• Previous lecture:
  – Why use OOP?
  – Attributes for properties and methods
• Today’s lecture:
  – Inheritance: extending a superclass
  – Overriding methods in superclass
  – New topic: Recursion
• Announcements:
  – Discussion this week in classrooms as listed on roster
  – Pick up Prelim paper from consultants during consulting hours
  – Project 6 due on Tuesday, May 10, at 11pm.
  – Remember academic integrity! We will check all submissions using MOSS.
  – Final exam on Thursday, May 19th, 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).

```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
```

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!

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

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

Constructor: must call the superclass’ constructor
Syntax
```matlab
classdef Child < Parent
properties
propC
end
methods
function obj = Child(argC, argP)
obj = obj@Parent(argP);
obj.propC = argC;
…
end
end
```

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

Important ideas in inheritance
• Keep common features as high in the hierarchy as reasonably possible
• Use the superclass' features as much as possible
• “Inherited” => “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)= Die();
  A(2)= TrickDie(2,10); % OK
• A simple array can reference objects of only one single class
  B(1)= Die();
  B(2)= TrickDie(2,10); % 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.)

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

Important ideas in inheritance
• Keep common features as high in the hierarchy as reasonably possible
• Use the superclass' features as much as possible
• “Inherited” => “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)= Die();
  A(2)= TrickDie(2,10); % OK
• A simple array can reference objects of only one single class
  B(1)= Die();
  B(2)= TrickDie(2,10); % 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.)
Recursion

- The Fibonacci sequence is defined recursively:
  \[ F(1) = 1, \quad F(2) = 1, \quad F(3) = F(1) + F(2) = 2, \quad F(k) = F(k-2) + F(k-1) \]
- 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` → `gcaatcggac`

Example: removing all occurrences of a character

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

- 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)`

```matlab
function s = removeChar(c, s)
    if length(s)==0  % Base case: nothing to do
        return
    else
        if s(1)~=c  % return string is 
            s= [s(1) removeChar(c, s(2:length(s)))];
        else        % the remaining s with char c removed
            s= removeChar(c, s(2:length(s)));
        end
    end
end
```
Function: \( s = \text{removeChar}(c, s) \)

- if length\((s)\) == 0
  - return
- else
  - if \( s(1) \neq c \)
    - \( s = [s(1) \text{ removeChar}(c, s(2:length(s)))] \)
  - else
    - \( s = \text{removeChar}(c, s(2:length(s))) \)
  - 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