Christopher Lum
lum@u.washington.edu

Matlab Class Tutorial (DEPRECATED)

Introduction

This document is designed to act as a tutorial for creating a class object in Matlab. The user should be fairly comfortable with Matlab.

Note that this method of creating classes in Matlab is somewhat deprecated. Matlab now supports a more traditional method of creating classes (similar to C++ and C#) which is more flexible and should be used instead. Therefore, this document should only serve as a reference for those that are interested in another method of declaring a class.

Christopher Lum
lum@u.washington.edu

Creating the Object

The class is similar to a structure in the fact that it is an object with one or more fields. In this example, let's create an object of type 'particle'. This will have three fields

```
.x        = x position of particle
.y        = y position of particle
.time     = current time
```

In order to do this, we need to create a particle constructor. This must be a file called `particle.m` and located in the directory `@particle`. The directory which has the folder `@particle` must be in the Matlab path, but the actual folder `@particle` should not be in the path. This directory structure is shown below in Figure 1.

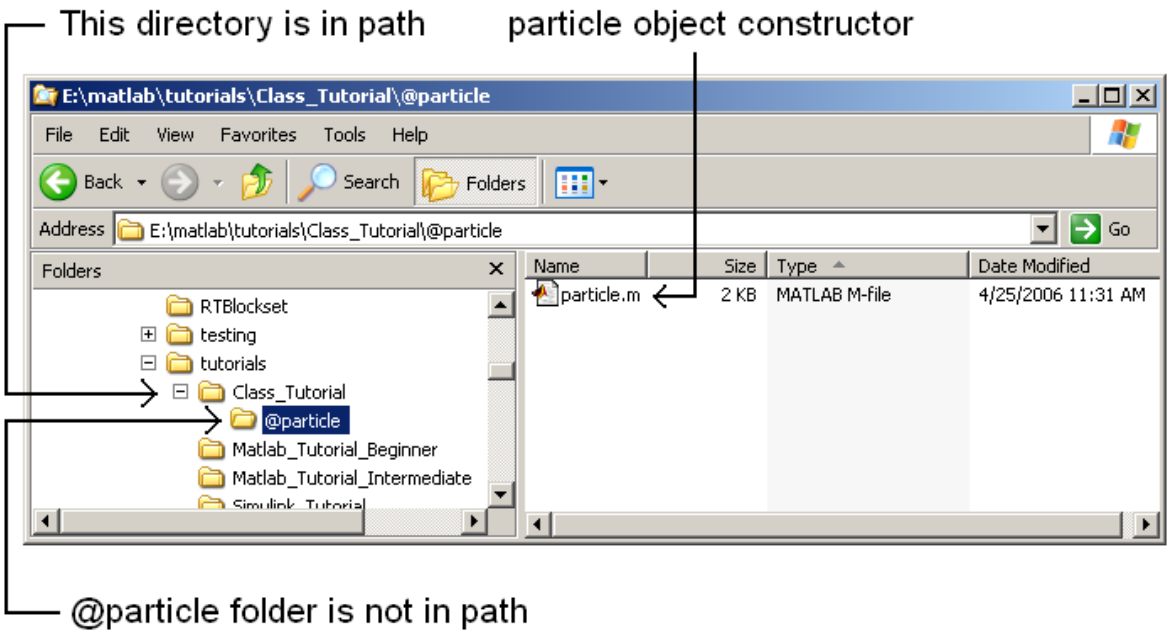

Figure 1: Directory structure and class constructor

The constructor `particle.m` is a function which must do three things

1. Create a default particle object when it is called with no arguments
2. If passed a particle object, simply return it.
3. Create a particle object when called with the appropriate number of arguments.

An example `particle.m` constructor is shown below in Figure 2.

```
function A = particle(varargin)

switch nargin
    case 0
        %No input, create default object
        A.x    = 0;
        A.y    = 0;
        A.time = 0;

        A = class(A, 'particle');

    case 1
        %If a single argument of class particle, return it
        if (isa(varargin{1},'particle'))
            A = varargin{1};
        else
            error('Input argument is not an particle object')
        end

    case 3
        %Create object using specified values
        A.x    = varargin{1};
        A.y    = varargin{2};
        A.time = varargin{3};

        A = class(A, 'particle');

    otherwise
        error('Invalid number of input arguments')

end
```

Figure 2: Sample particle.m file (particle object constructor)

With `particle.m` functioning correctly, we can create a particle object by either calling `particle(), particle(A),` or `particle(x,y,time)`. Let's use the last option (calling particle with three arguments corresponding to x, y, and time) as shown in Figure 3.

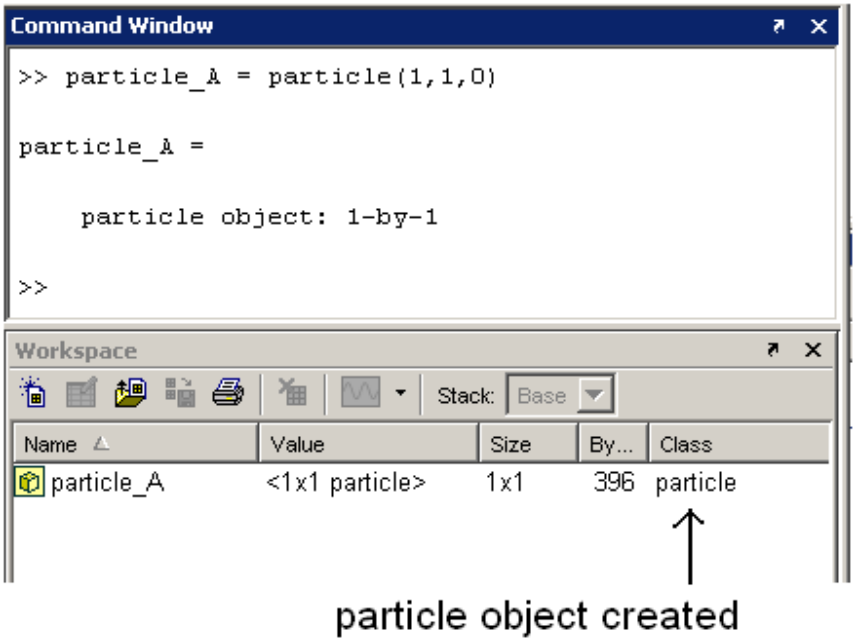

Figure 3: Creating a particle object

Adding Methods (Member Functions)

Just like C++, in order for a data object to be useful, it must have member functions which manipulate data in its fields and performs other useful operations with the data object. In Matlab, these member functions are known as "methods".

display.m

Notice in Figure 3 that we did not suppress the output with a semi-colon. When the semi-colon is left off, Matlab calls the function `display` with the object as its only argument to print the output to the screen. Just like C++, this function can be overloaded and the version of the function to call is based on the type of the argument. In order to overload the function so it is called when its argument is a particle object, place the file `display.m` in the `@particle` directory. An example `display.m` file is shown below in Figure 4.

```
function display(A)

disp(['.x:    ',num2str(A.x)])
disp(['.y:    ',num2str(A.y)])
disp(['.time: ',num2str(A.time)])
```

Figure 4: Sample display.m function

Now when the semi-colon is left off, the particle object is displayed to the screen in a more intelligent fashion (compare Figure 3 with Figure 5).

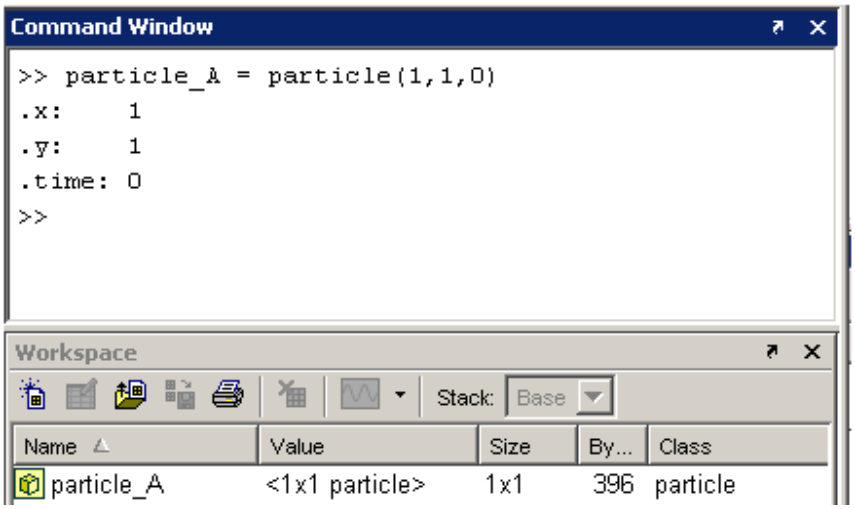

Figure 5: Now display.m in the @particle directory is called when semi-colon is omitted on a particle object

subsref.m

In Matlab, when you would like to access data inside a matrix or a structure, you would use the () or . operators. For example to access the 2,3 element of a matrix, you would use `A(2,3)`. Likewise, to access a certain field, one could use `A.field_name`. When you use syntax like this, Matlab actually calls the function `subsref` with the object as its only argument. Just like C++, this function can be overloaded and the version of the function to call is based on the type of the argument. In order to overload the function so it is called when its argument is a particle object, place the file `subsref.m` in the `@particle` directory. This function should do 3 things

1. Return the appropriate value when indexing is done using ()
2. Return the appropriate value when indexing is done using .
3. Return the appropriate value when indexing is done using {}

An example `subsref.m` file is shown below in Figure 6.

```
function b = subsref(a,index)

switch index.type
    case '()'
        switch index.subs{:}
            case 1
                b = a.x;

            case 2
                b = a.y;

            case 3
                b = a.time;

            otherwise
                error('Invalid index')
        end

    case '.'
        switch index.subs
            case 'x'
                b = a.x;

            case 'y'
                b = a.x;

            case 'time'
                b = a.time;

            otherwise
                error('Invalid field name')
        end

    case '{}'
        error('Cell array indexing not supported by particle objects')
end
```

Figure 6: Sample subsref function

Writing the `subsref` function in this fashion means that the user will be able to access all the data fields of the particle object using either () or . indexing (but not {} indexing).

Note that by default, data fields of a data object are "private". This means that if they are not included in the subsref function, then only particle methods (member functions) will have access to them. This may or may not be desirable depending on your application.

subsasgn.m

In addition to accessing data fields, we would like to also be able to assign values to data fields. In a matrix case, if we would like to assign a value of 6.2 to the 2,3 element, we would use `A(2,3) = 6.2`. Likewise, to write a certain field, one could use `A.field_name = 6.2`. When you use syntax like this, Matlab actually calls the function `subsasgn` with the object as its only argument. Just like C++, this function can be overloaded and the version of the function to call is based on the type of the argument. In order to overload the function so it is called when its argument is a particle object, place the file `subsasgn.m` in the `@particle` directory. This function should do 3 things

4. Assign the appropriate value when indexing is done using ()
5. Assign the appropriate value when indexing is done using .
6. Assign the appropriate value when indexing is done using {}

An example `subsasgn.m` file is shown below in Figure 7.

```
function a = subsasgn(a,index,val)

switch index.type
    case '()'
        switch index.subs{:}
            case 1
                a.x = val;

            case 2
                a.y = val;

            case 3
                a.time = val;

            otherwise
                error('Invalid index')
        end

    case '.'
        switch index.subs
            case 'x'
                a.x = val;

            case 'y'
                a.y = val;

            case 'time'
                a.time = val;

            otherwise
                error('Invalid field name')
        end

    case '{}'
        error('Cell array index assignment not supported for particle objects')
```

Figure 7: Sample subsasgn function

Other User Defined Methods

The four previously mentioned methods (`particle.m`, `display.m`, `subsref.m`, and `subsasgn.m`) are the most crucial to having a working class. Only `particle.m` is required, the others are merely recommended.

Table 1: Crucial methods for class

Name	**Purpose**
particle.m	Creates data object (constructor)
display.m	Displays data object
subsref.m	Allows functions other than particle methods to access data fields.
subsasgn.m	Allows functions other than particle methods to write to data fields

Other user defined methods (member functions) can be constructed in a similar fashion to add functionality to the class. All of these methods (member functions) must all be placed in the `@particle` directory.

Version History:	04/25/06: Created:	
	04/12/12: Updated:	Added note that this is effectively a deprecated technique to generate classes.