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!
- 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

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.

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!

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.
Attributes for properties and methods

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

% Example client code

classdef Interval < handle
    properties
        left
        right
    end
    methods
        function Inter = Interval(lt, rt)
            % Constructor: construct an Interval object
            left = lt;
            right = rt;
        end
        function scale(self, f)
            % Scale the interval by a factor f
            self.left = self.left * f;
            self.right = self.right * f;
        end
        function Inter = overlap(self, other)
            % Overlap
            disp('Overlap is');
        end
    end
end

Client code

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

Server code

```matlab
% Server

r.setLeft(9) % OK
r.right = 9; % error
r= Interval(4, 14);
r.setRight(9) % OK
```

Public "getter" method

- Provides client the ability to *get* a property value

% Client code

```matlab
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

```matlab
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)
            % Here code that always uses the getters & setters
        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
end
```

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

Always use available methods, even when within same class

```matlab
classdef Interval < handle
    properties (Access=private)
        left; right; width
    end
    methods
        function Inter = Interval(lt, rt)
            % Here code that always uses the getters & setters
        end
        function lt = getLeft(self)
            lt = self.left;
        end
        function rt = getRight(self)
            rt = self.getRight();
        end
        function w = getWidth(self)
            w = self.width;
        end
    end
end
end
```

OOP ideas \rightarrow 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

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=protected)
        function setTop(…)
    end
end
```

```matlab
classdef TrickDie < Die
    properties (Access=private)
        sides=6;
        top
        favoredFace
        weight=1;
    end
    methods
        function D = TrickDie(…)  …
        function f = getFavoredFace(…)
            f = self.favoredFace;
        end
        function w = getWeight(…)
            w = self.weight;
        end
    end
end
```

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=protected)
        function setTop(…)
    end
end
```

What about a trick die?

```matlab
classdef TrickDie < Die
    properties (Access=private)
        sides=6;
        top
        favoredFace
        weight=1;
    end
    methods
        function D = TrickDie(…)  …
        function f = getFavoredFace(…)
            f = self.favoredFace;
        end
        function w = getWeight(…)
            w = self.weight;
        end
    end
end
```

Yes! Make TrickDie a subclass of Die

```matlab
classdef TrickDie < Die
    properties (Access=private)
        sides=6;
        top
        favoredFace
        weight=1;
    end
    methods
        function D = TrickDie(…)  …
        function f = getFavoredFace(…)
            f = self.favoredFace;
        end
        function w = getWeight(…)
            w = self.weight;
        end
    end
end
```
Inheritance

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

```
  handle
  Die
  TrickDie
```

An is-a relationship: the child is a more specific version of the parent. E.g., 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

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!
- Let’s create a TrickDie and play with it …

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

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

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

```
classdef Child < Parent
  properties
    propC
  end
  methods
    function obj = Child(argC, argP)
      obj = obj@Parent(argP);
      obj.propC = argC;
      end
  end
end
```

See constructor in TrickDie.m