• Previous lecture:
  – Array of objects
  – Methods that handle a variable number of arguments
  – Using a class in another

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

• Announcement:
  – Exercise 12 Q4 (from dis) due on Sunday at 11pm
  – Project 6 (Parts A & B) due May 10th (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

• Object properties (data) need not be passed to instance methods—only the object handle (reference) is passed. Important for large data sets!
Pass reference, not properties

When an instance method executes, the properties—data—are accessible through the handle (reference). No local copy of the data is needed in the method’s memory space.
OOP ideas

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

• Object properties (data) need not be passed to instance methods—only the object handle (reference) is passed. Important for large data sets!

• 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 *left* and *right* once the object has been created. **Force users to use the provided methods**—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')
```
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
', 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

Server

Example client code
Data that the client does not need to access should be protected: \textcolor{red}{private}
Provide a set of methods for \textcolor{red}{public} access.

The “client-server model”
Attributes for properties and methods

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

- **private**
  - Client cannot access

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

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
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

```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 scale(self, f)
    w= self.right - self.left;
    self.right= self.left + w*f;
  end
end

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

Both GetAccess and SetAccess are private.
Public “getter” method

- Provides client the ability to get a property value

% 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).

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

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

% Client code
...
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 ...
Always use available methods, even when within same class

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

New Interval implementation

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

In here... code that always uses the getters & setters

Rewrite the getters/setters. Everything else stays the same! Cool! Happy clients!
OOP ideas → Great for managing large projects

- Aggregate variables/methods into an abstraction (a class) that makes their relationship to one another explicit
- Object properties (data) need not be passed to instance methods—only the object handle (reference) is passed. Important for large data sets!
- 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…

```
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

```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

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

```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

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
    “Inherit” the components of class Die
    properties (Access=private)
        favoredFace
        weight=1;
    end
    methods
        function D = TrickDie(...) ...
        function f = getFavoredFace(...) ...
        function w = getWeight(...) ...
    end
end
```
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
```

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

Inheritance relationships are shown in a class diagram, with the arrow pointing to the parent class.

An is-a relationship: the child is a more specific version of the parent. Eg., a trick die is a die.

Multiple inheritance: can have multiple parents e.g., Matlab
Single inheritance: can have one parent only e.g., Java
Inheritance

• Allows programmer to *derive* a class from an existing one

• Existing class is called the *parent class*, or *superclass*

• Derived class is called the *child class* or *subclass*

• The child class *inherits* the (public and protected) members defined for the parent class

• Inherited trait can be *accessed as though it was locally defined*
Which components get “inherited”? 

- **public** components get inherited
- **private** components exist in object of child class, but cannot be directly accessed in child class ⇒ we say they are **not** inherited
- Note the difference between inheritance and existence!
Which components get “inherited”?  

• **public** components get inherited  
• **private** components **exist** in object of child class, but cannot be **directly accessed** in child class ⇒ we say they are **not** inherited  
• Note the difference between inheritance and existence!
**protected attribute**

- Attributes dictate which members get inherited

- **private**
  - Not inherited, can be *accessed* by local class only

- **public**
  - Inherited, can be *accessed* by all classes

- **protected**
  - Inherited, can be *accessed* by subclasses

- **Access**: access as though defined locally

- **All** members from a superclass *exist* in the subclass, but the **private** ones cannot be *accessed* directly—can be accessed through inherited (public or protected) methods
Let’s play with dice—**Dies** and **TrickDies**

% In Command Window—not class Die or TrickDie

d = Die(6) % disp method of Die used
disp(d.top) % Error; top is private to class Die
d.getTop()

t = TrickDie(2,10,6) % disp method of TrickDie used
disp(t.top) % Error; top is private to class Die
t.getTop() % getTop not defined in TrickDie class but
    % is inherited

d.setTop(5) % Error; setTop is protected so available

t.setTop(5) % only to class Die and its subclasses