

- 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 superclass
  - Overriding methods in superclass
- Announcements:
  - Discussion this week in classrooms, not the lab
  - Pick up prelim 2 at ACCEL Green Rm during consulting hrs
  - Please check your final exam schedule now and notify us (by emailing Randy Hess) if you have an exam conflict

A weather object can make use of **Intervals** ...

- Define a class **LocalWeather** to store the weather data of a city, including monthly high and low temperatures and precipitation
  - Temperature: low and high → an **Interval**
    - For a year → **length 12 array of Intervals**
  - Precipitation: a scalar value
    - For a year → **length 12 numeric vector**
  - Include the city name: a string

```
classdef LocalWeather < handle
    properties
        city % string
        temps % array of Intervals
        precip % numeric vector
    end

    methods
        ...
    end
end
```

```

classdef LocalWeather < handle
  properties
    city=""; temps=Interval.empty(); precip=0;
  end
  methods
    function lw = LocalWeather(fname)
      fid= fopen(fname,'r');
      s= fgetl(fid);
      lw.city= s(3:length(s));
      for k= 1:3
        s= fgetl(fid);
      end
      for k=1:12
        s= fgetl(fid);
        lw.temps(k)= Interval(str2double(s(4:8), ...
                                str2double(s(12:16))));
        lw.precip(k)= str2double(s(20:24));
      end
      fclose(fid);
    end
    ...
  end %methods
end %classdef

```

```

//Ithaca
//Monthly temperature and p
//Lows (cols 4-8), Highs (c
//Units: English
      15      31      2.08
      17      34      2.06
      23      42      2.64
      34      56      3.29
      44      67      3.19
      53      76      3.99
      58      80      3.83
      56      79      3.63
      49      71      3.69
      NaN     59      NaN
      32      48      3.16
      22      36      2.40

```

```
classdef LocalWeather < handle
  properties
    city=""; temps=Interval.empty();
    precip=0;
  end
  methods
    function lw = LocalWeather(fname)
      ...
    end

    function showCityName(self)
      ...
    end
  end %methods
end %classdef
```

```
classdef LocalWeather < handle
  properties
    city=""; temps=Interval.empty();
    precip=0;
  end
  methods
    function lw = LocalWeather(fname)
      ...
    end

    function showCityName(self)
      disp(self.city)
    end
    ...
  end %methods
end %classdef
```

## Function to show data of a month of **LocalWeather**

```
function showMonthData(self, m)
% Show data for month m, 1<=m<=12.

end
```

Should display which month, the high and low temperatures, and precipitation

## Function to show data of a month of `LocalWeather`

```
function showMonthData(self, m)
% Show data for month m, 1<=m<=12.

mo= { 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', ...
      'July', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec' };
fprintf('%s Data\n', mo{m})
fprintf('Temperature range: ')
disp(self.temps(m))
fprintf('Average precipitation: %.2f\n', ...
      self.precip(m))
end
```

See `LocalWeather.m`

## Observations about our class `Interval`

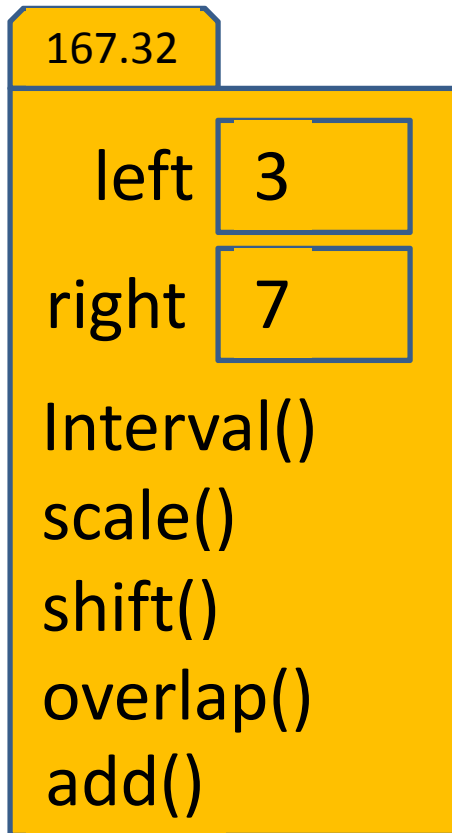
- We can use it (create `Interval` objects) anywhere
  - Within the `Interval` class, e.g., in method `overlap`
  - “on the fly” in the Command Window
  - In other function/script files – not class definition files
  - In another class definition
- Designing a class well means that it can be used in 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.

```
classdef Interval < handle

    properties
        left
        right
    end

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

        function shift(self, s)
            ...
        end

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

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

        ...
    end
end
```

# 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!

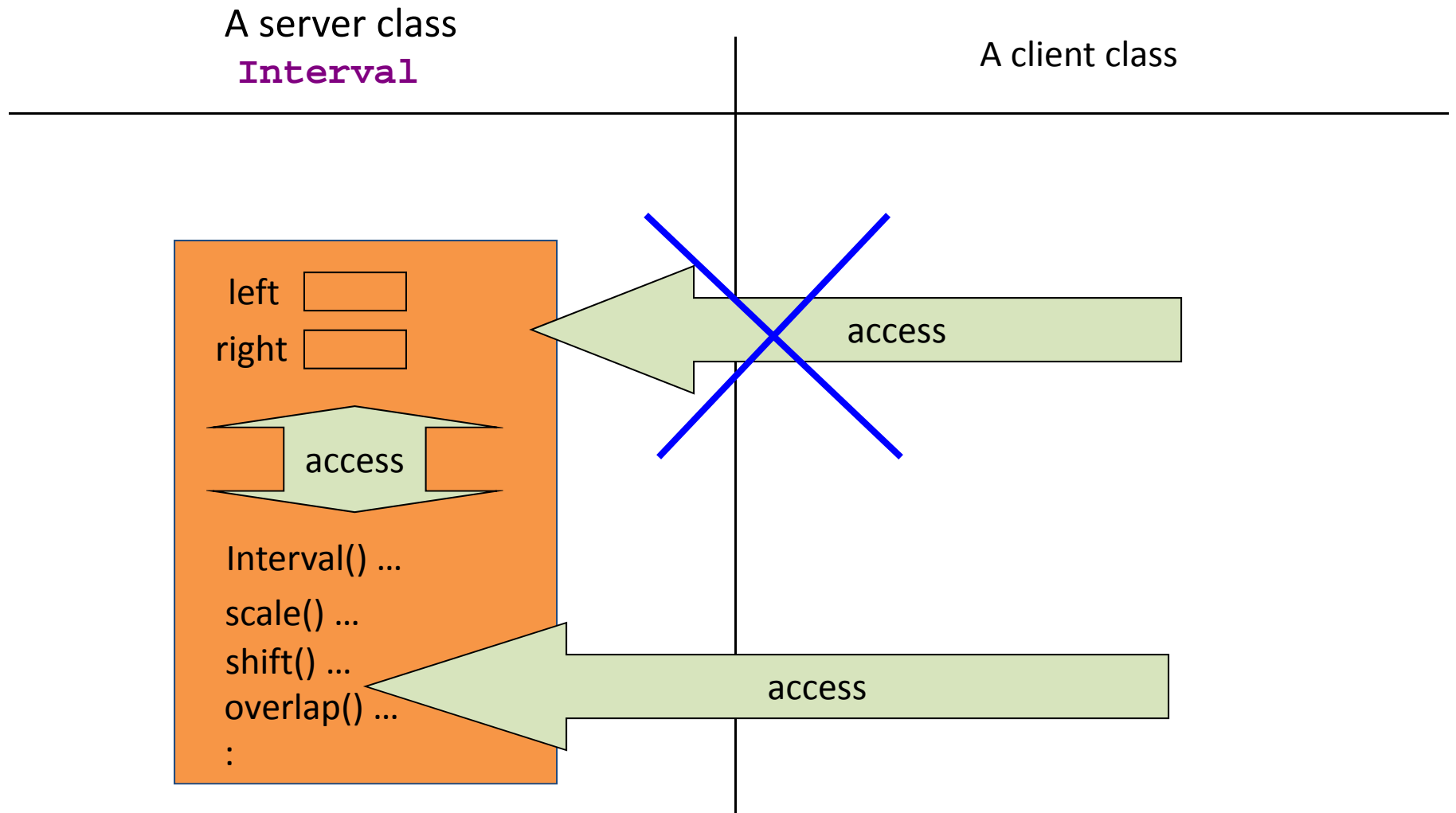
```
classdef Interval < handle
    properties
        left
        right
    end

    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
```

Alternative: use built-in function `error` to halt program execution, e.g.,  
`error('Error at instantiation: left>right')`

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.



Data that the client does not need to access should be protected: **private**  
Provide a set of methods for **public** access.

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
```

*Server*

```
% 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
```

*Example client code*



# 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

```
% 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 (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
end
```

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
```

```
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)
% It 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
```

```
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 setLeft(self, lt)
% the interval's left end gets lt
        self.left = lt;
    end

    function setRight(self, rt)
% the interval's right end gets rt
        self.right = rt;
    end
end
end
```

# 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
```

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

## % 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, 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
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!