• Previous lecture:
  – Why use OOP?
  – Attributes for properties and methods
  – Inheritance: extending a superclass

• Today’s lecture:
  – OOP: Overriding methods in superclass
  – New topic: Recursion

• Announcement:
  – Final exam on Thurs, May 9th, at 9am. 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 Apr 25th.
  – Prelim 2 to be returned at end of lecture. Unclaimed papers (and those on which students didn’t indicate the lecture time) can be picked up during consulting hours (Su-R 5-10p) at ACCEL Green Rm (Carpenter Hall) starting at 5pm today. Submit any regrade request by Sunday, 4/28.
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
```
Make TrickDie a subclass of Die

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*
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
Which components get “inherited”? 

- **public** components get inherited
- **private** components exist in object of child class, but cannot be **directly accessed** in child class \( \Rightarrow \) 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
td = TrickDie(2, 10, 6);
disp(td.sides)

% disp statement is incorrect because

A Property sides is private.
B Property sides does not exist in the TrickDie object.
C Both a, b apply
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`
Overridden methods: which version gets invoked?
To create a TrickDie: call the TrickDie constructor, which calls the Die constructor, which calls the roll method. Which roll method gets invoked?

classdef Die

function D=Die(...) ... D.roll()
end

function roll(self)

end
...
end

classdef TrickDie < Die

function TD=TrickDie(...) ...
TD@Die(...);

end

function roll(self)

end
...
end
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

```matlab
classdef Child < Parent

properties
    propC
end

methods
    ...
    function x = method(arg)
        y = method@Parent(arg);
        x = ... y ... ;
    end
    ...
end
```

See method `disp` in `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” $\Rightarrow$ “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
(Cell) array of objects

- A cell array can reference objects of different classes
  
  \[
  A\{1\} = \text{Die}();
  \]
  
  \[
  A\{2\} = \text{TrickDie}(2,10); \quad \% \text{ OK}
  \]

- A simple array can reference objects of only one single class
  
  \[
  B(1) = \text{Die}();
  \]
  
  \[
  B(2) = \text{TrickDie}(2,10); \quad \% \text{ ERROR}
  \]

- (Assignment to B(2) above would work if we define a “convert method” in class TrickDie for converting a TrickDie object to a Die. We won’t do this in CS1112.)
End of Matlab OOP in CS1112

OOP is a concept; in different languages it is expressed differently.

In CS (ENGRD) 2110 you will see Java OOP
Recursion

• The Fibonacci sequence is defined \textit{recursively}:
  \begin{align*}
  F(1) &= 1, \quad F(2) = 1, \\
  F(3) &= F(1) + F(2) = 2, \\
  F(4) &= F(2) + F(3) = 3
  \end{align*}
  \[ F(k) = F(k-2) + F(k-1) \]
  It is defined in terms of itself; its \textit{definition invokes itself}.

• Algorithms, and functions, can be recursive as well. I.e., a \textit{function can call itself}.

• Example: remove all occurrences of a character from a string
  \begin{verbatim}
  'gc aatc gga c '  \rightarrow  'gcaatcggac'
  \end{verbatim}
Example: removing all occurrences of a character

- Can solve using iteration—check one character (one component of the vector) at a time

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>...</th>
<th>k</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>'C'</td>
<td>'S'</td>
<td></td>
<td>'1'</td>
<td>'1'</td>
</tr>
</tbody>
</table>

Subproblem 1:
Keep or discard $s(1)$

Subproblem 2:
Keep or discard $s(2)$

Subproblem $k$:
Keep or discard $s(k)$

Iteration: Divide problem into a sequence of equal-sized, identical subproblems

See `RemoveChar_loop.m`
Example: removing all occurrences of a character

- Can solve using **recursion**
  - Original problem: remove all the blanks in string s
  - Decompose into two parts: 1. remove blank in s(1)
    2. remove blanks in s(2:length(s))
function s = removeChar(c, s)
% Return string s with character c removed

if length(s) == 0  % Base case: nothing to do do
    return
else
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0 % Base case: nothing to do
terun
else
    if s(1)~=c
        else
            end
    end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0  % Base case: nothing to do
    return
else
    if s(1)~=c
        % return string is
        % s(1) and remaining s with char c removed

        else


    end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0  % Base case: nothing to do
    return
else
    if s(1)~=c
        % return string is % s(1) and remaining s with char c removed
    else
        % return string is just
        % the remaining s with char c removed
    end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s) == 0  % Base case: nothing to do
    return
else
    if s(1) ~= c
        % return string is
        % s(1) and remaining s with char c removed
        s = [s(1) removeChar(c, s(2:length(s)));
    else
        % return string is just
        % the remaining s with char c removed
    end
end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s) == 0  % Base case: nothing to do
    return
else
    if s(1) ~= c
        % return string is
        % s(1) and remaining s with char c removed
        s = [s(1) removeChar(c, s(2:length(s)));
    else
        % return string is just
        % the remaining s with char c removed
        s = removeChar(c, s(2:length(s)));
    end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0  % Base case: nothing to do
    return
else
    if s(1)~=c
        % return string is
        % s(1) and remaining s with char c removed
        s= [s(1) removeChar(c, s(2:length(s)))];
    else
        % return string is just
        % the remaining s with char c removed
        s= removeChar(c, s(2:length(s)));
    end
end
end
function s = removeChar(c, s)
if length(s) == 0
    return
else
    if s(1) ~= c
        s = [s(1) removeChar(c, s(2:length(s)))];
    else
        s = removeChar(c, s(2:length(s)));  
    end
end

removeChar - 1st call

```plaintext
  c _
  s d_ o_ g
```
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s= [s(1) removeChar(c, s(2:length(s)))];
        else
            s= removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)==c
        s= [s(1) removeChar(c, s(2:length(s)))];
    else
        s= removeChar(c, s(2:length(s)));
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)~=c
        s = [s(1) removeChar(c, s(2:length(s)))]
    else
        s = removeChar(c, s(2:length(s)));
    end
end
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
    if length(s) == 0
        return
    else
        if s(1) ~= c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)~=c
        s= [s(1) removeChar(c, s(2:length(s)))];
    else
        s= removeChar(c, s(2:length(s)));
    end
end

removeChar – 1st call
removeChar – 2nd call
removeChar – 3rd call
removeChar – 4th call
removeChar – 5th call
removeChar – 6th call
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)==c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s= [s(1) removeChar(c, s(2:length(s)))];
        else
            s= removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)==c
        s= [s(1) removeChar(c, s(2:length(s)))];
    else
        s= removeChar(c, s(2:length(s)));
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)~=c
        s = [s(1) removeChar(c, s(2:length(s)))];
    else
        s = removeChar(c, s(2:length(s)));
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)~=c
        s= [s(1) removeChar(c, s(2:length(s)))];
    else
        s= removeChar(c, s(2:length(s)));
    end
end
Key to recursion

• Must identify (at least) one base case, the “trivially simple” case
  – no recursion is done in this case

• The recursive case(s) must reflect progress towards the base case
  – E.g., give a shorter vector as the argument to the recursive call – see removeChar