• **Previous lecture:**
  – Why use OOP?
  – *public* and *private* attributes for properties and methods

• **Today’s lecture:**
  – More on attributes, getters, and setters
  – Inheritance: extending a superclass
  – Overriding methods in superclass

• **Announcement:**
  – Project 6 due on Dec 1\(^{st}\) (Thurs) at 11pm.
  – Remember *academic integrity!* We will check all submissions using MOSS.
  – **Final exam** on Wednesday, Dec 7\(^{th}\), at 2pm for both Lec 1 and Lec 2. Email Randy Hess (rbh27) now if you have an exam conflict. **Specify your entire exam schedule** (course numbers/contacts and the exam times). We must have this information by the end of this week.
Data that the client does not need to access should be protected: \texttt{private}

Provide a set of methods for \texttt{public} access.

The “client-server model”
Public “getter” method

- Provides client the ability to get a property value

```matlab
% Client code
r = Interval(4,6);
disp(r.left)  % error
disp(r.getLeft()) % OK
```

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
```
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

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
```

New Interval implementation

```
classdef Interval < handle
    properties (Access=private)
        left; width
    end
methods
    function Inter = Interval(lt, w)
        ...
    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
```

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

```
... 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
```

```matlab
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

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

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!

![A Die](image_url)

Variable and method details:
- sides: 6
- top: 2
- setTop()
- Die()
- getSides()
- roll()
- getTop()
- disp()
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()

\[ t = \text{TrickDie}(2,10,6) \] % 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
Constructor: must call the superclass’ constructor

• In a subclass’ constructor, call the superclass’ constructor before assigning values to the subclass’ properties.

• Calling the superclass’ constructor cannot be conditional: explicitly make one call to superclass’ constructor

See constructor in TrickDie.m
Overriding methods

• Subclass can *override* definition of inherited method
• New method in subclass has the same name (but has different method body)

See method *roll* in *TrickDie.m*
Overriding methods

• Subclass can override definition of inherited method
• New method in subclass has the same name (but has different method body)
• Which method gets used??

The object that is used to invoke a method determines which version is used

• Since a TrickDie object is calling method roll, the TrickDie’s version of roll is executed
• In other words, the method most specific to the type (class) of the object is used
Accessing superclass’ version of a method

• Subclass can override superclass’ methods
• Subclass can access superclass’ version of the method

Syntax

classdef Child < Parent
properties
  propC
end
methods
...
  function x= \texttt{method}(arg)
    \texttt{y} = \texttt{method@Parent}(arg);
  end
  x = ... y ... ;
end
...
end

See method \texttt{disp} in \texttt{TrickDie.m}
Important ideas in inheritance

• Keep common features as high in the hierarchy as reasonably possible

• Use the superclass’ features as much as possible

• “Inherited” ⇒ “can be accessed as though declared locally”
  (private member in superclass exists in subclasses; they just cannot be accessed directly)

• Inherited features are continually passed down the line