CS 1114:
Objects, Events and Recursion

Prof. Graeme Bailey

http://cs1114.cs.cornell.edu

Cornell University
Computer Science

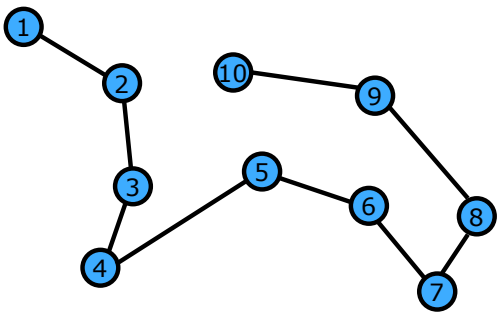
Object behaviours

- Passing by reference:
  - Since `a = pointy(1,2,3)` gives `a` the *address* of the data content, writing `b = a` ends up with both `a` and `b` pointing to the *same* data, so `a.set_x(9)` will mean that `a.get_x()` and `b.get_x()` will both return 9 … it's exactly the same data being changed by `set_x` and being accessed by `get_x` !
  - If you really want to *copy* `a` in order to end up with 2 *distinct* copies, then you'll have to write a `copy` method inside the class to return a new object cloning `a` , then changing one copy will leave the other copy unchanged.
  - Technically, `handle` is a pre-existing class in MATLAB, and writing `pointy < handle` is telling `pointy` to *inherit* everything from `handle` (such as the ability to pass by reference). It can of course have extra abilities all of its own if we define those inside `pointy`.

Linked Lists Revisited

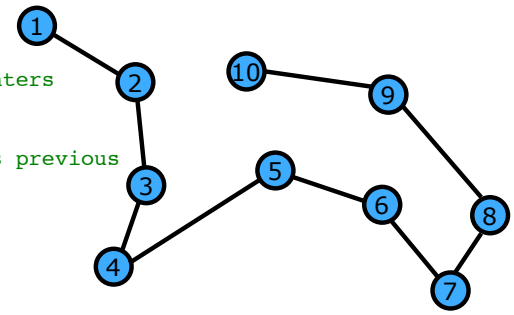
```
classdef node < handle
  % creates nodes for a doubly linked list.
    properties % better to make access private and have gettor and settor methods
        data;
        next;
        previous;
    end

    methods
       function obj = node(d, n, p)
          % class constructor
          if (nargin == 3) % if all input values were given
            obj.data = d;
            obj.next = n;
            obj.previous = p;
          end
          if (nargin == 1) % if only data given
            obj.data = d;
          end
       end % end constructor
    end
end
```



Linked Lists Revisited

```
Classdef linky < handle % creates a doubly linked list.
    properties % better to make access private and have gettor and settor methods
        header; length; % header points to the first real node when there's content
    end
    methods
       function obj = linky() % class constructor
          obj.header = node(); % empty node
          obj.length = 0;
       end % end constructor
       function vide = isempty(obj) % returns true if empty
          vide = (obj.length == 0);
       end % end isempty function
       function preface(obj, data) % insert at front
          temp = node(data) % creates a non-empty node
          temp.next = obj.header.next; % adjust temp's pointers
          temp.previous = obj.header;
          obj.header.next = temp; % adjust header's next
          temp.next.previous = temp; % adjust temp's next's previous
          obj.length = obj.length + 1;
       end % end insertion at front preface method
       function current_list = display(obj)
          current_list = ' ';
          temp = obj.header;
          for counter = 1 : obj.length
                temp = temp.next;
                current_list = [ current_list , num2str(temp.data) , ' → ' ] ;
          end % ending loop through all the list nodes
       end % overwriting the default display method
    end
end
```



Class Inheritance

- So we *inherited* `linky` from the `handle` class … what does this mean?
  - Every property `handle` has, `linky` gets
  - Every method `handle` has, `linky` gets
  - Any values that `handle` might have, `linky` doesn't get! Sorry, you don't get your parent's bank balance!

- We can use inheritance with our own classes, but why and how?
  - The notation is `A < B` for `A` inheriting from `B`
  - If you find yourself writing the same properties and methods for two different classes, consider pooling those that are common to both into a fresh class `C` and then have `A < C` and `B < C`
  - This saves effort and makes maintaining and debugging the code easier

Events in MATLAB

- Events are ways of triggering responses to stimuli
  - Something must be *listening* – event listener
  - Something must be *notifying* – event trigger
  - Triggers notify all listeners, so each listener has to decide if it wants to respond to that trigger
  - We need to list our events formally
  - Pointers are needed, so must inherit from `handle`
  - The `handle` class 'donates' the `addlistener()` and `notify()` methods to the 'child' class(es)
  - We can declare some properties to be 'observable' via `SetObservable = true` and `GetObservable = true`
    - If set, then the following event types notify automatically for those properties (these don't get listed in an event block)
    - `PreSet` and `PreGet` happen just *before* a value is accessed
    - `PostSet` and `PostGet` happen just *after* a value is accessed

Events in MATLAB - example

```
classdef RespondToToggle < handle
   methods
      function obj = RespondToToggle(toggle_obj)
         addlistener(toggle_obj,'ToggledFlag',@RespondToToggle.handleEvnt);
      end
   end
   methods (Static)
      function handleEvnt(src,evtdata)
         if src.flag
            disp('-->: ToggledFlag is true')
         else
            disp('-->: ToggledFlag is false')
         end
      end
   end
end
```

Inherit from handle class

Listens!

event name

'callback', a pointer to the function to be called when the event is triggered

Accessed by class name, not via object

```
classdef Toggle < handle
   properties
      flag = false
   end
   events
      ToggledFlag
   end
   methods
   ...
      function change_flag(obj,newState)
      % Call to set flag value
         if newState ~= obj.flag
            obj.flag = newState;
            notify(obj,'ToggledFlag');
         end
      end
   end
end
```

List the events

Notifies every listening object

Use like this:

```
t = Toggle();
r = RespondToToggle(t); % notice connection to t
t.change_flag(true);
-->: ToggledFlag is true
t.change_flag(true); % no change, so no trigger
t.change_flag(false);
-->: ToggledFlag is false
```


Recursion and Induction

Suppose you are given a 'rule' such as

$$a_n = n\, a_{n-1}$$

and also know that

$$a_1 = 1$$

Then we can use this to build a sequence of numbers

$$1\ ,\ 2\ ,\ 6\ ,\ 24\ ,\ 120\ ,\ 720\ ,\ \ldots$$

which you recognise as factorials. It's easy to see this by building up from the bottom

$$1\ ,\ 2 \times 1\ ,\ 3 \times (\,2 \times 1\,)\ ,\ 4 \times (3 \times 2 \times 1\,)\ ,\ \ldots$$

a_1 $\quad 2\ a_2$ $\quad 3\ a_3$ $\quad\quad 4\ a_4$

although using this approach one-by-one to show that this really only produces factorials would take an infinite amount of time !

Recursion and Induction

We could prove this in an intuitively rigorous inductive way by

① *Remark that* $a_1 = 1 = (1)!$

② *Notice that if we were to assume that*

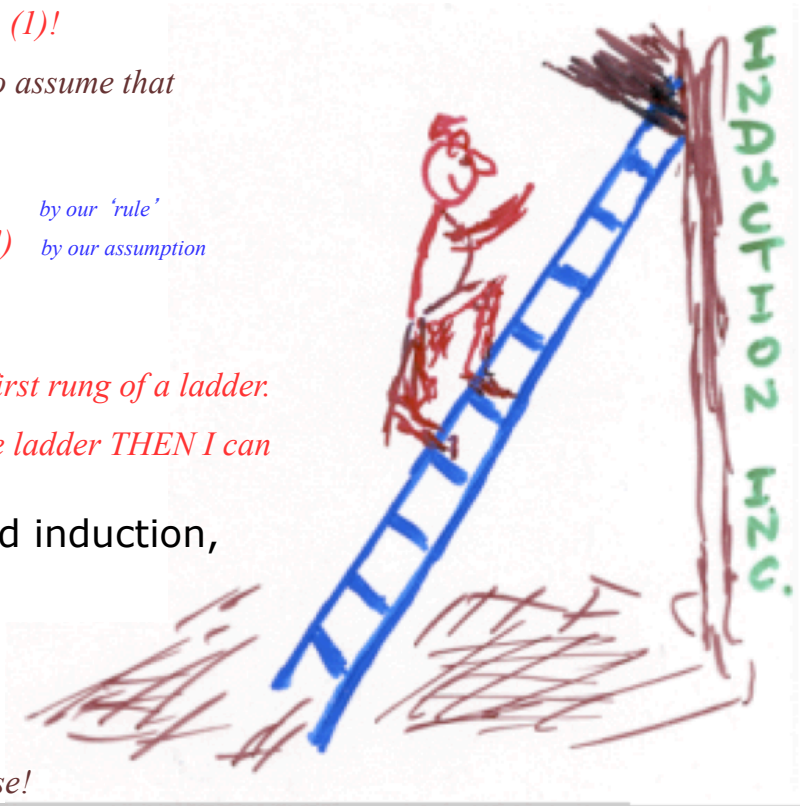
$a_n = n!$

then

$a_{n+1} = (n+1) \times a_n$ *by our 'rule'*

$= (n+1) \times (n!)$ *by our assumption*

$= (n+1)!$



This is rather like saying

① *I can put my foot on the first rung of a ladder.*

② *IF I'm on any rung of the ladder THEN I can step onto the next rung.*

This way of arguing, called induction, is very nice because

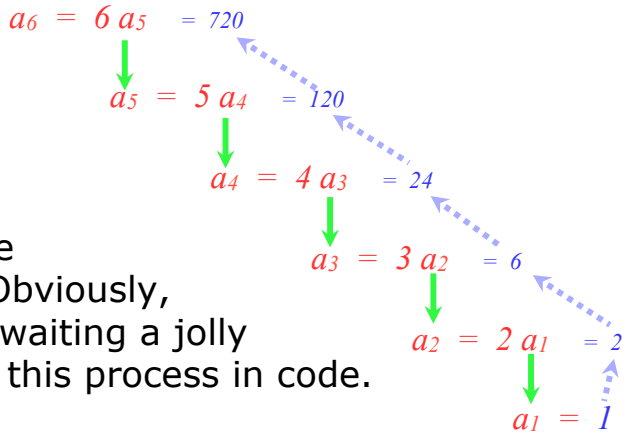
a. *We don't have to do infinitely many steps.*

b. *It's "jolly obvious" that we've covered every case!*

Recursion and Induction

Instead of working from the bottom up, we could work from the top down (provided our 'top' is only 'finitely high') . . .

RECURSION



If you follow the arrows, you can see that this process first finds the BOTTOM, and then assembles the calculation as it returns to the top. Obviously, if there is no bottom then we will be waiting a jolly long time for any results!! Let's see this process in code.

```
function value = facto ( n )
      if ( n == 1 ) value = 1 ;
      else          value = n * facto ( n-1 ) ;
end
```

Then our facto(n) behaves just like our a_n , so calling it for example via facto(6) would invoke a succession of calls to facto, producing the same bottom-hungry routine we saw for a_n until it calls facto(1).

Recursion and Induction

- In fact, any sequence defined by a recurrence relation can be converted very easily into recursive code. Without making any comments about efficiency(!), recursive code is typically very short.
- As experiments, first you should run this code (previous page) to compute facto(10) and then facto(50). After that, try to find the 10th and the 50th term in the following Fibonacci sequence, and then look at the schematic of the recursive calls on the previous page to understand what's going on (and then fix it)!

$$a_n = a_{n-1} + a_{n-2}$$

$$a_1 = 1, \quad \text{and} \quad a_2 = 1$$

Hint: to get information on the time taken to process lines of code, use `tic;` *(to start a matlab stopwatch) then* `t = toc;` *(to acquire the time taken). By making* `t` *into an array, you could even graph this using* `plot(t);`

- It's worth noting that very similar code can be used to compute the binomial coefficients $C_{n,r}$ (for the intuition behind this look at Pascal's triangle).

$$C_{n,r} = C_{n-1,r} + C_{n-1,r-1} \quad \text{with} \quad C_{n,n} = 1, \quad \text{and} \quad C_{n,0} = 1$$

- and powers of a (an example of a 'divide and conquer' approach). Note the vastly faster computation for powers when coded this way!

$$a^n = (a^{n/2})^2 \text{ for } n \text{ even}, \quad a^n = a(a^{n/2})^2 \text{ for } n \text{ odd}, \text{ and } \quad a^0 = 1$$