CS 1114:
Programming clarity via Objects - part 1

Prof. Graeme Bailey

http://cs1114.cs.cornell.edu

(notes modified from Matt Dunham, http://www.advancedmcode.org/object-oriented-programming-in-matlab.html)

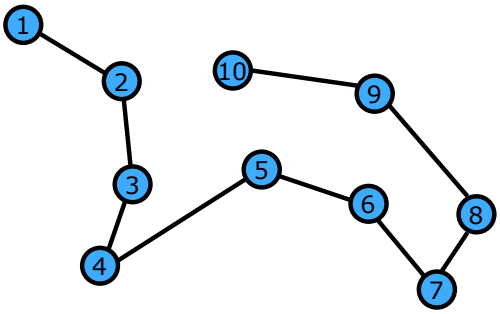
Cornell University
Computer Science

Why Objects?

- For an image, points may have colour and coordinates. Three options:
  - Maintain 3 arrays in sync for x-coords, y-coords and colours … bad if get out of sync.
  - Maintain 1 array for these values, putting all the x-coords in positions 1, 4, 7, 10,… , the y-coords in 2, 5, 8, 11,… , and the colours in 3, 6, 9, 12,… Makes for some tricky arithmetic to keep track.
  - Make objects which have 3 properties (x-coord, y-coord, colour) and have 1 array for these point objects. This is vastly easier to maintain.

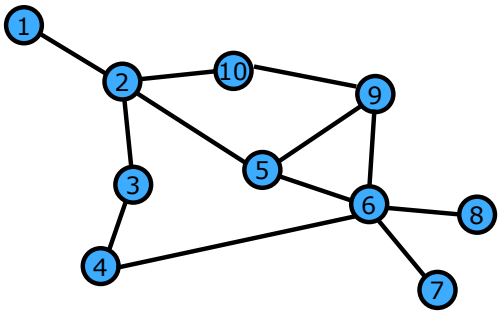
Why Objects?

- For linked lists, there could be
  - node objects:
    - A node has 3 properties: value, next, previous.
  - linked list objects:
    - A linked list has 2 properties: first_node, length
  - Manipulating a list is then relatively easy.



Why Objects?

- For graphs, there could be
  - graph objects:
    - A graph has 1 property: list of nodes.
  - node objects:
    - A node has 2 properties: value, list of adjacent nodes.



Objects in MATLAB

- Classes are manufacturers of things/objects

```
classdef point
  % write a description of the class here.
    properties
      % define the properties of the class here.
        x_coord; % integer values only
        y_coord; % integer values only
        colour; % one integer between 0 and 255
    end
    methods
      % methods, including the constructor are defined here
        function obj = point(x, y, colour)
          % class constructor
            obj.x_coord = x;
            obj.y_coord = y;
            obj.colour  = colour;
        end
    end
end
```

*These are **single** entities, each holding three values nicely packaged*

*Use the class to **construct** objects:*

```
centre = point(250, 375, 255);
median = point(125, 400, 200);
```


Objects in MATLAB

- We can access the packaged information:

```
centre = point(250, 375, 255);
median = point(125, 400, 200);

centre.x_coord

median.y_coord = 96;
median.colour = median.colour/2;

X_avg = (centre.x_coord + median.x_coord)/2;
Y_avg = (centre.y_coord + median.y_coord)/2;
C_avg = (centre.colour + median.colour)/2;

average = point(X_avg, Y_avg, C_avg);
```

Accesses and prints out centre's x coord

Changes the value of median's y coord

Accesses and then updates the value of median's colour

- This much easy access can be dangerous....

Protecting Access

```
classdef point
  % write a description of the class here.
    properties (SetAccess = 'private', GetAccess = 'private')
      % ....
    end
    methods
      % methods, including the constructor are defined here
        % class constructor
        function obj = point(x, y, colour)
          % ....
        end

        % gettors and settors
        function obj = set_x(obj, x_value)
            obj.x_coord = x_value;
        end % use via w = w.set_x(29); to overwrite the original object w
        function x_value = get_x(obj)
            x_value = obj.x_coord;
        end % can use via w.get_x() or a = w.get_x();
        % ... etc for y_coord and colour ...
    end
end
```

These can't be seen or changed from outside – prevents accidental 'updates'

Then have specific functions/methods to access the private values

There's a `set` *and* `get` *in old* MATLAB. *Please don't use them, they behave very badly!*

***Very ugly** – see later fix via handles*

We can also have private methods – typically used internally to the class to keep other methods shorter and cleaner

Default Construction

- It helps to have a more generic constructor:

```
classdef point
  % write a description of the class here.
    properties
      % ....
    end
    methods
      % methods, including the constructor are defined here
        function obj = point(x, y, colour)
          % class constructor
          if (nargin > 0) % if input values were given
            obj.x_coord = x;
            obj.y_coord = y;
            obj.colour  = colour;
          else % essentially allows point() as a default constructor
            obj.x_coord = 0;
            obj.y_coord = 0;
            obj.colour  = 0;
          end
        end
        % .... more functions here
    end
end
```

This part of the constructor is used to initialise an object with specific concrete values

This part is only invoked if there are no input variables to the constructor

Class Constants

- It's useful to have class constants:

```
classdef point
  % write a description of the class here.
    properties (Constant = true)
      % define the constant (or final) values here
        X_ORIGIN = 0;
        Y_ORIGIN = 0;
        BLACK = 0;
        WHITE = 255;
    end
    properties
      % ....
    end
    methods
      % methods, including the constructor are defined here
        function obj = point(x, y, colour)
          % ....
        end
        % .... more functions here
    end
end
```

This is to set constants for the class (by convention we use upper case). Notice we can have multiple blocks of properties (ditto functions/methods).

Try typing `properties(point)` *in* MATLAB

Pointers to Efficiency

- MATLAB's default behaviour is very wasteful
  - It likes to copy everything!!
  - Called 'passing by value'
  - x = y is done by copying all of y onto x
  - This is ok if y is a number, but what if it's a 10000x10000 matrix?
  - This happens also when passing values into a function! .....S.....L.....O.....W.....

- Far better to pass the *address* of the data!!!
  - Called 'passing by reference'
  - MATLAB has a handle class we can 'steal from':

*Handl*ing Operator Overloading

- We can redefine + (and *, <, &, ||, etc)!

```
classdef point < handle
  % write a description of the class here.
    properties
      % ....
    end
    methods
      % methods, including the constructor are defined here
      function obj = point(x, y, colour)
        % ....
      end
      function total = plus(a, b)
        temp_x = a.x_coord + b.x_coord;
        temp_y = a.y_coord + b.y_coord;
        temp_c = max(a.colour, b.colour);
        total = point(temp_x, temp_y, temp_c);
      end
      % .... more functions here
    end
end
```

Really elegant – allows passing of addresses instead of full copying of all values; vastly faster and less memory-intensive.

This overides the usual meaning of `+` *and so would be used as follows:*

```
a = point(200, 300, 125);
b = point(159, 203, 224);
c = a + b;
```

giving `c` *the values: 259, 503, and 349 for* x_coord, y_coord, *and* colour

Also makes updating values look cleaner:

```
a = pointy(200, 300, 125);
a.set_colour(98); % changes a's colour to 98
```


Pointers to Efficiency

```
classdef pointy < handle %%% this makes a HUGE difference!!!
    % This class is a pointer-based implementation (via subclassing the
    % handle class) of an image point.  It has fields for x and y coords
    % as well as a field for b/w colour value (0-255, not enforced).  In
    % addition, the + operator has been overwritten to allow addition of
    % points in the usual way (though the colour value chosen is the max
    % of the two input colours).  Being pointer-based, pointy objects are
    % passed by reference, not by value, hence saving on memory and
    % processing time.

    properties (Constant = true)
        % define the constant (or final) values here
            X_ORIGIN = 0;
            Y_ORIGIN = 0;
            BLACK = 0;
            WHITE = 255;
    end
    properties (GetAccess = 'private', SetAccess = 'private')
        % define the properties of the class here.
            x_coord; % integer values only
            y_coord; % integer values only
            colour; % one integer between 0 and 255
    end % ending properties blocks

    methods
        % methods, including the constructor are defined here
            % class consturctor
                function obj = pointy(x, y, colour)
                    if (nargin > 0) % if input values were given
                        obj.x_coord = x;
                        obj.y_coord = y;
                        obj.colour  = colour;
                    else % essentially allows point() as a default constructor
                        obj.x_coord = 0;
                        obj.y_coord = 0;
                        obj.colour  = 0;
                    end
                end
            % gettors and settors
                function set_x(obj, x_value)
                    obj.x_coord = x_value;
                end % use via w = w.set_x(29) to overwrite the original object w
                function x_value = get_x(obj)
                    x_value = obj.x_coord;
                end % can use via w.get_x() or a = w.get_x();

                function set_y(obj, y_value)
                    obj.y_coord = y_value;
                end % use via w = w.set_x(29) to overwrite the original object w
                function y_value = get_y(obj)
                    y_value = obj.y_coord;
                end % can use via w.get_x() or a = w.get_x();

                function set_colour(obj, color)
                    obj.colour = color;
                end % use via w = w.set_x(29) to overwrite the original object w
                function color = get_colour(obj)
                    color = obj.colour;
                end % can use via w.get_x() or a = w.get_x();
            % overiding standard operators
                function total = plus(a, b)
                    temp_x = a.x_coord + b.x_coord;
                    temp_y = a.y_coord + b.y_coord;
                    temp_c = max(a.colour, b.colour);
                    total = pointy(temp_x, temp_y, temp_c);
                end
    end % ending method block
end
```