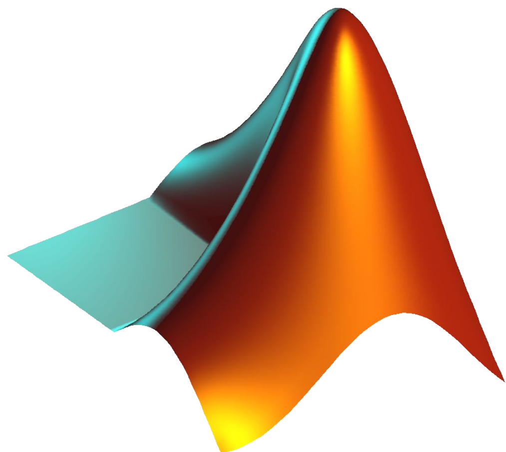
CS 1112 Introduction to Computing Using MATLAB



Instructor: Dominic Diaz

Website:
https://www.cs.cornell.edu/courses/cs1112/2022fa/

Today: object-oriented programming

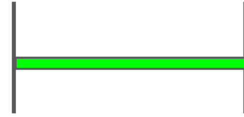
Agenda and announcements

- Last time
  - Object-oriented programming
- Today
  - Object-oriented programming
    - Why use OOP?
    - Private and public properties and methods
    - Intro to inheritance
- Announcements
  - Ex 12 due Tuesday Nov 15th (tonight)
  - Project 5 late deadline until Wednesday at 11 PM (deduction is 5% for late submission)
  - Project 6 released Thursday 11/17 (due Dec 1st)
  - Prelim 2 grades released tonight!
    - Submit regrade requests by 11/22

Observations about our class Intervals

- We can create an interval object anywhere (this applied to any class!)
  - Within the Interval class itself (for example, in method overlap)
  - In the Command Window
  - In other function/script files
  - In another class definition (for example, in class LocalWeather)

```
I = Interval(2,5);
```



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

Why is OOP useful?

- Aggregate variables and methods into a single construct (a class)
- Object properties do not need to be passed to instance methods or other functions—only the object handle is passed. Useful for large data sets!
- Objects can protect and manage themselves
    - Hide details from clients, but expose services that clients need
    - Don’t allow clients to invalidate data (which then breaks the provided services)
        - For example, we decide that users of the 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.
    - Information hiding is very important in large projects.

Preserving sensible relationships between properties

Set default values for properties. If I call `I = Interval()` then `I.right = 0` and `I.left = 0`.

If nargin ~= 2, will set left and right to default values

If nargin == 2 but the right input is less than the left input, throw an error (stop the code).

```
classdef Interval < handle
  properties
    left = 0;
    right = 0;
  end

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

Privatizing information

The below code could create an interval with left = 2 and right = 0.

We can prevent this from happening by forcing clients to use code provided in the class to change property values once the Interval has been created.

If users cannot directly set properties left and right, they cannot accidentally mess up an Interval.

```
I = interval(2,5);
I.right = 0;
```

```
classdef Interval < handle
  properties
    left = 0;
    right = 0;
  end

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

Privatizing information

Access can be set to:

- Public
  - Client has access
  - Default
- Private
  - Client cannot directly access
- Protected
  - Client cannot directly access

```
% Code outside classdef
r = Interval(0,1);
r.scale(5);
r = Interval(4,14);
r.right = 15;  % error
disp(r.right)  % error
```

```
classdef Interval < handle
  properties (Access=private)
    left = 0;
    right = 0;
  end

  methods
    function Inter = Interval(lt,  rt)
      if nargin==2
        if lt <= rt
          Inter.left = lt;
          Inter.right = rt;
        else
          error('Error at instantiation: left>right')
        end
      end
    end
    function scale(self,f)
      w = self.right - self.left;
      self.right = self.left + w*f;
    end
  end
end
```

Privatizing information

```
% Code outside classdef
r = Interval(0,1);
r.scale(5);
r = Interval(4,14);
r.right = 15;  % error
disp(r.right)  % error
```

Setting the `access` to be `private` makes it so that you cannot access or set that value directly outside of the `classdef`. You can still access it or set the value inside the `classdef`.

```
classdef Interval < handle
  properties (Access=private)
    left = 0;
    right = 0;
  end

  methods
    function Inter = Interval(lt,  rt)
      if nargin==2
        if lt <= rt
          Inter.left = lt;
          Inter.right = rt;
        else
          error('Error at instantiation: left>right')
        end
      end
    end
    function scale(self,f)
      w = self.right - self.left;
      self.right = self.left + w*f;
    end
  end
end
```

Can still access `self.left` and `self.right` inside `methods` (regardless of access)!

Public “getter method”

- Provides way to get a property value outside of the `classdef`

```
r = Interval(4,6);
disp(r.left)        % error
disp(r.getLeft())   % works!
```

```
classdef Interval < handle
  properties (Access=private)
    left = 0;
    right = 0;
  end

  methods
    function Inter = Interval(lt,  rt)
      if nargin==2
        if lt <= rt
          Inter.left = lt;
          Inter.right = rt;
        else
          error('Error at instantiation: left>right')
        end
      end
    end
    function lt = getLeft(self)
      lt = self.left;
    end
    function setLeft(self, lt)
      self.left = lt;
    end
  end
end
```

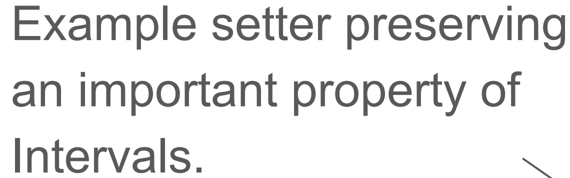
Public “setter method”

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

```
r = Interval(4,6);
r.left = 2;          % error
r.setLeft(2)         % works!
```

```
classdef Interval < handle
  properties (Access=private)
    left = 0;
    right = 0;
  end

  methods
    function Inter = Interval(lt,  rt)
      if nargin==2
        if lt <= rt
          Inter.left = lt;
          Inter.right = rt;
        else
          error('Error at instantiation: left>right')
        end
      end
    end
    function lt = getLeft(self)
      lt = self.left;
    end
    function setLeft(self, lt)
      self.left = lt;
    end
  end
end
```

Setter with error checking



Example setter preserving an important property of Intervals.

```
classdef Interval < handle
  properties (Access=private)
    left = 0;
    right = 0;
  end

  methods
    function Inter = Interval(lt,  rt)
      ...
    end
    function setLeft(self, lt)
      rt = self.right;
      if lt > self.right:
        tmp = lt;
        lt = rt;
        rt = tmp;
      end
      self.left = lt;
      self.right = rt;
    end
  end
end
```

Getters and setters: What have we achieved?

- Getters let a user accesses a property without the risk of incorrectly changing those properties.
- Setters (or a lack of setters) let us control how property values can change.
  - Should have error checking on attempts to write to properties

Poll everywhere

- 1 works because you can always set `properties` inside the `classdef`
- 2 works because you can always access `properties` inside `class methods`
- 3 works because `area` is a `public method`
- 4 does not work because I cannot directly access `shape.s` outside of the class definition

```
classdef Square < handle
   properties (Access=private)
      s = 1  % side length
   end
   methods (Access=public)
      function obj = Square(side)
         if nargin == 1
      (1)   obj.s = side;
         end
      end
      function a = area(self)
   (2)   a = self.s*self.s;
      end
   end
end
```

```
shape = Square(2);
(3) a1= shape.area();
(4) a2= shape.s*shape.s;

shape.s= 1;
```

You can have different access types for properties and methods

Default Access is public

somePrivateFunction is private… so can we access it ever?

Yes! In public methods!

```
classdef Interval < handle
  properties
    left = 0;
    right = 0;
  end

  properties (Access=private)
    width = 0;
  end

  methods
    function Inter = Interval(lt, rt)
      ...
    end
    ...
  end

  methods (Access=private)
    function somePrivateFunction(a, b)
      ...
    end
  end
end
```

Let’s consider a new fair die class

```
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?

Closely related trick die class

```
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 = getWeight(...) ...
    function f = getFavFace(...) ...
  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?

```
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 = getWeight(...) ...
    function f = getFavFace(...) ...
  end

end
```

Can we get all the functionality of Die in TrickDie without re-writing all the Die components in class TrickDie? YES!

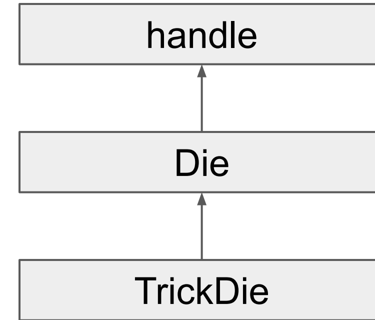
```
classdef Die < handle
  properties (Access=private)
    sides=6;
    top
  end

  methods
    function D = Die(...)  ...
    function roll(...)  ...
    function disp(...)  ...
    function s = getSides(...)  ...
    function t = getTop(...)  ...
  end

  methods (Access=private)
    function setTop(...)  ...
  end
end
```

```
classdef TrickDie < Die

  properties (Access=private)
    favoredFace
    weight=1;
  end

  methods
    function D = TrickDie(...) ...
    function f = getWeight(...) ...
    function f = getFavFace(...) ...
  end

end
```

< [className] allows us to inherit properties and methods from another class!

Inheritance

- Classes can inherit methods and properties from other classes
  - Main advantage: don't need to rewrite long code in multiple classes.
- Inheritance relationships can be shown in a class diagram, with arrows pointing from a child class to a parent class.



- The child is a more specific version of the parent. For example, a trick die is a die.

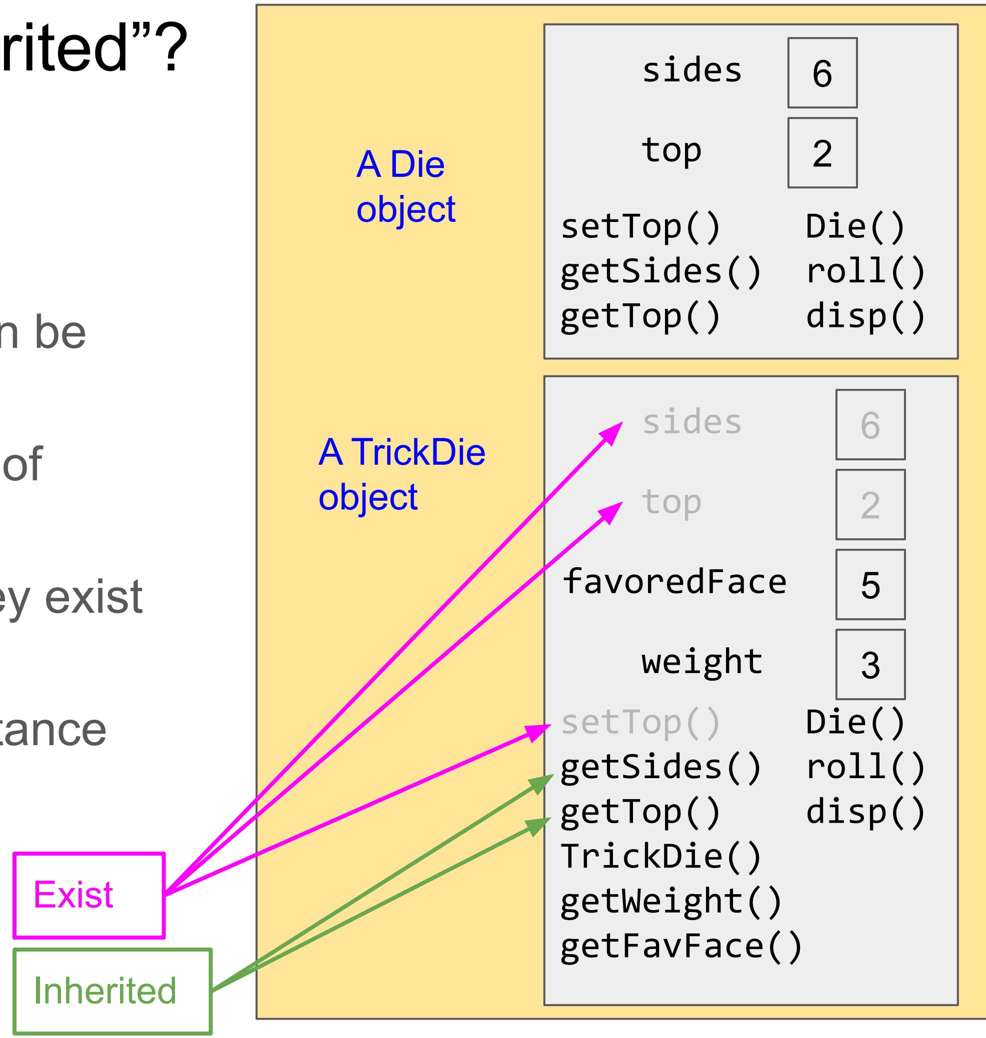
Multiple inheritance: child classes can have multiple parents → for example, MATLAB
Single inheritance: child classes can only have one parent → for example, Java

Inheritance vocabulary

- Allows programmer to derive a class from an existing one (Allows programmer to extend a class)
- 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

Which components get “inherited”?

- Public components get inherited
  - Properties and methods of the parent class that are public can be accessed (get) and set.
- Private components exist in object of child class, but cannot be directly accessed in child class (we say they exist but are not inherited)
- Note the difference between inheritance and existence!



```
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 < Die
  properties (Access=private)
    favoredFace; weight=1;
  end
  methods
    function D = TrickDie(...) ...
    function f = getWeight(...) ...
    function f = getFavFace(...) ...
  end
end
```

A Die object

sides	6
top	2

setTop() Die()
getSides() roll()
getTop() disp()

A TrickDie object

sides	6
top	2
favoredFace	5
weight	3

setTop() Die()
getSides() roll()
getTop() disp()
TrickDie()
getWeight()
getFavFace()