- Previous lecture:
  - Introduction to objects and classes
- Today's lecture:
  - Defining a class
    - Properties
    - Constructor and other methods
  - Objects are passed by reference to functions
- Announcements:
  - Project 5 due Friday at 11pm
  - Optional review session on Sunday at 1:30pm
  - Prelim 2 on Tues at 7:30pm

Class Interval

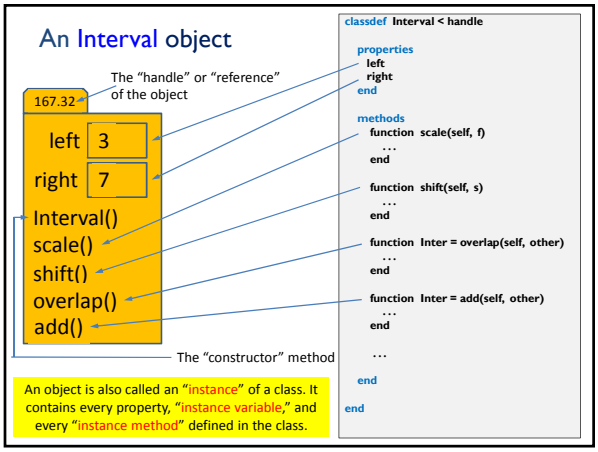
- An interval has two prope
  - left, right
- Actions—methods—of an
  - Scale, i.e., expand
  - Shift
  - Add one interval to anothe
  - Check if one interval is in a
  - Check if one interval overl

To specify the properties and actions of an object is to define its class

```
classdef Interval < handle

  properties
    left
    right
  end

  methods
    function scale(self, f)
      …
    end

    function shift(self, s)
      …
    end

    function Inter = overlap(self, other)
      …
    end

    function Inter = add(self, other)
      …
    end

    …

  end
end
```

These methods (functions) are inside the classdef

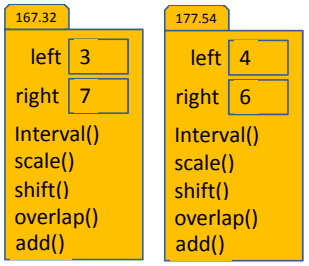
An Interval object



An object is also called an "instance" of a class. It contains every property, "instance variable," and every "instance method" defined in the class.

```
classdef Interval < handle

  properties
    left
    right
  end

  methods
    function scale(self, f)
      …
    end

    function shift(self, s)
      …
    end

    function Inter = overlap(self, other)
      …
    end

    function Inter = add(self, other)
      …
    end

    …

  end
end
```

Multiple Interval objects

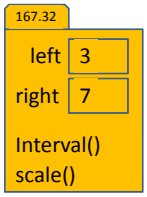


Every object (instance) contains every "instance variable" and every "instance method" defined in the class. Every object has its own handle.

```
classdef Interval < handle

  properties
    left
    right
  end

  methods
    function scale(self, f)
      …
    end

    function shift(self, s)
      …
    end

    function Inter = overlap(self, other)
      …
    end

    function Inter = add(self, other)
      …
    end

    …

  end
end
```

Simplified Interval class

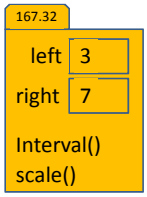
To create an Interval object, use its class name as a function call: p = Interval(3,7)




```
classdef Interval < handle
% An Interval has a left end and a right end

  properties
    left
    right
  end

  methods
    function Inter = Interval(lt, rt)
    % Constructor: construct an Interval obj
      Inter.left= lt;
      Inter.right= rt;
    end

    function scale(self, f)
    % Scale the interval by a factor f
      w= self.right - self.left;
      self.right= self.left + w*f;
    end
  end
end
```

The *constructor* method

To create an Interval object, use its class name as a function call: p = Interval(3,7)




```
classdef Interval < handle
% An Interval has a left end and a right end

  properties
    left
    right
  end

  methods
    function Inter = Interval(lt, rt)
    % Constructor: construct an Interval obj
      Inter.left= lt;
      Inter.right= rt;
    end
```

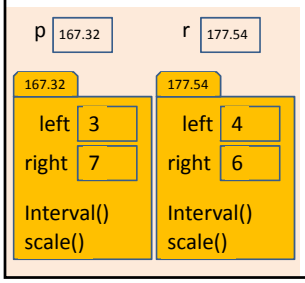
The *constructor* is a special method whose main jobs are to

- compute the handle of the new object,
- execute the function code (to assign values to properties) at that handle,
- return the handle of the object.

Constructor has the name of the class.

A handle object is referenced by its handle

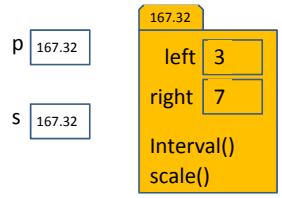
```
p = Interval(3,7);
r = Interval(4,6);
```



A handle, also called a reference, is like an address; it indicates the memory location where the object is stored.

What is the effect of referencing?

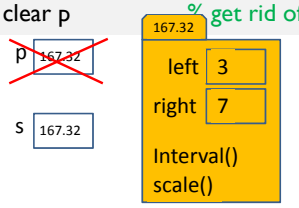
```
p = Interval(3,7);   % p references an Interval object
s = p;               % s stores the same reference as p
s.left = 2;          % change value inside object
disp(p.left)         % 2 is displayed
```



The object is not copied—no new object is created! s and p both reference the same object.

What is the effect of referencing?

```
p = Interval(3,7);   % p references an Interval object
s = p;               % s stores the same reference as p
s.left = 2;          % change value inside object
disp(p.left)         % 2 is displayed
clear p              % get rid of p from memory
```



The object still can be accessed through s.

In contrast, structs are stored by value …

```
P.x=5; P.y=0;  % A point struct P
Q=P;           % Q gets a copy of P--Q is ANOTHER
               %   struct with same field values
Q.y=9;         % Changes Q's copy only, not P's
disp(P.y)      % What is displayed?
```

In fact, storing-by-value is true of all non-handle-object variables. You already know this from before …

```
a=5;
b=a+1;   % b stores the value 6, not
         %   some "definition" a+1
a=8;     % Changing a does not change b
disp(b)  % 6 is displayed
```

Syntax for calling an instance method

```
r = Interval(4,6);
r.scale(5)
```

Method name

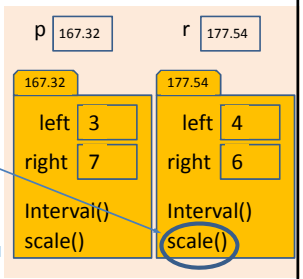
Reference of the object whose method is to be dispatched

Argument for the second parameter specified in function header (f). Argument for first parameter (self) is absent because it is the same as r, the owner of the method

```
classdef Interval < handle
% An Interval has a left end and a right end

  properties
    left
    right
  end

  methods
    function Inter = Interval(lt, rt)
    % Constructor: construct an Interval obj
      Inter.left= lt;
      Inter.right= rt;
    end

    function scale(self, f)
    % Scale the interval by a factor f
      w= self.right - self.left;
      self.right= self.left + w*f;
    end
  end
end
```

Calling an object's method (instance method)

```
p = Interval(3,7);
r = Interval(4,6);
r.scale(5)
```



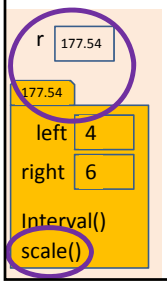
The owner of the method to be dispatched

Syntax:

<reference>.<method>(<arguments for 2nd thru last parameters>)

Executing an instance method

```
r = Interval(4,6);
r.scale(5)
disp(r.right) %What will it be?
```

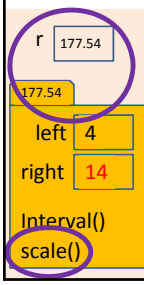


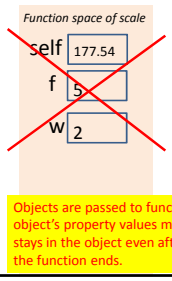
1st parameter (self) automatically references itself, i.e., its own handle

```
classdef Interval < handle
% An Interval has a left end and a rig
  properties
    left
    right
  end
  methods
    function scale(self, f)
    % Scale the interval by a factor f
      w= self.right - self.left;
      self.right= self.left + w*f;
    end
  end
end
```

Object is passed to a function by reference

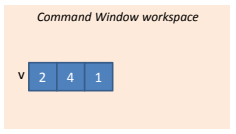
```
r = Interval(4,6);
r.scale(5)
disp(r.right) % updated value
```

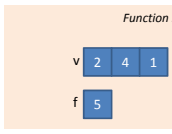





```
classdef Interval < handle
% An Interval has a left end and a rig
  properties
    left
    right
  end
  methods
    function Inter = Interval(lt, rt)
    % Constructor: construct an Inte
      Inter.left= lt;
      Inter.right= rt;
    end
    function scale(self, f)
    % Scale the interval by a factor f
      w= self.right - self.left;
      self.right= self.left + w*f;
    end
  end
end
```

Objects are passed to functions by reference. Changes to an object's property values made through the local reference (self) stays in the object even after the local reference is deleted when the function ends.






```
v= [2 4 1];
scale2(v,5)
disp(v) %???
```

```
function scale2(v,f)
% Scale v by a factor f
v= v*f;
```

Non-objects are passed to a function by value

Objects are passed to a function by reference

```
r = Interval(4,6);
r.scale(5)
disp(r.right) % updated value
```

```
classdef Interval < handle
  :
  methods
    :
    function scale(self, f)
    % Scale the interval by a factor f
      w= self.right - self.left;
      self.right= self.left + w*f;
    end
  end
end
```

```
v= [2 4 1];
scale2(v,5)
disp(v) %NO CHANGE
```

```
function scale2(v,f)
% Scale v by a factor f
v= v*f;
```

Non-objects are passed to a function by value

Syntax for calling an instance method:

<reference>.<method>(<arguments for 2nd thru last parameters>)

```
p = Interval(3,7);
r = Interval(4,6);
```

Better!

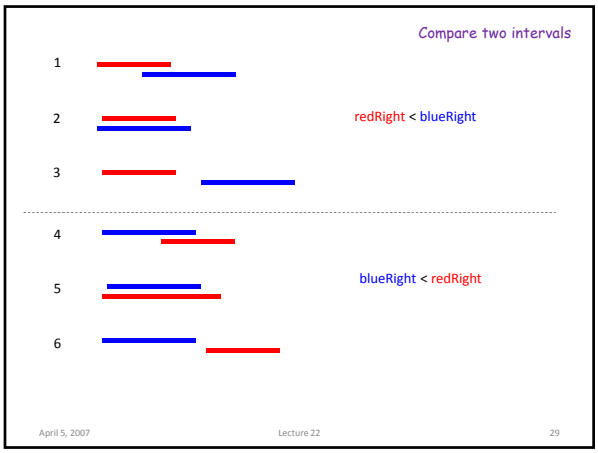
```
yesno= p.isIn(r);
% Explicitly call
% p's isIn method
```

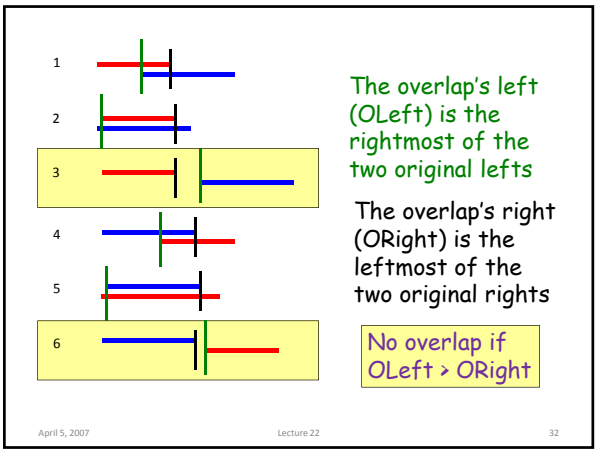
```
yesno= isIn(p,r);
% Matlab chooses the
% isIn method of one
% of the parameters.
```

```
classdef Interval < handle
  :
  methods
    :
    function scale(self, f)
    % Scale self by a factor f
      w= self.right - self.left;
      self.right= self.left + w*f;
    end

    function tf = isIn(self, other)
    % tf is true if self is in other interval
      tf= self.left>=other.left && ...
          self.right<=other.right;
    end


  end
end
```

Method to find overlap between two Intervals

```
function Inter = overlap(self, other)
% Inter is overlapped Interval between self
% and the other Interval. If no overlap then
% self is empty Interval.
```






```
function Inter = overlap(self, other)
% Inter is overlapped Interval between self
% and the other Interval. If no overlap then
% self is empty Interval.

  Inter= Interval.empty();
  left= max(self.left, other.left);
  right= min(self.right, other.right);
  if right-left > 0
      Inter= Interval(left, right);
  end
end
```

Built-in function to create an empty array of the specified class

Built-in function isempty

```
% Example use of overlap function
A= Interval(3,7);
B= Interval(4,4+rand*5);
X= A.overlap(B);
if ~isempty(X)
    fprintf('(%f,%f)\n', X.left,X.right)
end
```

April 5, 2007

classdef syntax summary

A class file has the name of the class and begins with keyword classdef:

Properties

```
classdef Interval < handle
% An Interval has a left end and a right end
  properties
    left
    right
  end
```

```
classdef classname < handle
```

The class specifies handle objects

Constructor returns a reference to the class object

Constructor

```
  methods
    function Inter = Interval(lt, rt)
    % Constructor: construct an Interval ob
      Inter.left= lt;
      Inter.right= rt;
    end
```

Each instance method's first parameter must be a reference to the instance (object) itself

Instance methods (functions)

```
    function scale(self, f)
    % Scale the interval by a factor f
      w= self.right - self.left;
      self.right= self.left + w*f;
    end
  end
end
```

Use keyword end for classdef, properties, methods, function.

This file's name is Interval.m