• Previous lecture:
  – Array of objects
  – Methods that handle variable numbers of arguments

• Today’s lecture:
  – Why use OOP?
  – Attributes for properties and methods
  – Inheritance: extending a superclass
  – Overriding methods in superclass

• Announcement:
  – Prelim 2 will be returned Tues
Observations about our class **Interval**

• We can use it (create **Interval** objects) anywhere
  – Within the **Interval** class, e.g., in method **overlap**
  – “on the fly” in the Command Window
  – In other function/script files – not class definition files
  – In another class definition

• Designing a class well means that it can be used in many different applications and situations
OOP ideas

• Aggregate variables/methods into an abstraction (a class) that makes their relationship to one another explicit
• Objects (instances of a class) are self-governing (protect and manage themselves)
• Hide details from client, and restrict client’s use of the services
• Provide clients with the services they need so that they can create/manipulate as many objects as they need
Restricting access to properties and methods

- **Hide data** from “outside parties” who do not need to access that data—need-to-know basis
- E.g., we decide that users of Interval class cannot directly change \texttt{left} and \texttt{right} once the object has been created. Force users to use the provided methods—constructor, scale, shift, etc.—to cause changes in the object data
- **Protect data** from unanticipated user action
- **Information hiding is very important in large projects**
Constructor can be written to do error checking

classdef Interval < handle
    properties
        left
        right
    end

    methods
        function Inter = Interval(lt, rt)
            if nargin==2

                Inter.left= lt;
                Inter.right= rt;

            end
        end
    end
end
Constructor can be written to do error checking!

```matlab
classdef Interval < handle
    properties
        left
        right
    end
    methods
        function Inter = Interval(lt, rt)
            if nargin==2
                if lt <= rt
                    Inter.left = lt;
                    Inter.right = rt;
                else
                    disp('Error at instantiation: left>right')
                end
            end
        end
    end
end
```

Should force users (clients) to use code provided in the class to create an Interval or to change its property values once the Interval has been created.

E.g., if users cannot directly set the properties `left` and `right`, then they cannot accidentally “mess up” an Interval.

Alternative: use built-in function `error` to halt program execution, e.g.,
```matlab
error('Error at instantiation: left>right')
```
Data that the client does not need to access should be protected: \textcolor{red}{\textit{private}}

Provide a set of methods for \textcolor{green}{\textit{public}} access.

\textbf{The “client-server model”}
classdef Interval < handle

    properties
        left
        right
    end

    methods
        function scale(self, f)
            ...
            end

        function Inter = overlap(self, other)
            ...
            end

        function Inter = add(self, other)
            ...
            end

        ...
    end
end

% Interval experiments
for k=1:5
    fprintf('Trial %d\n', k)
a= Interval(3, 3+rand*5);
b= Interval(6, 6+rand*3);
disp(a)
disp(b)
c= a.overlap(b);
if ~isempty(c)
    fprintf('Overlap is ')
disp(c)
else
    disp('No overlap')
end
pause
end
Attributes for properties and methods

- **public**
  - Client has access
  - Default

- **private**
  - Client cannot access

```matlab
% Client code
r = Interval(4, 6);
r.scale(5); % OK
r = Interval(4, 14); % OK
r.right = 14; % error
disp(r.right) % error
```

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

properties (SetAccess=private, GetAccess=private)
  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
```

Within the class, there is always access to the properties, even if private.
Attributes for properties and methods

- **public**
  - Client has access
  - Default
- **private**
  - Client cannot access

```plaintext
% Client code
r = Interval(4,6);
r.scale(5); % OK
r = Interval(4,14); % OK
r.right=14; % error
disp(r.right) % error
```

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

properties (Access=private)
  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
```

Both `getAccess` and `setAccess` are private.
Public "getter" method

- Provides client the ability to get a property value

Code:

```matlab
classdef Interval < handle
  % An Interval has a left end and a right end
  properties (Access=private)
    left
    right
  end
  methods
    function Inter = Interval(lt, rt)
      Inter.left= lt;
      Inter.right= rt;
    end
    function lt = getLeft(self)
      % lt is the interval’s left end
      lt= self.left;
    end
    function rt = getRight(self)
      % rt is the interval’s right end
      rt= self.right;
    end
  end
end

% Client code
r= Interval(4,6);
disp(r.left) % error
disp(r.getLeft()) % OK
```
Public “setter” method

• Provides client the ability to set a property value
• Don’t do it unless really necessary! If you implement public setters, include error checking (not shown here).

% Client code
r= Interval(4,6);
r.right= 9; % error
r.setRight(9) % OK
Always use available methods, even when within same class

```matlab
classdef Interval < handle
    properties (Access=private)
        left; right
    end
methods
    function Inter = Interval(lt, rt)
        ...
    end
    function lt = getLeft(self)
        lt = self.left;
    end
    function rt = getRight(self)
        rt = self.right;
    end
    function w = getWidth(self)
        w= self.getRight() – self.getLeft() ;
    end
end
```

% Client code

```matlab
A = Interval(4,7);
disp(A.getRight() )
```

% ... lots of client code that uses
% class Interval, always using the
% provided public getters and
% other public methods ...

In here... code that always uses the getters & setters
Always use available methods, even when within same class

```matlab
classdef Interval < handle
    properties (Access=private)
        left; right
    end
methods
    function Inter = Interval(lt, rt)
        ...
    end
    function lt = getLeft(self)
        lt = self.left;
    end
    function rt = getRight(self)
        rt = self.right;
    end
    function w = getWidth(self)
        w = self.getRight() – self.getLeft();
    end
    ...
end
end
```
OOP ideas → Great for managing large projects

• Aggregate variables/methods into an abstraction (a class) that makes their relationship to one another explicit

• Objects (instances of a class) are self-governing (protect and manage themselves)

• Hide details from client, and restrict client’s use of the services

• Provide clients with the services they need so that they can create/manipulate as many objects as they need

Rewrite the getters/setters. Everything else stays the same! Cool! Happy clients!
A fair die is...

```matlab
classdef Die < handle
    properties (Access=private)
        sides=6;
        top
    end
    methods
        function D = Die(...) ...
        function roll(...) ...
        function disp(...) ...
        function s = getSides(...) ...
        function t = getTop(...) ...
    end
    methods (Access=private)
        function setTop(...) ...
    end
end
```

What about a trick die?
Separate classes—each has its own members

classdef Die < handle
    properties (Access=private)
    sides=6;
    top
end
methods
    function D = Die(...)  ...
    function roll(...)  ...
    function disp(...)  ...
    function s = getSides(...)  ...
    function t = getTop(...)  ...
end
methods (Access=private)
    function setTop(...)  ...
end
end
classdef TrickDie < handle
    properties (Access=private)
    sides=6;
    top
    favoredFace
    weight=1;
end
methods
    function D = TrickDie(...)  ...
    function roll(...)  ...
    function disp(...)  ...
    function s = getSides(...)  ...
    function t = getTop(...)  ...
    function f = getFavoredFace(...)  ...
    function w = getWeight(...)  ...
end
methods (Access=private)
    function setTop(...)
end
end
Separate classes—each has its own members

classdef Die < handle  
    properties (Access=private)  
        sides=6;  
        top  
    end  
    methods  
        function D = Die(...)  
        function roll(...)  
        function disp(...)  
        function s = getSides(...)  
        function t = getTop(...)  
    end  
    methods (Access=private)  
        function setTop(...)  
    end  
end  

classdef TrickDie < handle  
    properties (Access=private)  
        sides=6;  
        top  
        favoredFace  
        weight=1;  
    end  
    methods  
        function D = TrickDie(...)  
        function roll(...)  
        function disp(...)  
        function s = getSides(...)  
        function t = getTop(...)  
        function f = getFavoredFace(...)  
        function w = getWeight(...)  
    end  
    methods (Access=private)  
        function setTop(...)  
    end  
end
Can we get all the functionality of **Die** in **TrickDie** without re-writing all the **Die** components in class **TrickDie**?

```matlab
classdef Die < handle
    properties (Access=private)
        sides=6;
        top
    end
    methods
        function D = Die(...) ...
        function roll(...) ...
        function disp(...) ...
        function s = getSides(...) ...
        function t = getTop(...) ...
    end
    methods (Access=private)
        function setTop(...) ...
    end
end
```

```matlab
classdef TrickDie < handle
    properties (Access=private)
        favoredFace
        weight=1;
    end
    methods
        function D = TrickDie(...) ...
        function f = getFavoredFace(...) ...
        function w = getWeight(...) ...
    end
end
```

"Inherit" the components of class Die
### Yes! Make TrickDie a **subclass** of Die

```matlab
classdef Die < handle
    properties (Access=private)
        sides=6;
        top
    end
    methods
        function D = Die(...) 
        function roll(...) 
        function disp(...) 
        function s = getSides(...) 
        function t = getTop(...) 
    end
    methods (Access=protected)
        function setTop(...) 
    end
end

classdef TrickDie < Die

    properties (Access=private)
        favoredFace
        weight=1;
    end
    methods
        function D = TrickDie(...) 
        function f=getFavoredFace(...) 
        function w = getWeight(...) 
    end
end
```