className < handle % Subclass handle
  ...
  events % Define an event called StateChange
    StateChange
  end
  ...
  methods
    function upDateGUI(obj)
      ...
      % Broadcast notice that StateChange event has occurred
      notify(obj,'StateChange');
    end
  end
end
```

Any number of objects can be listening for the *StateChange* event to occur. When notify executes, MATLAB calls all registered listener callbacks and passes the handle of the object generating the event and an event structure to these functions. To register a listener callback, use the *addlistener()* method of the handle class.

```
addlistener(eventObj,'StateChange',@myCallback)
```

In section 1.10 we will see an example of handle class using events to notify changes to its properties.

Event attributes Possible attributes for events are:

- *Hidden* (logical, default is false): if true, event does not appear in list of events returned by events function (or other event listing functions or viewers).
- *ListenAccess* (enumeration, default is 'public'): determines where you can create listeners for the event:
  - public: unrestricted access;
  - protected: access from methods in class or derived classes;
  - private: access by class methods only (not from derived classes)
- *NotifyAccess* (enumeration, default is 'public'): determines where code can trigger the event:
  - public: any code can trigger event;
  - protected: can trigger event from methods in class or derived classes;
  - private: can trigger event by class methods only (not from derived classes)

1.8 Example: value vs. handle class

We will now see a few example classes to get familiar with the concepts discussed so far. We will write our first class both as a **value** and as a **handle** class to pinpoint the differences. The class Counter implements a simple counter that can be incremented and queried for its current value:

```
classdef Counter
  properties ( SetAccess = protected, GetAccess = public )
    value
  end
  methods
    function obj = Counter( val )         % Constructor
        obj.value = val;
    end
    function obj = increment( obj )       % The method increment
        obj.value = obj.value + 1;        % returns the object
    end
    function value = get.value( obj )
        value = obj.value;
    end
    function display( obj )
        disp( [ 'Value = ', num2str( obj.value ) ] );
    end
  end
end
```

The class CounterHandle has the exact same functionality than Counter, but is implemented as a handle class.

```
classdef CounterHandle < handle
  properties ( SetAccess = protected, GetAccess = public )
    value
  end
  methods
    function obj = CounterHandle( val )   % Constructor
      obj.value = val;
    end
    function increment( obj )             % The method increment
      obj.value = obj.value + 1;          % does not return
    end                                   % the object
    function value = get.value( obj )
      value = obj.value;
    end
    function display( obj )
      disp( [ 'Value = ', num2str( obj.value ) ] );
    end
  end
end
```

One immediate difference between the classes is that the methods of the handle class change the actual object (in place), while the methods of the value classes always changes a copy of it: to update the object, one has to assign the modified copy back to the original object.

```
c = Counter( 0 );
c = c.increment()        % or: c = increment( c );
Value = 1

d = CounterHandle( 0 );
d.increment()            % or: increment( d );
Value = 1

for i = 1 : 5
  c.increment();         % We do not assign the
end                      % result back to c
c
```

```
Value = 1                % Value is still 1!

for i = 1 : 5
  d.increment();
end
d
Value = 6                % Value is now 6
```

1.9 Example: class inheritance

Imagine that we are now told we need a universal counter that can both increment and decrement its internal value. We could add a *decrement*() method to either Counter or CounterHandle as defined in the previous section, but this would mean that we lost the original Counter (that can only increment). We could also create a new class called UniversalCounter that contains all methods of, say, CounterHandle with the additional *decrement*() method, but this would be a useless repetition of code: UniversalCounter would be an almost identical copy of CounterHandle with one additional method[1].

A more elegant solution is to have our new class UniversalCounter **inherit** from CounterHandle.

```
classdef UniversalCounter < CounterHandle
  methods
    function obj = UniversalCounter( val )  % UniversalCounter constructor
      obj = obj@CounterHandle( val );       % calls the CounterHandle constructor
    end
    function decrement( obj )               % Our new decrement() method
      obj.value = obj.value - 1;
    end
  end
end
```

Our new UniversalCounter has all methods and properties of the base class CounterHandle plus the new *decrement*() method. Inheriting from a class allows us to *just implement the differences* between the base class and the desired final class.

Notice that the UniversalCounter constructor must call the CounterHandle constructor and pass the expected input parameter *val*: failing to do so results in an error when the base object is constructed. Calling the base constructor explicitly is not needed if this does not take any input parameters (MATLAB will call it transparently with no parameters).

Notice also that since the new *decrement*() method modifies the *value* property declared in the base class, the inheriting class must explicitly be given

[1]Even for very small classes this duplication is not desirable: imagine you create a large number of derivative classes in a way similar to our UnversalCounter example: not only would the total amount of code increase significantly: if you wanted to make a minor structural change to classes (or fix a bug), you would have to change them all, whereas with inheritance modifying just the base class would propagate the changes automatically to all the derived classes. As a rule of thumb, try to avoid code duplication in any case!

permission to do so. By setting the *SetAccess* attribute to *protected* in CounterHandle, we allow any class that inherits from CounterHandle to modify the property. If we set *SetAccess* to *private* in CounterHandle, the UniversalCounter would have only been allowed to increment the value, but not decrement it! This is easily explainable by the fact that the *increment*() method is defined in CounterHandle (which can obviously access its own properties), whereas the *decrement*() method is defined in UniversalCounter, which has to be given permission to modify value.

```
e = UniversalCounter( 10 );
e.increment( )     % The increment() method is implemented in CounterHandle (and inherited)
e.decrement( )     % The decrement() method is implemented in UniversalCounter
e.decrement( )
e
Value = 9
```

Finally, notice an important characteristic of inheritance:

```
class( e )
ans =
UniversalCounter
>> isa( e, 'UniversalCounter' )
ans =
     1
>> isa( e, 'CounterHandle' )
ans =
     1
>> isa( e, 'Counter' )
ans =
     0
```

Class inheritance builds an *is-a* relationship between classes. If you define a class Employee that inherits from a class Person, you can say than an Employee is a Person. This is true also for our UniversalCounter, which is both an UniversalCounter and a CounterHandle (but is not a Counter).

1.10 Example: events

In this example, we will write a class to convert temperature from Celsius to Fahrenheit and vice versa. There are plenty of ways to implement this kind of functionality: here we will use *events*.

```
classdef Converter < handle

  % Direct access to the properties allowed
  properties ( Access = public )
    % Temperatures in Celsius and Fahrenheit
    tC, tF
  end

  % Define events that can be triggered: access is public
  events ( ListenAccess = public, NotifyAccess = public )
    changedTC, changedTF
  end

  methods
    % Constructor
    function obj = Converter( )
      % Set listeners
      addlistener( obj, 'changedTC', @convertCtoF );
      addlistener( obj, 'changedTF', @convertFtoC );

      % Set reasonable starting values
      % This will trigger the changedTC event and update tF
      obj.tC = 0;
    end
    % Setters
    % The events will be triggered only if the properties change,
    % this prevents a cascade of events being generated
    function set.tC( obj, value )
      obj.tC = value;
      obj.notify( 'changedTC' );
    end

    function set.tF( obj, value )
      obj.tF = value;
      obj.notify( 'changedTF' );
    end

    function display( obj )
      fprintf( 1, ...
        [ 'Current temperature: %.2f Celsius corresponding ', ...
        'to %.2f Fahrenheit.\n' ], obj.tC, obj.tF );
    end
  end

  methods ( Access = private )
    function convertCtoF( obj, evnt )
      obj.tF = ( obj.tC * 9 / 5 ) + 32;
    end
    function convertFtoC( obj, evnt )
      obj.tC = ( obj.tF - 32 ) * 5 / 9;
    end
  end
end
```

This class can be used as follows:

```
c = Converter( )
Current temperature: 0.00 Celsius corresponding to 32.00 Fahrenheit.
```

Instantiating a Converter objects sets a reasonable initial temperature of 0 degrees Celsius. Temperatures can then be changed by directly setting the properties *tC* and *tF*.

```
c.tC = 25
Current temperature: 25.00 Celsius corresponding to 77.00 Fahrenheit.

c.tF = 100
Current temperature: 37.78 Celsius corresponding to 100.00 Fahrenheit.
```

Obviously, when one of the two temperatures are changed by the user, the other one should be updated accordingly. In our example, this is done by **notifying** the change in the two *set.tC* and *set.tF* methods: the *notify()* function (inherited from the handle class) creates an **event**.

```
obj.notify( 'changedTC' );
```

In the constructor we added **listeners** for the *changedTC* and *changedTF* events, and we assigned the *convertCtoF*() and *convertFtoC*() callbacks to be invoked whenever the listener would capture the *changedTC* and *changedTF* events.

When the user sets a new temperature *tC*, this is what happens:

```
c.tC = 25
    →set.tC is called
    →the event changedTC is generated
    →the listener captures the changedTC event and calls the convertCtoF() callback
    →convertCtoF() sets the new value for tF
    →set.tF is called
    →the event changedTF is generated
    →the listener captures the event changedTF and calls the convertFtoC() callback
    →convertFtoC() sets the new value for tC which is identical to the previous
    →no new events are generated, and the cascade of events is stopped
```

One could argue that using events to update one temperature when the other is changed is an overkill: indeed, a simple function call to *convertCtoF*() from *set.tC()* and, correspondingly, a call to *convertFtoC()* from *set.tF()* would obtain the same effect and avoid recalculating *tC* from *tF* after *tF* has just been updated from *tC*.

So why bothering creating events, then? The decision to notify an event when something changes within a class uncouples the class to the use its clients will make of it. An user interface that uses the Converter class could add its own listeners to the events generated by the class and react to the *changedTC* and *changedTF* events (since access to them is public) transparently from the Converter class. Every object that adds listeners for these events will be notified when the temperature changes. This makes the class reusable by new clients without any additional modification to the class itself[2].

1.11 More extensive example: a polynomial class

This example implements a class to represent polynomials in the MATLAB language. A value class is used because the behavior of a polynomial object within the MATLAB environment should follow the copy semantics of other MATLAB variables. This example is taken from the official Object-Oriented Programming documentation but omits some of the more advanced (and not so relevant) details. For the complete example, see the references.

This class overloads a number of MATLAB functions, such as *roots*, *polyval*, *diff*, and *plot* so that these function can be used with the new polynomial object.

[2]The class does not even need to know that those clients *exist*!

1.11.1 Creating the needed folder and file

To use the class, create a folder named *@DocPolynom* and save *DocPolynom.m* to this folder. The parent folder of *@DocPolynom* must be on the MATLAB path.

1.11.2 Using the DocPolynom Class

The following examples illustrate basic use of the DocPolynom class: we will see how to create a *DocPolynom* object, how to find its roots and how to add two *DocPolynom* objects together.

We will start by creating two *DocPolynom* objects: note that the argument to the constructor function contains the polynomial coefficients for our polynomials $f(x) = x^3 - 2x - 5$ and $g(x) = 2x^4 + 3x^2 + 2x - 7$.

```
p1 = DocPolynom([1 0 -2 -5])
p1 =
    x^3 - 2*x - 5
p2 = DocPolynom([2 0 3 2 -7])
p2 =
    2*x^4 + 3*x^2 + 2*x - 7
```

The *DocPolynom disp* method displays the polynomial in MATLAB syntax.

We can find the roots of the polynomial using the overloaded *roots* method.

```
roots(p1)
ans =
     2.0946
    -1.0473 + 1.1359i
    -1.0473 - 1.1359i
```

We can also add the two polynomials p1 and p2.

The MATLAB runtime calls the *plus* method defined for the *DocPolynom* class when you add two *DocPolynom* objects.

```
p1 + p2
ans =
    2*x^4 + x^3 + 3*x^2 - 12
```

The sections that follow describe the implementation of the methods illustrated here, as well as some implementation details.

1.11.3 The DocPolynom Constructor Method

The following function is the *DocPolynom* class constructor, which is in the file *@DocPolynom/DocPolynom.m*:

```
function obj = DocPolynom(c)
  % Construct a DocPolynom object using the coefficients supplied
  if isa(c,'DocPolynom')
    obj.coef = c.coef;
  else
    obj.coef = c(:).';
  end
end
```

The coefficients are stored in the *coef* property of the *DocPolynom* class:

```
classdef DocPolynom     % DocPolynom is a value class

  properties
    coef
  end

  methods
    function obj = DocPolynom(c)
    % Construct a DocPolynom object using the coefficients supplied
...
```

Constructor Calling Syntax You can call the *DocPolynom* constructor method with two different arguments:

- Input argument is a *DocPolynom* object — If you call the constructor function with an input argument that is already a DocPolynom object, the constructor returns a new DocPolynom object with the same coefficients as the input argument. The *isa* function checks for this situation. This is a so-called *copy constructor*.
- Input argument is a coefficient vector — If the input argument is not a DocPolynom object, the constructor attempts to reshape the values into a vector and assign them to the coef property.

The *coef* property set method restricts property values to doubles. See next section for a discussion of the property set method.

Removing Irrelevant Coefficients MATLAB software represents polynomials as row vectors containing coefficients ordered by descending powers. Zeros in the coefficient vector represent terms that drop out of the polynomial. Leading zeros, therefore, can be ignored when forming the polynomial.

Some *DocPolynom* class methods use the length of the coefficient vector to determine the degree of the polynomial. It is useful, therefore, to remove leading zeros from the coefficient vector so that its length represents the true value.

The *DocPolynom* class stores the coefficient vector in a property that uses a *set* method to remove leading zeros from the specified coefficients before setting the property value.

```
function obj = set.coef(obj,val)
  % coef set method
  if ~isa(val,'double')
    error('Coefficients must be of class double')
  end
  ind = find(val(:).'~=0);
  if ~isempty(ind)
    obj.coef = val(ind(1):end);
  else
    obj.coef = val;
  end
end
```

Example use An example use of the *DocPolynom* constructor is the statement:

```
p = DocPolynom([1 0 -2 -5])
p =
    x^3 - 2*x -5
```

This statement creates an instance of the *DocPolynom* class with the specified coefficients. Note how class methods display the equivalent polynomial using MATLAB language syntax. The *DocPolynom* class implements this display using the *disp* and *char* class methods.

1.11.4 Converting DocPolynom Objects to Other Types

The *DocPolynom* class defines two methods to convert DocPolynom objects to other classes:

- double — Converts to standard MATLAB numeric type so you can perform mathematical operations on the coefficients.
- char — Converts to string; used to format output for display in the command window

The DocPolynom to Double Converter The *double* converter method for the *DocPolynom* class simply returns the coefficient vector, which is a double by definition:

```
function c = double(obj)
  % DocPolynom/Double Converter
  c = obj.coef;
end
```

For the *DocPolynom* object *p*:

```
p = DocPolynom([1 0 -2 -5])
```

the statement:

```
c = double(p)
```

returns:

```
c =
    1 0 -2 -5
```

which is of class *double*:

```
class(c)
ans =
    double
```

The DocPolynom to Character Converter The *char* method produces a character string that represents the polynomial displayed as powers of an independent variable, x. Therefore, after you have specified a value for x, the string returned is a syntactically correct MATLAB expression, which you can evaluate.

The char method uses a cell array to collect the string components that make up the displayed polynomial.

The *disp* method uses *char* to format the *DocPolynom* object for display. Class users are not likely to call the char or disp methods directly, but these methods enable the *DocPolynom* class to behave like other data classes in MATLAB.

Here is the *char* method.

```
function str = char(obj)
  % Create a formatted display of the polynom
  % as powers of x
  if all(obj.coef == 0)
    s = '0';
  else
    d = length(obj.coef)-1;
    s = cell(1,d);
    ind = 1;
    for a = obj.coef;
      if a ~= 0;
        if ind ~= 1
          if a > 0
            s(ind) = {' + '};
            ind = ind + 1;
          else
            s(ind) = {' - '};
            a = -a;
            ind = ind + 1;
          end
        end
        if a ~= 1 || d == 0
          if a == -1
            s(ind) = {'-'};
            ind = ind + 1;
          else
            s(ind) = {num2str(a)};
            ind = ind + 1;
            if d > 0
              s(ind) = {'*'};
              ind = ind + 1;
            end
          end
        end
        if d >= 2
          s(ind) = {['x^' int2str(d)]};
          ind = ind + 1;
        elseif d == 1
          s(ind) = {'x'};
          ind = ind + 1;
        end
      end
      d = d - 1;
    end
  end
  str = [s{:}];
end
```

Evaluating the Output If you create the *DocPolynom* object *p*:

```
p = DocPolynom([1 0 -2 -5]);
```

and then call the *char* method on p:

```
char(p)
```

the result is:

```
ans =
    x^3 - 2*x - 5
```

The value returned by *char* is a string that you can pass to *eval()* after you have defined a scalar value for x. For example: x = 3;

```
x = 3;
eval(char(p))
ans =
    16
```

1.11.5 The DocPolynom disp method

To provide a more useful display of *DocPolynom* objects, this class overloads *disp* in the class definition.

This *disp* method relies on the *char* method to produce a string representation of the polynomial, which it then displays on the screen.

```
function disp(obj)
  % DISP Display object in MATLAB syntax
  c = char(obj);
  if iscell(c)
    disp(['     ' c{:}])
  else
    disp(c)
  end
end
```

The statement:

```
p = DocPolynom([1 0 -2 -5])
```

creates a *DocPolynom* object. Since the statement is not terminated with a semicolon, MATLAB calls the *disp* method of the *DocPolynom* object to display the output on the command line:

```
p =
    x^3 - 2*x - 5
```

1.11.6 Defining the + Operator

If either *p* or *q* is a *DocPolynom* object, the expression

```
p + q
```

generates a call to a function *@DocPolynom/plus*.

The following function redefines the *plus* (+) operator for the *DocPolynom* class:

```
function r = plus(obj1,obj2)
  % Plus Implement obj1 + obj2 for DocPolynom
  obj1 = DocPolynom(obj1);
  obj2 = DocPolynom(obj2);
  k = length(obj2.coef) - length(obj1.coef);
  r = DocPolynom([zeros(1,k) obj1.coef]+[zeros(1,-k) obj2.coef]);
end
```

Here is how the function works:

- Ensure that both input arguments are *DocPolynom* objects so that expressions such as

  ```
  p + 1
  ```

  that involve both a *DocPolynom* and a *double*, work correctly.

- Access the two coefficient vectors and, if necessary, pad one of them with zeros to make both the same length. The actual addition is simply the vector sum of the two coefficient vectors.

- Call the *DocPolynom* constructor to create a properly typed result.

1.11.7 Overloading MATLAB Functions roots and polyval for the DocPolynom Class

The MATLAB language already has several functions for working with polynomials that are represented by coefficient vectors. You can overload these functions to work with the new *DocPolynom* class.

In the case of DocPolynom objects, the overloaded methods can simply apply the original MATLAB function to the coefficients (i.e., the values returned by the *coef* property).

This section shows how to implement the following MATLAB functions.

Defining the roots function The *DocPolynom roots* method finds the roots of *DocPolynom* objects by passing the coefficients to the overloaded roots function:

```
function r = roots(obj)
  % roots(obj) returns a vector containing the roots of obj
  r = roots(obj.coef);
end
```

If *p* is the following *DocPolynom* object:

```
p = DocPolynom([1 0 -2 -5]);
```

then the statement:

```
roots(p)
```

gives the following answer:

```
ans =
     2.0946
    -1.0473 + 1.1359i
    -1.0473 - 1.1359i
```

Defining the polyval function The MATLAB *polyval* function evaluates a polynomial at a given set of points. The *DocPolynom polyval* method simply extracts the coefficients from the *coef* property and then calls the MATLAB version to compute the various powers of x:

```
function y = polyval(obj,x)
  % polyval(obj,x) evaluates obj at the points x
  y = polyval(obj.coef,x);
end
```

The following code evaluates the polynom $p = x^3$ for $x = -2 : 0.1 : 2$ and plots it.

```
p = DocPolynom( [ 1 0 0 0 ] )
p =
   x^3
x = -2:0.1:2;
y = polyval( p, x );
plot( x, y );
```

1.11.8 Complete DocPolynom example

The complete DocPolynom example class can be found in the MATLAB documentation code:

```
edit  ([docroot '/techdoc/matlab_oop/examples/@DocPolynom/DocPolynom.m']);
```

1.12 References

- http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_oop/ug_intropage.html
- http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matlab_oop.pdf

2 Interfacing with Java

2.1 Java Virtual Machine (JVM)

Every installation of MATLAB software includes Java Virtual Machine (JVM) software, so that you can use the Java interpreter via MATLAB commands, and you can create and run programs that create and access Java objects[3].

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 workspace
- Call Java object methods, using either Java or MATLAB syntax
- Pass data between MATLAB variables and Java objects

MATLAB can run Java code in the form of *.class* or *.jar* files.

2.2 The Java classpath

MATLAB loads Java class definitions from files that are on the Java class path. The class path is a series of file and directory specifications that MATLAB software uses to locate class definitions. When loading a particular Java class, MATLAB searches files and directories in the order they occur on the class path until a file is found that contains that class definition. The search ends when the first definition is found. The Java class path consists of two segments: the static path and the dynamic path. MATLAB loads the static path at startup. If you change the path you must restart MATLAB. You can load and modify the dynamic path at any time using MATLAB functions. MATLAB always searches the static path before the dynamic path. You can view these two path segments using the *javaclasspath* function.

2.2.1 Static classpath

To see which *classpath.txt* file is currently being used by your MATLAB environment, use the which function:

```
which classpath.txt
```

To edit either the default file or the copy in your own directory, type:

```
edit classpath.txt
```

[3]To use a different JVM than the one that comes with MATLAB, you now can set the MATLAB_JAVA system environment variable to the path of your JVM software.

2.2.2 Dynamic classpath

The dynamic class path can be loaded any time during a MATLAB software session using the *javaclasspath* function. You can define the dynamic path (using *javaclasspath*), modify the path (using *javaaddpath* and *javarmpath*), and refresh the Java class definitions for all classes on the dynamic path (using *clear* with the keyword *java*) without restarting MATLAB.

2.2.3 Adding JAR packages

In contrast to the static and dynamic classpaths, where the **directory** containing the *.class* file is added to the path, to make the contents of a JAR file available for use in MATLAB, specify the full path, including full file name, for the JAR file. You can also put the JAR file on the MATLAB path. For example, to make available the JAR file *e:\java\classes\utilpkg.jar*, add the following file specification to your static class path (i.e. the *classpath.txt* file):

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

or use the *javaaddpath* function to add it to the dynamic path:

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

2.3 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.

2.4 Creating and using Java objects

You create a Java object in the MATLAB workspace 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.

2.4.1 Constructing Java objects

You construct Java objects in the MATLAB workspace by calling the Java class constructor, which has the same name as the class. For example, the following constructor creates a *myDate* object:

```
myDate = java.util.Date
```

MATLAB displays information similar to:

```
myDate =
Thu Aug 23 12:58:54 EDT 2007
```

2.4.2 Invoking methods on Java objects

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

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

or the MATLAB syntax[4]:

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

In the following example, myDate is a *java.util.Date* object, and *getHours* and *setHours* are methods of that object.

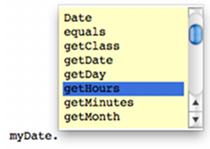
```
myDate = java.util.Date;
myDate.setHours(3)
myDate.getHours          // Java syntax
ans =
      3
getHours( myDate )       // MATLAB syntax
ans =
      3
```

2.4.3 Obtaining information about methods

MATLAB offers several ways 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 might 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.

[4]There is an alternative syntax, which makes use of the javaObject() and javaMethod() functions: myDate = javaObject('java.util.Date'); h = javaMethod('getHours', myDate);

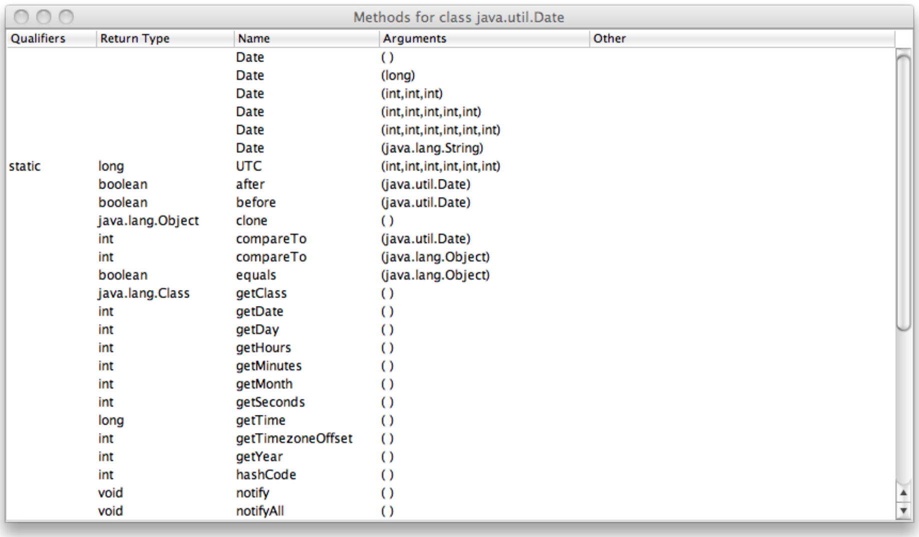
Tab key The simplest way is to type the name of a Java object on the MATLAB console followed by '.' and press the *Tab* key.



The methodsview function A more complete way is by using the *methodsview* function[5]:

```
methodsview java.util.Date       // Syntax 1
methodsview('java.util.Date')    // Syntax 2
myDate = java.util.Date;
methodsview(myDate)              // Syntax 3
```

A new window appears, listing one row of information for each method in the class.



Qualifiers	Return Type	Name	Arguments	Other
		Date	()	
		Date	(long)	
		Date	(int,int,int)	
		Date	(int,int,int,int,int)	
		Date	(int,int,int,int,int,int)	
		Date	(java.lang.String)	
static	long	UTC	(int,int,int,int,int,int)	
	boolean	after	(java.util.Date)	
	boolean	before	(java.util.Date)	
	java.lang.Object	clone	()	
	int	compareTo	(java.util.Date)	
	int	compareTo	(java.lang.Object)	
	boolean	equals	(java.lang.Object)	
	java.lang.Class	getClass	()	
	int	getDate	()	
	int	getDay	()	
	int	getHours	()	
	int	getMinutes	()	
	int	getMonth	()	
	int	getSeconds	()	
	long	getTime	()	
	int	getTimezoneOffset	()	
	int	getYear	()	
	int	hashCode	()	
	void	notify	()	
	void	notifyAll	()	

Each row in the window displays up to six fields of information describing the method:

- Qualifiers (i.e. method type qualifiers): abstract, synchronized, ...
- Return Type (i.e. type returned by the method): void, java.lang.String, ...
- Name (i.e. method name): getHours, addActionListener, ...
- Arguments (i.e. types of arguments passed to method): boolean, java.lang.Object, ...

[5]The *methodsview* function can also be used for MATLAB classes.

- Other (i.e. other relevant information): throws java.io.IOException, ...
- Inherited From (i.e. parent of the specified class): java.awt.MenuComponent, ...

The methods function The methods function returns information on methods of MATLAB and 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.

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

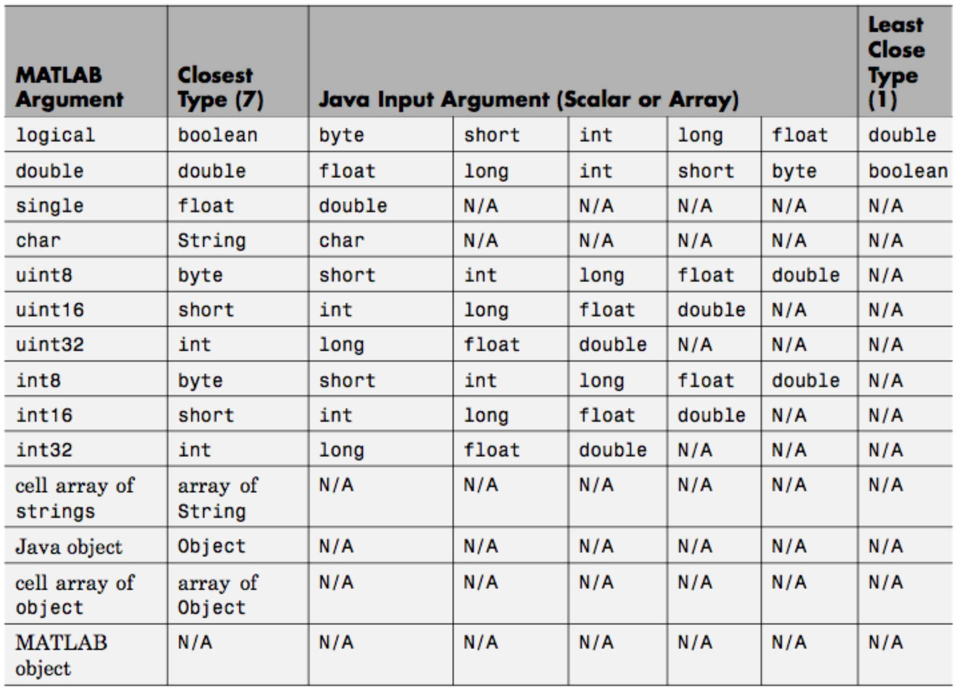
```
methods java.awt.Dimension -full

Methods for class java.awt.Dimension:
Dimension(int,int)
Dimension()
Dimension(java.awt.Dimension)
java.lang.Object clone()  % Inherited from java.awt.geom.Dimension2D
...
```

2.5 Passing arguments to and from a Java method

When you make a call to Java in MATLAB code, any MATLAB types you pass in the call are converted to types native to the Java language. MATLAB performs this conversion on each argument that is passed, except for those arguments that are already Java objects. If data is to be returned by the method being called, MATLAB receives this data and converts it to the appropriate MATLAB format wherever necessary.

2.5.1 Conversion of MATLAB data types

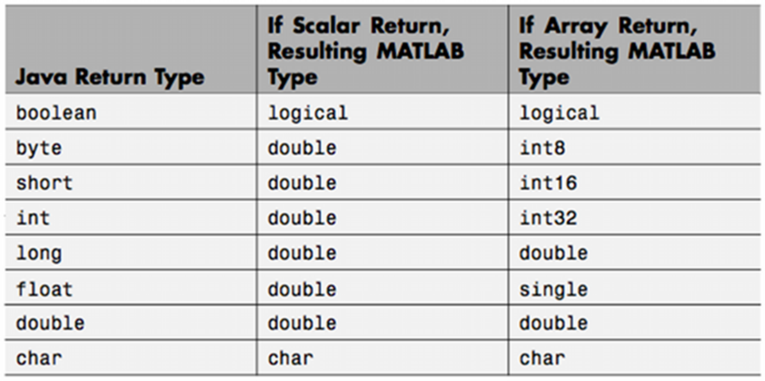
MATLAB data, passed as arguments to Java methods, are converted by MATLAB into types that best represent the data to the Java language. The table below shows all of the MATLAB base types for passed arguments and the Java base types defined for input arguments. Each row shows a MATLAB type followed by the possible Java argument matches, from left to right in order of closeness of the match. The MATLAB types (except cell arrays) can all be scalar (1-by-1) arrays or matrices. All of the Java types can be scalar values or arrays.



MATLAB Argument	Closest Type (7)	Java Input Argument (Scalar or Array)					Least Close Type (1)
logical	boolean	byte	short	int	long	float	double
double	double	float	long	int	short	byte	boolean
single	float	double	N/A	N/A	N/A	N/A	N/A
char	String	char	N/A	N/A	N/A	N/A	N/A
uint8	byte	short	int	long	float	double	N/A
uint16	short	int	long	float	double	N/A	N/A
uint32	int	long	float	double	N/A	N/A	N/A
int8	byte	short	int	long	float	double	N/A
int16	short	int	long	float	double	N/A	N/A
int32	int	long	float	double	N/A	N/A	N/A
cell array of strings	array of String	N/A	N/A	N/A	N/A	N/A	N/A
Java object	Object	N/A	N/A	N/A	N/A	N/A	N/A
cell array of object	array of Object	N/A	N/A	N/A	N/A	N/A	N/A
MATLAB object	N/A	N/A	N/A	N/A	N/A	N/A	N/A

2.5.2 Conversion of Java return data types

In many cases, data returned from a Java method is incompatible with the types operated on in the MATLAB environment. When this is the case, MATLAB converts the returned value to a type native to the MATLAB language. The following table lists Java return types and the resulting MATLAB types. For some Java base return types, MATLAB treats scalar and array returns differently, as described following the table.



Java Return Type	If Scalar Return, Resulting MATLAB Type	If Array Return, Resulting MATLAB Type
boolean	logical	logical
byte	double	int8
short	double	int16
int	double	int32
long	double	double
float	double	single
double	double	double
char	char	char

2.6 References

- http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f44062.html
- http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf

3 Interfacing with C/C++

3.1 MEX files

You can call your own C, C++, or Fortran subroutines from the MATLAB command line as if they were built-in functions. These programs, called binary MEX-files, are dynamically-linked subroutines that the MATLAB interpreter loads and executes. MEX stands for "MATLAB executable". In this course we won't discuss Fortran MEX files.

MEX-files have several applications:

- Calling large pre-existing C/C++ and Fortran programs from MATLAB without rewriting them as MATLAB functions
- Replacing performance-critical routines with C/C++ implementations

The second point has become less critical over the years, with MATLAB becoming faster and faster.

A computational routine is the source code that performs functionality you want to use with MATLAB. For example, if you created a standalone C program for this functionality, it would have a *main()* function. MATLAB communicates with your MEX-file using a *gateway routine*. The MATLAB function that creates the gateway routine is *mexfunction()*. You use *mexfunction()* instead of *main()* in your source file.

3.2 Overview of Creating a C/C++ Binary MEX-File

To create a binary MEX-file:

- Assemble your functions and the MATLAB API functions into one or more C/C++ source files.
- Write a gateway function in one of your C/C++ source files.
- Use the MATLAB *mex* function, called a build script, to build a binary MEX-file.
- Use your binary MEX-file like any MATLAB function.

3.3 Configuring your environment

Before you start building binary MEX-files, select your default compiler. In the MATLAB console, type:

```
>> mex -setup

  Options files control which compiler to use, the compiler and link command
  options, and the runtime libraries to link against.
  Using the 'mex -setup' command selects an options file that is
  placed in ~/.matlab/R2011a and used by default for 'mex'. An options
  file in the current working directory or specified on the command line
  overrides the default options file in ~/.matlab/R2011a.
  To override the default options file, use the 'mex -f' command
  (see 'mex -help' for more information).
The options files available for mex are:
  1: /Applications/MATLAB_R2011a.app/bin/gccopts.sh :
       Template Options file for building gcc MEX-files
  2: /Applications/MATLAB_R2011a.app/bin/mexopts.sh :
       Template Options file for building MEX-files via the system ANSI compiler

  0: Exit with no changes
Enter the number of the compiler (0-2):
1
/Applications/MATLAB_R2011a.app/bin/gccopts.sh is being copied to
/Users/aaron/.matlab/R2011a/mexopts.sh
**************************************************************************
  Warning: The MATLAB C and Fortran API has changed to support MATLAB
           variables with more than 2^32-1 elements.  In the near future
           you will be required to update your code to utilize the new
           API. You can find more information about this at:
           http://www.mathworks.com/support/solutions/en/data/1-5C27B9/?
             solution=1-5C27B9
           Building with the -largeArrayDims option enables the new API.
**************************************************************************
```

Now, MATLAB's *mex* is configured to use the selected compiler to build your MEX-files. We will discuss the Warning message from newer MATLAB versions below.

3.4 Using MEX-files to call a C program

Suppose you have some C code, called *arrayProduct*, that multiplies an 1-dimensional array *y* with *n* elements by a scalar value *x* and returns the results in array *z*. It might look something like the following:

```
void arrayProduct(double x, double *y, double *z, int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    z[i] = x * y[i];
  }
}
```

If $x = 5$ and y is an array with values 1.5, 2, and 9, then calling:

```
arrayProduct(x,y,z,n)
```

from within your C program creates an array z with the values 7.5, 10, and 45.
The following steps show how to call this function in MATLAB, using a MATLAB matrix, by creating the MEX-file *arrayProduct*.

3.4.1 Create a source MEX file

Open MATLAB Editor and copy your code into a new file. Save the file on your MATLAB path and name it *arrayProduct.c*. This file is your *computational routine*, and the name of your MEX-file is *arrayProduct*. We will now modify the code to turn it into a valid (and usable) MEX-file.

3.4.2 Create a *gateway routine*

At the beginning of the file, add the C/C++ header file:

```
#include "mex.h"
```

After the computational routine, add the gateway routine mexFunction[6]:

```
/* The gateway function */
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
  /* variable declarations here */

  /* code here */
}
```

mexFunction is the entry point for the MEX-file, i.e. what MATLAB will call when launching your MEX-file. This is currently just a placeholder. We will now add the content of the *gateway function*, but before that we will spend a couple of words on the signature of *mexFunction*.

```
void mexFunction( int nlhs,              // Number of input parameters
                  mxArray *plhs[],       // Array of pointers to inputs
                  int nrhs,              // Number of output parameters
                  const mxArray *prhs[]) // Array of pointers to const
                                         // outputs
{ ... }
```

Input parameters (found in the *prhs* array) are read-only; do not modify them in your MEX-file. Changing data in an input parameter can produce undesired side effects.
The MATLAB language works with only a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell

[6]This example is written in C and in C all variable declarations MUST be at the beginning of the function. This constraint does not exist in C++.

arrays, structures, and objects, are stored as MATLAB arrays. In C/C++, the MATLAB array is declared to be of type **mxArray**. The mxArray structure contains, among other things:

- Its type
- Its dimensions
- The data associated with this array
- If numeric, whether the variable is real or complex
- If sparse, its indices and nonzero maximum elements
- If a structure or object, the number of fields and field names.

3.4.3 Use preprocessor macros

The MX Matrix Library and MEX Library functions use MATLAB preprocessor macros for cross-platform flexibility. Edit your computational routine to use *mwSize* for **mxArray** size *n* and index *i*.

```
void arrayProduct(double x, double *y, double *z, mwSize n)
{
  mwSize i;
  for (i=0; i<n; i++)
  {
    z[i] = x * y[i];
  }
}
```

mwSize replaces *int* to ensure that **mxArray**s with more than $2^{32} - 1$ elements can be addressed correctly.

3.4.4 Verify Input and Output Parameters

In this example, there are two input arguments (a matrix and a scalar) and one output argument (the product). To check that the number of input arguments *nrhs* is two and the number of output arguments *nlhs* is one, put the following code inside the *mexFunction* routine:

```
/* check for proper number of arguments */
if(nrhs!=2)
{
  mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs",
    "Two inputs required.");
}
if(nlhs!=1)
{
  mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs",
    "One output required.");
}
```

The following code validates the input values:

```
/* make sure the first input argument is scalar */
if( !mxIsDouble(prhs[0]) ||
    mxIsComplex(prhs[0]) ||
    mxGetNumberOfElements(prhs[0])!=1 )
{
  mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notScalar",
    "Input multiplier must be a scalar.");
}
```

The second input argument must be a row vector.

```
/* check that number of rows in second input argument is 1 */
if(mxGetM(prhs[1])!=1) {
  mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notRowVector",
    "Input must be a row vector.");
}
```

3.4.5 Read input data

Put the following declaration statements at the beginning of your *mexFunction*:

```
double multiplier;    /* input scalar */
double *inMatrix;     /* 1xN input matrix */
mwSize ncols;         /* size of matrix */
```

Add these statements to the code section of *mexFunction*:

```
/* get the value of the scalar input */
multiplier = mxGetScalar(prhs[0]);
/* create a pointer to the real data in the input matrix */
inMatrix = mxGetPr(prhs[1]);
/* get dimensions of the input matrix */
ncols = mxGetN(prhs[1]);
```

3.4.6 Prepare output data

Put the following declaration statement after your input variable declarations:

```
double *outMatrix; /* output matrix */
```

Add these statements to the code section of *mexFunction*:

```
/* create the output matrix */
plhs[0] = mxCreateDoubleMatrix(1,ncols,mxREAL);
/* get a pointer to the real data in the output matrix */
outMatrix = mxGetPr(plhs[0]);
```

3.4.7 Perform Calculation

The following statement executes your function:

```
/* call the computational routine */
arrayProduct(multiplier,inMatrix,outMatrix,ncols);
```

3.4.8 Build the Binary MEX-File

You can build your MEX-file by typing[7]:

```
>> mex arrayProduct.c
```

If your source file is correct, mex should compile silently. In case something is wrong, mex will output error messages to the MATLAB console.

Test the MEX-File Type:

```
s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A)
```

This should output:

```
B =
    7.5000 10.0000 45.0000
```

Exercise Try the following:

```
s = 5;
A = [1.5 2; 9 11];
B = arrayProduct(s,A)
```

What do you get? Where is the mistake?

Exercise Try the following:

```
s = 3
A = uint8( [1, 5, 9] );
B = arrayProduct(s,A)
```

What do you get? Where is the mistake?

3.5 References

- http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f7667.html
- http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf

[7]To enable the new API for arrays with more than $2^{32} - 1$ elements, use: *mex arrayProduct.c -largeArrayDims*. Soon the new API will be enabled by default and you will not need to specify the *-largeArrayDims* option any more.