■ Previous Lecture:
  ■ Inheritance in OOP
  ■ Overriding methods

■ Today’s Lecture:
  ■ Recursion
    ■ Remove all occurrences of a character in a string
    ■ A mesh of triangles

■ Announcements:
  ■ Discussion in the lab this week. Attendance is optional but be sure to do the posted exercise.
  ■ Project 6 due Thurs Dec 1 at 11pm. Remember academic integrity!
  ■ Office/consulting hours end Tuesday (tonight) for Thanksgiving Break and will resume Monday
Inheritance

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

![Class Diagram]

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
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

```matlab
classdef Child < Parent
    properties
        propC
    end
    methods
        ...
        function x = method(arg)
            y = method@Parent(arg);
            x = ... y ... ;
        end
        ...
        end
    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 recursively:
  \[ F(1) = 1, \quad F(2) = 1, \]
  \[ F(3) = F(1) + F(2) = 2 \]
  \[ F(4) = F(2) + F(3) = 3 \]

- It is defined in terms of itself; its definition invokes itself.

- Algorithms, and functions, can be recursive as well. I.e., a function can call itself.

- Example: remove all occurrences of a character from a string
  `gc aatc gga c` \(\rightarrow\) `gcaatcggac`
Example: removing all occurrences of a character

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

\[
\begin{array}{ccccccc}
\text{s} & \text{1} & \text{2} & \ldots & \text{k} & \ldots \\
\text{`c'} & \text{`s'} & \text{'} & \text{'} & \text{`1'} & \text{`1'} & \text{`1'} & \text{`2'} \\
\end{array}
\]

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 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
    return
else
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
        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  % s(1)==c

    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  % s(1)==c
        % 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  % s(1)==c
        % 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  % s(1)==c
        % 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  % s(1)==c
        % 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)
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

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

s = d_o_g

removeChar - 1st call

s = d_o_g

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
end

removeChar - 1st call

removeChar - 2nd 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

removeChar – 1st call
\[
\begin{array}{c}
\text{c} \\
\text{s} \rightarrow \text{d o g} \\
[\text{d } \_ \_ ]
\end{array}
\]

removeChar – 2nd call
\[
\begin{array}{c}
\text{c} \\
\text{s} \rightarrow \_ o g \\
[\_ o g ]
\end{array}
\]

removeChar – 3rd call
\[
\begin{array}{c}
\text{c} \\
\text{s} \rightarrow \_ g \\
[\_ g ]
\end{array}
\]
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
```
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
```matlab
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)))]
            s = removeChar(c, s(2:length(s)));
        end
    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

removeChar – 1st call
[ d o g ]

removeChar – 2nd call
[ o g ]

removeChar – 3rd call
[ o g ]

removeChar – 4th call
[ g ]

removeChar – 5th call
[ g ]

removeChar – 6th call
[ g ]
function s = removeChar(c, s)
    if length(s) == 0
        return
    elseif 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
Divide-and-conquer methods, such as recursion, is useful in geometric situations.

Chop a region up into triangles with smaller triangles in “areas of interest”

Recursive mesh generation
Why is mesh generation a divide-&-conquer process?

Let’s draw this graphic
Start with a triangle
A “level-1” partition of the triangle

(obtained by connecting the midpoints of the sides of the original triangle)

Now do the same partitioning (connecting midpts) on each corner (white) triangle to obtain the “level-2” partitioning
The “level-2” partition of the triangle
The “level-3” partition of the triangle
The “level-4” partition of the triangle
The “level-4” partition of the triangle
The basic operation at each level

if the triangle is small
Don’t subdivide and just color it yellow.

else
Subdivide:
Connect the side midpoints;
color the interior triangle magenta;
apply same process to each outer triangle.

end
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning. Assume hold is on.

if L==0
  % Recursion limit reached; no more subdivision required.
  fill(x,y,'y')  % Color this triangle yellow

else
  % Need to subdivide: determine the side midpoints; connect
  % midpts to get “interior triangle”; color it magenta.

  % Apply the process to the three "corner" triangles...

end
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning. Assume hold is on.

if L==0
  % Recursion limit reached; no more subdivision required.
  fill(x,y,'y')  % Color this triangle yellow
else
  % Need to subdivide: determine the side midpoints; connect midpts to get “interior triangle”; color it magenta.
  a = [(x(1)+x(2))/2 (x(2)+x(3))/2 (x(3)+x(1))/2];
  b = [(y(1)+y(2))/2 (y(2)+y(3))/2 (y(3)+y(1))/2];
  fill(a,b,'m')
  % Apply the process to the three "corner" triangles...
end
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning. Assume hold is on.

if L==0
  % Recursion limit reached; no more subdivision required.
  fill(x,y,'y')  % Color this triangle yellow

else
  % Need to subdivide: determine the side midpoints; connect
  % midpts to get “interior triangle”; color it magenta.
  a = [(x(1)+x(2))/2 (x(2)+x(3))/2 (x(3)+x(1))/2];
  b = [(y(1)+y(2))/2 (y(2)+y(3))/2 (y(3)+y(1))/2];
  fill(a,b,'m')

  % Apply the process to the three "corner" triangles...
  MeshTriangle([x(1) a(1) a(3)],[y(1) b(1) b(3)],L-1)
  MeshTriangle([x(2) a(2) a(1)],[y(2) b(2) b(1)],L-1)
  MeshTriangle([x(3) a(3) a(2)],[y(3) b(3) b(2)],L-1)
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
  - E.g., ask for a lower level of subdivision in the recursive call – see MeshTriangle