14:440:127– Introduction to Computers for Engineers

Notes for Lecture 11

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 Object Oriented Programming- I'm Down With OOP

`http://www.cs.ubc.ca/~murphyk/Software/matlabTutorial/html/objectOriented.html` contains a good primer for this section since it's too new to be covered in your book. You can also see the official Matlab page at: `http://www.mathworks.com/products/matlab/object_oriented_programming.html`

Sometimes when you're designing a Matlab program, you come across a situation where it's easiest to think about the type of data you're using as some sort of 'object' or 'thing.' For instance, if you were working for the New Jersey Devils hockey team and creating a program to track everything you know about the players, you could think about a 'hockey player' as the object you're creating. Each 'hockey player' has a number of properties, or variables– his or her name, age, address, weight, and so on. You might recall the 'struct' data type, which I previously mentioned as a natural fit for databases like this in terms of having different fields for each variable. However, we can now use a paradigm called object-oriented programming (OOP for short) to go a step beyond 'structs.'

Whereas each 'struct' was basically just a collection of different variables all tied together, an 'object' also has functions associated with that variable. For instance, for our 'hockey player,' we might have functions to display that player's statistics, or perhaps to trade the player. Having the ability to create functions for our object is the main difference between objects and structs.

We define objects by creating a 'class.' To continue with our example, we could create a class called 'hockeyplayer.' We then need to list all of the variables each 'hockeyplayer' will have and define the functions that hockey player would have inside a class definition file, which in general appear as follows (replacing everything all in capitals):

```
classdef CLASSNAME
    properties % variables
       VARIABLE1;
       VARIABLE2;
    end

    methods  % functions
        function obj = CLASSNAME( )  % constructor
                % here, put code to create the class
                % to set VARIABLE1 equal to 5, do
                % obj.VARIABLE1 = 5 since obj is the output
        end
        function obj = FUNCTION2(obj)
            % here, you have code for another function.
            % use obj.VARIABLE2 to access VARIABLE2
        end
    end
end
```

The most puzzling part of this example might be what I identify as the 'constructor' function. The idea of a constructor is that it's the function that creates the object and initializes all of the variables. For that reason, it needs to have the same name as the class. You could have a constructor that just assigns values to the variables by itself, or perhaps a constructor that takes inputs and uses those to set all of the variables. In the hockeyplayer example that follows, you'll see a constructor that first looks to see if values were given as inputs to the function and then assigns values if there aren't. All of your classes should have constructor functions.

You also might wonder why a variable called 'obj' is the output of the constructor function and why we need to use `obj.VARIABLE1 = 5` to set VARIABLE1 equal to 5. Basically, the constructor needs to return an object (what's just been created), so we need to create this object, in this case using the arbitrary variable name 'obj.'

Now, for FUNCTION2, you might be curious why obj is both an input and an output. 'obj' as in input will represent the object itself, meaning that we can access all of the properties it has. You can call this function for an object by typing either FUNCTION2(x) or x.FUNCTION2 is x is a CLASSNAME object.

Let's look at a hockey player class, which has three variables (name, age, goals) and three functions (the constructor, trade, and makeolder).

```
classdef hockeyplayer
    properties  % these are the variables
       name;
       age;
       goals;
    end

    methods  % these are the functions
        function obj = hockeyplayer(a,b,c)  % constructor
            if(nargin==3)
                obj.name = a;
                obj.age = b;
                obj.goals = c;
            else
                obj.name = 'Marty Brodeur';
                obj.age = 37;
                obj.goals = 2;
            end
        end
        function obj = trade(obj) % trade him
            fprintf('We have traded %s\n',obj.name)
        end
        function obj = makeolder(obj) % trade him
            obj.age = obj.age+1;
        end
    end
end
```

We can now create hockey players as follows, noting that what I type is preceded by `>>`. In this example, we'll create a hockey player as the variable 'x' using the default values for the constructor, and also a hockey player 'y' using values we supply for the variables:

```
>> x = hockeyplayer
  x =
  hockeyplayer
  Properties:
     name: 'Marty Brodeur'
      age: 37
    goals: 2
>> x.name
  ans =
  Marty Brodeur

>> y = hockeyplayer('Colin White',32,0)
  y =
  hockeyplayer
  Properties:
     name: 'Colin White'
      age: 32
    goals: 0
>> y.trade;
   We have traded Colin White

>> x.makeolder
  ans =
  hockeyplayer
  Properties:
     name: 'Marty Brodeur'
      age: 38
    goals: 2
>> x.age
  ans =
    37
```

1.1 Updating Values- 1

Everything looked good in this example until we made Marty Brodeur older, yet his age didn't seem to change. This is because Matlab doesn't automatically update objects, using what's called a 'pass-by-value' approach to functions. How do we fix this? One option is to always save the updated object, as follows:

```
>> y = hockeyplayer('Colin White',32,0)
  y =
  hockeyplayer
  Properties:
     name: 'Colin White'
      age: 32
    goals: 0
>> y = y.makeolder;
>> y = y.makeolder;
>> y.age
  ans =
    34
```

1.2 Updating Values- 2

Another way of making sure Matlab updates values you change in any functions (AKA methods) in a class is to tell the class to implement the behavior of the 'handle' class, which tells Matlab to use 'pass-by-reference' behavior, meaning that if a variable is given to a function as an input and the function modifies that variable, it's actually modifying the original. We can make the following edits to the class to give it the desired behavior:

```
classdef hockeyplayer < handle  % CHANGE
    properties
       name;
       age;
       goals;
    end
    methods
        function obj = hockeyplayer(a,b,c)
            if(nargin==3)
                obj.name = a;
                obj.age = b;
                obj.goals = c;
            else
                obj.name = 'Marty Brodeur';
                obj.age = 37;
                obj.goals = 2;
            end
        end
        function [ ] = trade(obj) % CHANGE
            fprintf('We have traded %s\n',obj.name)
        end
        function [ ] = makeolder(obj) % CHANGE
            obj.age = obj.age+1;
        end
    end
end
```

And now the following works:

```
>> y = hockeyplayer('Colin White',32,0)
  y =
  hockeyplayer
  Properties:
     name: 'Colin White'
      age: 32
    goals: 0
>> y.makeolder;
>> y.makeolder;
>> y.age
  ans =
    34
```

2 Ball Bouncing Example

Now, let's make a more complicated example of a class for a bouncing ball. We'll keep track of its position (x and y), its velocity (vx and vy) and its radius (r). We'll have a constructor (bball), a function for moving the ball (move), and a function for returning points that can be graphed and create a circle of radius r centered around the point (x,y).

```
classdef bball < handle
    properties  % these are the variables
       x;
       y;
       vx;
       vy;
       r;
    end
    properties(Constant)  % these are the constants
       minx = -10;
       maxx = 10;
       miny = -10;
       maxy = 10;
    end

    methods  % these are the functions for objects of type "bball"
        function obj = bball( )  % constructor--- make a single ball
            obj.x = 18*rand(1)-9;
            obj.y = 18*rand(1)-9;
            obj.vx = rand(1)-.5;
            obj.vy = rand(1)-.5;
            obj.r = rand(1)*.9 + .1;
        end
        function [xpts ypts] = getpts(obj)  % return points (Circle) to plot
            t = 0:.1:2*pi;
            xpts = obj.r*cos(t) + obj.x;
            ypts = obj.r*sin(t) + obj.y;
        end
        function [] = move(obj) % move the ball
            obj.x = obj.x + obj.vx;
            if(obj.x>(obj.maxx-obj.r)) % hit right wall
                obj.vx = -obj.vx;
                obj.x = obj.x + obj.vx;
            end
            if(obj.x<(obj.minx+obj.r)) % hit right wall
                obj.vx = -obj.vx;
               obj.x = obj.x + obj.vx;
            end
            obj.y = obj.y + obj.vy;
            if(obj.y>(obj.maxy-obj.r)) % hit top
               obj.vy = -obj.vy;
               obj.y = obj.y + obj.vy;
            end
            if(obj.y<(obj.miny+obj.r)) % hit bottom
               obj.vy = -obj.vy;
               obj.y = obj.y + obj.vy;
            end
        end
    end
end
```

We save the file above as 'bball.m'. To use this class to make a vector full of bouncing balls (and thus display 40 at once), we can use the following code:

```
% make 40 balls
for k = 1:40
    red(k) = bball;
end
for j = 1:100000 % each loop is one movement for all balls
    xplot = [ ];
    yplot = [ ];
    for k = 1:40
        red(k).move;  % move the 40 balls
        [x y] = red(k).getpts( ); % get points to plot
        xplot = [xplot x];
        yplot = [yplot y];
    end
    plot(xplot,yplot,'rx'); axis([-10 10 -10 10])
    pause(.01)
end
```