Object Oriented Programming and Classes in MATLAB[1]

1 Introduction

Object Oriented Programming (OOP) and Classes are two very important concepts in modern computer programming. They are often confused with each other but are really two separate concepts. One can do OOP in any programming language, even in C and Fortran, but in programming languages such as C++ and Java it is easier and "stricter" due to the presence of the notion of *class*. MATLAB just like C++ and Java allows one to write programs within the OOP paradigm using classes. If you have prior experience with C++ or Java, first see the appendix at the end of these notes before reading further.

1.1 Terminology and simple examples

First of all, what do we mean by OOP? OOP is really a way of thinking about computer programs and the data within them. It is a way of thinking which places the data of one's problem at the forefront. In its simplest sense it says that when you write a computer program you should first think about what the objects (data) of the problem will be (in their most abstract form) and what will be done with them. The program is then designed at this abstract level with the presumption that methods (functions) will be available to execute tasks on the objects. How the objects are actually stored or the methods written is not important at this stage. Importantly, if one can define the objects and the methods to act on them, then large teams of programmers can easily work together on a single software project without getting in the way of each other. One of the main reasons for this is that the only points of interaction are through the creation of objects and the application of methods upon them. Even if one is not working in a large team environment, OOP is a good way of thinking as it forces upon the programmer the notion that the important things in a program are the data objects that are manipulated inside the code. Think data first, function(method) second. Other benefits include forced modularity of code and ease of change over time. More succinctly

> Object Oriented Programming is the abstraction of ideas and items into data and the methods (functions) that operate on them.

A *class* is a template for ideas/items. It is composed of a definition of a data structure and methods that can operate on the data structure (if created). An *object* is an actual instance

[1] See MATLAB's online guide *MATLAB Object-Oriented Programming* for further technical details on OOP in MATLAB.

of a class. To use a class one only needs to know the interface to the class (i.e. the methods of the class).

1.1.1 Example: Rational Numbers

Suppose we are to write an arithmetic system that will deal only with rational numbers (i.e. the ratios of two integers). This is an ideal situation for creating a class (a template for rational numbers). The basic data for any rational number would be its numerator and its denominator. A possible thing we may want to do with rational numbers it to add them together – this will require a method.

1.1.2 Example: Card Game

Suppose we are to write a computer program that plays a game of cards against a user. In this situation there are many distinctly different classes of objects that we would probably like to define. For instance, it would be useful to have a class for a deck of cards. We could then easily create decks of cards and say shuffle and deal the cards out, if we have defined the methods shuffle and deal. We would also likely create a class for players and methods for players to turn over cards in the game etc.

2 Rational Numbers

There are a number of different options for creating classes in MATLAB. To keep things simple we will only examine one methodology and within that methodology we will concentrate only on the most basic options. A *class* within MATLAB is defined using a file called `classname.m` where `classname` is the name of the class you wish to define. Within this file the two most basic elements are: (1) a declaration of the data of the class (the properties) and (2) a constructor for your class. The properties is simply a list of the variables that you will use to reference the data stored in instances of your class (objects). The constructor is a method (function) with the same name as your class which contains the rules for storing the data associated with instances (objects) of the class. It is needed to create objects.

The rational number class is perhaps the simplest so we will use that for this example. We will name this class *ratnum*. Thus the class definition file will be called `ratnum.m`. The most rudimentary version will look like:

```
classdef ratnum
    % RATNUM -- class for rational numbers

    % Properties for the class
    properties (Access=protected)
        n     % Numerator
        d     % Denomenator
    end
```

```
    methods
        function r = ratnum(numerator,denomenator)
        % Usage: r = ratnum(numerator,denomenator)
        % Purpose: Constructor for rational number objects
        % Input: numerator   -- numerator for rational number
        %        denomenator -- denomenator for rational number
        % Output: r -- rational number object
            r.n = numerator;
            r.d = denomenator;
        end
    end
end
```

The file is delimited by the `classdef ratnum`/`end` pair which declares that this file will provide the class definition for **`ratnum`** objects. Next we have the `properties` block which is terminated by the `end` tag. The code in the `properties` block says that objects of this class will have two data elements **`n`** and **`d`**. The attribute **`(Access=protected)`** is optional but we will always employ it as it enforces a stricter object oriented framework on us. The `methods` block is terminated with an `end` tag. It contains a single method for our class, the constructor **`ratnum`**. The constructor has the same name as the class and takes two arguments, the values of the numerator and the denominator. It then stores them in what looks like a structure, **`r`**, and returns this as its output. MATLAB will interpret this variable as being of class **`ratnum`**.

If one were for instance to type in the command window

```
>> a=ratnum(1,3)
a =
  ratnum with no properties.
  Methods
```

MATLAB would create (construct) a rational number (1/3) and the result would be bound to the variable **`a`** which in the lingo is now a *ratnum object* (or instance of the ratnum class). Look in the workspace window and you will see that it is listed as being of class ratnum or try typing `class(a)`. Since we do not have any methods, beyond the constructor, in our class there is little that we can do with **`a`**. We can not even look at the numerator or denominator; attempting to do so generates an error.

```
 >> a.n
??? Getting the 'n' property of the 'ratnum' class is not allowed.
```

In a strict OOP framework one must have a method defined if one wants to do anything with an object. This is known as the concept of data encapsulation. The internal data of the object is (encapsulated) protected from other parts of the program. A user of a class does not need to know the internal storage details.

As a first method beyond the constructor, let us add the **`disp`** method to our class definition so that rational numbers can print in the command window. The only thing the method has

to do is print out the rational number. The methods within the class definition have access to the internal data of instances of the class so this is rather easy to write. A simple version is as follows:

```
function disp(r)
% Usage: disp(r)
% Purpose: display a rational number object
% Input: r -- rational number object
% Output: display the rational number
    if (r.d ~= 1)
        fprintf('%d/%d\n',r.n,r.d);
    else
        fprintf('%d\n',r.n);
    end
end
```

This method is placed within the `methods` block of our class definition file. If we now go back and create our rational number object, we will see `1/3` printed to the screen[2].

```
>> a=ratnum(1,3)
a =
1/3
```

By default, when an object is created in MATLAB and MATLAB needs to print it to the screen, MATLAB calls the `disp` method in the class definition file for the object. If you wish to directly print the object you can also call its `disp` method; e.g. `disp(a)`.

Continuing, let us now add some real functionality to the class. Let us create a method `add` which will add two rational numbers together and return the output as a rational number object. Formally the addition rule is given as

$$\frac{c}{d} + \frac{a}{b} = \frac{cb + ad}{db} \tag{1}$$

The required method should look like:

```
function r = add(r1,r2)
% Usage: r = add(r1,r2)
% Purpose: add two rational numbers
% Inputs: r1 -- rational number object
%             r2 -- rational number object
% Output: r -- Sum of r1 and r2 as a rational number object
    r = ratnum(r1.n*r2.d + r2.n*r1.d,r1.d*r2.d);
end
```

[2]Every time you change your class file you should type `clear classes` at the command prompt otherwise MATLAB will not be able to use your changes.

Note that the output is a rational number which we created by calling the rational number constructor `ratnum`. Also note that the method has access to the internal data of the two rational number objects. As an example, if one typed

```
>> a = ratnum(1,3);
>> b = ratnum(1,2);
>> c = add(a,b)
```

then `c` would be an instance of our class and the screen output would be `5/6`.

In class definition we have set up so far, there is no mechanism for looking at the numerator or the denominator of a rational number object nor is there a way to reset these values once an object has been instantiated (created). When designing a class one has to decide if these are useful features that you wish to give users of the class. If they are then they are added to the methods block of the class definition. Methods which allow you to query property values are known as *getters* and the methods that permit you to set property values are known as *setters*. For example, if we wanted to permit users to directly get and set the numerator and denominator of a rational number object, then we would define four separate methods for these purposes: `setN`, `setD`, `getN`, and `getD`. The appropriate code, placed within the methods block for the numerator getter and setter would like:

```
        function n = getN(r)
        % Usage: n = getN(r)
        % Purpose: Get the numerator of a rational number object
        % Input: r -- rational number object
        % Output: n -- the value of the numerator
            n = r.n;
        end

        function r = setN(r,numerator)
        % Usage: r = setN(r,numerator)
        % Purpose: Set the numerator of a rational number object
        % Input: r -- rational number object
        %        numerator -- new numerator value
        % Output: r -- reset rational number object
            r.n = numerator;
        end
```

Then the usage would then look like:

```
>> r = ratnum(3,7)
r =
3/7
>> getN(r)
ans =
     3
>> r = setN(r,5)
```

```
r =
5/7
```

Note that getters are quite common in class definitions. Setters on the other hand should only be set up after careful consideration – does the user really need direct access to the internal data of an object?

3 Example: Jabberwocks

The object oriented paradigm is very useful for dealing with non-numeric objects. This very simple example (adapted from SAMS Java 2.1) is meant to help you see that point. Consider the swordsman in the poem Jabberwocky by Lewis Carroll. He attacks the jabberwock with his vorpal blade; Carroll writes:

> One, two! One, two! And through and through
>
> The vorpal blade went snicker-snack!
>
> He left it dead, and with its head
>
> He went galumphing back.

In an object oriented framework the swordsman could be an instance of a `Knight` class. The `Knight` class would be the template for any type of knight with appropriate data to describe knights and methods for things they do. So, for instance, when the swordsman chops the head off the jabberwock, he could invoke a method to say "Hey! I chopped your head off. You are dead." The method could then call another method which would change the state of the jabberwock from alive to dead.

Just to keep this example simple let us only create the beginnings of a class for jabberwocks. The data for our generic jabberwock will be its color, sex, whether or not it is hungry, and whether or not it is dead. For methods, our display method will tell us the state of our jabberwocks. And we will have one other method to try and feed it. The class definition with the properties block will look like:

```
classdef jabberwock
    % JABBERWOCK -- Class definition file

    properties (Access=protected)
        color    % string for color
        sex      % string for sex
        hungry   % logical for hunger state
        alive    % logical for metabolic state
    end

end % end classdef
```

The data for a jabberwock will have 4 fields as described. The methods block will contain our constructor, display method, and our feed method. It is placed just after the properties block.

```
    methods
        function j = jabberwock(color,sex,hungry,alive)
            % Usage: j = jabberwock(color,sex,hungry,alive)
            % Purpose: Constructor for jabberwork objects
            %
            % Inputs: color      -- jabberwock color as a string
            %             sex    -- jabberwock sex as a string
            %             hungry -- logical variable indicating if the
            %                           jabberwock is hungry
            %             alive  -- logical variable indicating if the
            %                            jabberwock is alive
            % Output: j          -- jabberwock object

            j.color  = color;
            j.sex    = sex;
            j.hungry = hungry;
            j.alive  = alive;

        end

        function disp(j)
        % Usage: disp(j)
        % Purpose: Display the state of jabberwork objects
        %
        % Inputs: j --  jabberwock
        % Output: screen output of state
            if j.alive
                fprintf('This is a %s %s jabberwock which is alive.\n',...
                   j.sex,j.color);
                if j.hungry
                    fprintf('The jabberwock is hungry.\n');
                else
                    fprintf('The jabberwock is full.\n');
                end
            else
                fprintf('This was a %s %s jabberwock which is now dead.\n',...
                    j.sex,j.color);
            end
        end

        function j=feed(j)
        % Usage: j=feed(j)
        % Purpose: Feed a jabberwork
        %
        % Inputs:  j --  jabberwock
        % Output:  j --  Full jabberwock if alive
```

```
            if j.alive
                if j.hungry
                    fprintf('Yum -- a peasant!\n');
                    j.hungry = false;
                else
                    fprintf('No, thanks -- already ate.\n');
                end
            else
                fprintf('Are you crazy! This jabberwock is dead!\n');
            end
        end

    end % end methods block
```

A sample test run looks as follows

```
>> j=jabberwock('blue','male',true,true);
>> disp(j);
This is a male blue jabberwock which is alive.
The jabberwock is hungry.
>> j=feed(j);
Yum -- a peasant!
>> disp(j);
This is a male blue jabberwock which is alive.
The jabberwock is full.
>> j=feed(j);
No, thanks -- already ate.

% Create a new jabberwock
>> a=jabberwock('red','female',false,false);
>> disp(a);
This was a female red jabberwock which is now dead.
>> a=feed(a);
Are you crazy! This jabberwock is dead!
```

With the addition of other methods etc. we could build software containing jabberwocks and treat them abstractly as data objects without having to worry about the implementation details (as long as we don't change the interface itself).

4 Overloading

Overloading is the idea that a given method name will do different things depending upon the class of its input arguments. We have already seen this in action with the `disp.m` method. Every time MATLAB tries to display any object it uses the display method defined within the corresponding class definition; i.e. it first looks up the classname of the object it is trying to display and then it executes the appropriate method upon the object. This is enormously useful for a number of reasons the most obvious being that we don't have to invent different

names for methods that essentially do the same thing on different objects (such as display them).

MATLAB even allows one to overload operators like `+` and `*` etc. This is done by creating methods with special reserved names. For instance if we have two rational numbers, r1 and r2, we can add them using `add(r1,r2)`. But it would be nice to add them by typing `r1 + r2`. This is accomplished by using MATLAB's reserved name, **`plus`**, for `+`. Thus all we need to do is rename our addition method **`add`** to **`plus`**. MATLAB will then interpret statements like `r1 + r2` as `plus(r1,r2)`. Other reserved names can be found by typing `help matlab/ops` (scroll to the top of the list and you will see names on the left that are the reserved names corresponding to the operator on the right).[3]

5 Inheritance

Inheritance is another important concept in OOP. In many situations, one will want to create a number of classes that have a set of common properties and associated methods. Inheritance is a way in which you can recognize this commonality and take advantage of it. Inheritance is generally a very complex subject and here we will only look at it in its simplest form. In MATLAB if a class (say 'class_a') *inherits* from another class (say 'class_b'), then you can treat instances of 'class_a' just as though they were instances of 'class_b'. In particular you can use methods from the class definition of 'class_b' on objects of 'class_a'.

The essential set up in MATLAB is that in the constructor for 'class_a' you explicitly instantiate your object as being of class 'class_b' and then set up the 'class_a' data. Note this is different from having the data for 'class_a' contain a member of 'class_b'.

To set up inheritance one needs to first set indicate that 'class_a' will inherit from 'class_b'. This is done on the `classdef` line as:

```
classdef class_a < class_b
```

Then in the constructor for 'class_a', we need the following

```
function a = class_a(init_data_for_a,init_data_for_b)
% Usage: a = class_a(init_data_for_a,init_data_for_b)
%
% Purpose: constructor for class_a with inheritance from class_b
%
% Inputs: init_data_for_a -- data to initialize class_a objects
%         init_data_for_b -- data to initialize class_b objects

  a = a@class_b(init_data_for_b);  % call the superclass constructor
  a.data = init_data_for_a;              % set the class_a data
```

[3]This concept should give you the added insight that the arithmetic binary and unary operators are really just methods in disguise.

```
end
```

At this stage, if there are any methods for 'class_b' we can use them with instances of 'class_a'; e.g.

```
>> a = class_a(data,init_b);
>> class_b_method(a);
```

would be perfectly legal. On the other hand if we did not use inheritance and simply added an object of 'class_b' to data structure of 'class_a' then this would not be legal; i.e. if the constructor looked like

```
function a = class_a(init_data_for_a,init_data_for_b)

  a.b = class_b(init_data_for_b);
  a.data = init_data_for_a;

end
```

then to apply 'class_b' methods onto `a.b` would require a specific 'class_a' method that had access to the internal structure of instances of 'class_a'. Put another way, the functionality that inheritance provides can substantially simplify coding. There are many more features of inheritance but we will leave those topics for a second course.

5.1 Inheritance Example

Perhaps the best way to get a basic understanding of inheritance is to look at a simple example. Suppose we are writing a program that deals with geometric shapes like circles and rectangles. All of our shapes will have a center position and a color but beyond that circles and rectangles differ from each other in data and even perhaps methods we would like to apply to them. It makes sense then to have a shape class with things that are common to circles and squares and let separate classes for circles and squares deal only with the specific differences between them. A possible shape class could look like:

```
classdef shape
    properties (Access=protected)
        x
        y
        color
    end
    methods
        function s=shape(x,y,color)
            s.x = x;
            s.y = y;
            s.color = color;
        end
```

```
        function disp(s)
            fprintf('The shape is centered at (%f,%f) and has color %s\n',...
                s.x,s.y,s.color);
        end

        function color=get_color(s)
            color = s.color;
        end
    end
end
```

This class has a constructor, a display method, and a getter for the color.

Then for our circle class we could inherit from shape as:

```
classdef circle < shape
    properties (Access=protected)
        r
    end

    methods
        function c = circle(radius,x,y,color)
            c = c@shape(x,y,color);  % Special construct for instantiating the
                                     % superclass
            c.r = radius;
        end

        function disp(c)
            disp@shape(c);  % Call the superclass display first (optional)
            fprintf('Radius = %f\n',c.r);
        end

        function a = area(c)
            a = pi*c.r^2;
        end
    end
end
```

With this set up `circle` inherits from `shape`. And for our rectangle class we could have:

```
classdef rect < shape
    properties (Access=protected)
        h
        w
    end

    methods
        function r = rect(height,width,x,y,color)
            r = r@shape(x,y,color);  % Special construct for instantiating the
                                     % superclass
```

```
            r.h = height;
            r.w = width;
        end

        function disp(r)
            disp@shape(r);  % Optional call to the superclass display method
            fprintf('Height = %f and Width = %f\n',r.h,r.w);
        end

        function a = area(r)
            a = r.w*r.h;
        end
    end
end
```

If we now create instances of rectangles or circles we can utilize and methods for shapes on them. For example `get_color(c)` would be perfectly legal if `c` were either an instance of a rectangle or a circle (or a shape for that matter). For example:

```
>> c = circle(3,1,1,'blue')
c =
The shape is centered at (1.000000,1.000000) and has color blue
Radius = 3.000000
>> get_color(c)
ans =
blue
>> rect=rect(1,2,0,0,'Black')
rect =
The shape is centered at (0.000000,0.000000) and has color Black
Height = 1.000000 and Width = 2.000000
>> get_color(rect)
ans =
k
```

Note that the superclass generates the part of the display text that is common for all shapes and then the subclass takes care of the part of the display that is unique to the type of shape. `get_color` is a method that is common to all shapes and since our two classes, `circle` and `rect`, have inherited from `shape`, they have access to this method without having to directly implement it in each subclass.

A C++ and Java programmers

Tips for C++ and Java Programmers If you are accustomed to programming in other object-oriented languages, such as C++ or Java, you will find that the MATLAB programming language differs from these languages in some important ways:

1. Dot syntax is available in MATLAB. Thus an alternate to `method(object)` is `object.method()`.

2. In MATLAB it is possible to make properties public.

3. In MATLAB there are no implicit parameters to methods. Methods that act on objects must have them declared in the function header.

4. In MATLAB, method dispatching is not syntax based, as it is in C++ and Java. When the argument list contains objects of equal precedence, MATLAB uses the left-most object to select the method to call.

5. To call a superclass method one uses the syntax `method@superclass` as opposed to `superclass::method` (in C++) or `superclass.method` (in Java).

6. In MATLAB, the destructor method is called `delete`. For simple classes it is not needed.

7. In MATLAB, there is no passing of variables by reference. When writing methods that update an object, you must pass back the updated object and use an assignment statement. The exception to this rule is the case where your class inherits from the `handle` class.

8. In MATLAB, there is no equivalent to C++ templates or Java generics.