Lecture 5
Advanced MATLAB:
Object-Oriented Programming

Matthew J. Zahr

CME 292
Advanced MATLAB for Scientific Computing
Stanford University

7th October 2014

What is OOP?

- Procedural programming is a list of instructions for the computer to perform to accomplish a given task
  - Code and data
  - No association between functions and the data on which they operate
  - Languages: FORTRAN, C
- Object-oriented programming (OOP) is a programming paradigm organized around *objects* equipped with data fields and associated methods.
  - Data (state) and methods (behavior) associated via objects
  - Objects used to interact with each other
  - Languages: C++, Objective-C, Smalltalk, Java, C#, Perl, Python, Ruby, PHP

Why use OOP?

OOP enables a level of modularity and abstraction not generally available in procedural languages

- Increased code understanding
- Code maintenance
- Code expansion/evolution

OOP Fundamentals

- **Class**: template for creating *objects*, defining properties and methods, as well as default values/behavior
- **Object**: instance of a *class* that has a state (properties) and behavior (methods)
- **Properties**: data associated with an object
- **Methods**: functions (behavior) defined in a class and associated with an object
- **Attributes**: modify behavior of classes and class components
- **Inheritance**: object or class (subclass) derived from another object or class (superclass)
- **Polymorphism**: single interface to entities of different types

Other OOP features include *events* and *listeners*, which will not be covered

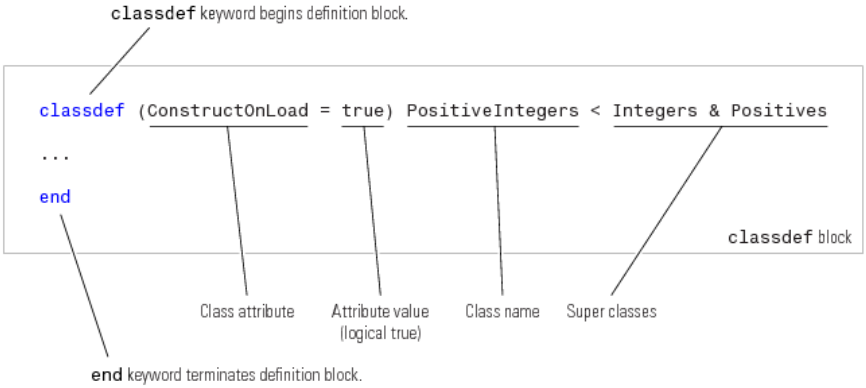
Class Components in MATLAB

- `classdef` block
  - Contains class definition, class attributes, and defines superclasses
- `properties` block
  - Defines all properties to be associated with a class instance
  - Defines attributes of all properties and default values
- `methods` block
  - Defines methods associated with the class and their attributes
  - First method must have the same name as the class, called the *constructor*
- `event` block
- `enumeration` block

`http://www.mathworks.com/help/matlab/matlab_oop/class-components.html`

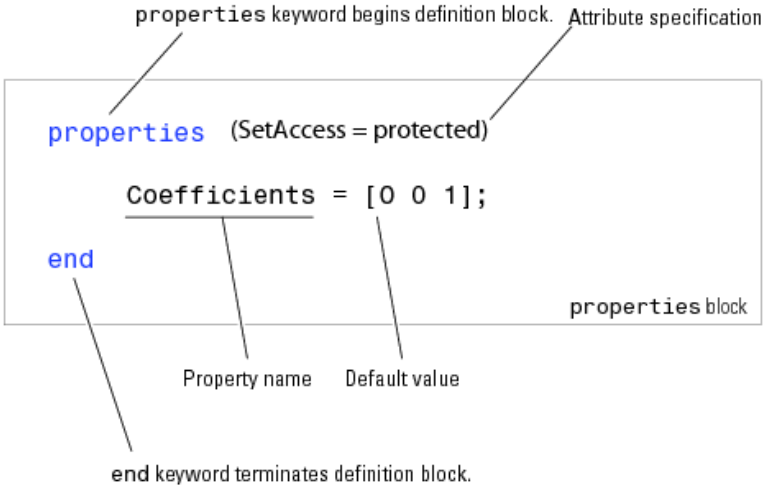
Class Block

- *Class definitions* - blocks of code delineated with `classdef .. end` keywords
- Specify attributes and superclasses
- Contains `properties`, `methods`, `events` subblocks
- One class definition per file
- Only comments and blanks can precede `classdef`



Properties: Definition/Initialization

- Properties are variables associated a particular class
- Defined in special `properties` block
- Can be multiple `properties` blocks, each with own attributes



Properties: Initialization/Attributes

```
1 classdef class_name
2    properties
3       PropertyName
4    end
5    properties (SetAccess=private,GetAccess=public)
6       PropertyName = 'some text';
7       PropertyName = sin(pi/12);
8    end
9 end
```

- Property attributes: `http://www.mathworks.com/help/matlab/matlab_oop/property-attributes.html`

Methods

- Methods are MATLAB functions associated with a particular class
- Defined in special `methods` block
- Can be multiple `methods` blocks

```
1  classdef ClassName
2     methods
3        function obj = ClassName(arg1,arg2,..)
4        end
5        function normal_method(obj,arg1,..)
6        end
7     end
8     methods (Static = true)
9        function static_method(arg1,..)
10       end
11    end
12 end
```

Value vs. Handle Class

- There are two *fundamentally* different types of classes in MATLAB
  - *Value* class
  - *Handle* class
- An instance of a *value* class behaves similar to *most* MATLAB objects
  - A variable containing an instance of a value class *owns* the data associated to it
  - Assigning object to new variable *copies* the variable
- Conversely, an instance of a *handle* class behaves similar to MATLAB graphics handles
  - A variable containing an instance of a handle class is a *reference* to the associated data and methods
  - Assigning object to a new variables makes a new *reference* to same object
  - Events, listeners, dynamic properties
- `http://www.mathworks.com/help/matlab/handle-classes.html`

Examples

The remainder of this lecture will be done in the context of two examples

- `polynomial.m`
  - A value class for handling polynomials of the form

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_m x^m$$

    in a convenient and simple way
  - Simple interface for performing operations of polynomials to create new ones
- `dsg_elem_def.m`
  - A handle class for graphically deforming the deformation of a body

Both examples are incomplete. We will (mostly) complete `polynomial.m` throughout the remainder of the lecture. You will have the opportunity to extend both in Homework 3.

polynomial class

```
classdef polynomial
    %POLYNOMIAL
    properties (GetAccess=public,SetAccess=private)
        coeffs=0;
        order =0;
    end

    methods
        function  self = polynomial(arg)
        function  [tf] = iszero(poly)
        function  [y]  = evaluate(poly,x)
        function  [apoly] = plus(poly1,poly2)
        function  [mpoly] = minus(poly1,poly2)
        function  [ipoly] = integrate(poly,const)
        function  [dpoly] = differentiate(poly)
        function  [iseq] = eq(poly1,poly2)
        function  [] = plot_it(poly,x,pstr,ax)
        function  [] = disp(poly)
```

Constructor - Create instance of class

To create an instance of a class for a list of arguments, call its *constructor*

- By definition, the constructor is the *first* method in the first method block
- It is required to have the same name as the class (`polynomial` in our case)
- Responsible for setting properties of class based on input arguments
  - Properties not set will be given default value
  - Default value either `[]` or defined in `properties` block
- Returns instance of class
- See `polynomial` in `polynomial.m`

```
>> p1 = polynomial([1,2,3]);   %3x^2+2x+1
>> p2 = polynomial(p1);        %3x^2+2x+1
>> p3 = polynomial([1,2,3,0]); %3x^2+2x+1
```

Object Arrays

Similar to arrays of numbers, cells, and structures, we can define *objects* arrays as an array where each element is an instance, or object, of a particular class

```
>> p(1,7) = polynomial([1,2,3]);    %3x^2+2x+1
>> length(p)
ans =
     7
>> p(3)
ans =
0.0000
>> p(7)
ans =
1.0000 + 2.0000 x + 3.0000 x^2
```

Accessing Properties

Properties are accessed using the `.` operator, similar to accessing *fields* in a structure.

```
>> p1.order
ans =
     2
>> p2.coeffs
ans =
     1     2     3
```

Public vs. private properties

Recall the `properties` block definition of `polynomial`

```
properties (GetAccess=public,SetAccess=private)
        coeffs=[];
        order =0;
end
```

- `GetAccess`, `SetAccess` define where the properties can be queried or set, respectively
- `public` properties have unrestricted access
- `protected` properties can only be accessed from *within* class or subclass
- `private` properties can only be accessed from *within* class

```
p3.coeffs = [5,2,3];
??? Setting the 'coeffs' property of the 'polynomial' class ...
    is not allowed.
```

Types of Methods

This information is directly from `http://www.mathworks.com/help/matlab/matlab_oop/how-to-use-methods.html`

- **Ordinary** methods - functions that act on one or more objects (plus additional data) and return a new object or some computed value
- **Constructor** methods - special function that creates the objects of a class
- **Destructor** methods - function called when instance of class is deleted
- **Statics** methods - functions associated with a class that do not necessarily act on class objects

Using Methods

- All methods must accept the *class instance* as their *first* argument
- Methods can be accessed in two main ways
  - Using the `.` operator with the class instance
    - Implicitly passes the class instance as the first argument
  - Directly passing the class instance as the first argument

```
>> p3.iszero()
ans =
     0
>> p3.evaluate(0:0.25:1.0)
ans =
    1.0000    1.6875    2.7500    4.1875    6.0000
>> p4 = polynomial(0);
>> p4.iszero()
ans =
     1
```

Implementing Operators

- Operators such as +, −, *, .*, ==, <, >, etc can be overload for a given class
- Simply implement a method with an appropriate *name* and number of argument
  - A list of operators and their corresponding name are listed here
- When operator such as + called, it uses the data type to determine when function is called

```
function  [iseq] = eq(poly1,poly2)
       iseq = all(poly1.coeffs == poly2.coeffs);
end
```

```
>> p1 == p2
ans =
     1
>> p1 == p4
ans =
     0
```

Assignment: `polynomial`

In `polynomial.m`, implement

- `plus` to overload the + operator to return $p_3(x) = p_1(x) + p_2(x)$
- `minus` to overload the − operator to return $p_3(x) = p_1(x) - p_2(x)$
- `differentiate` to return $p'(x)$
- `integrate` to return $\int p(x)\, dx$

Then, define $p_1(x) = 10x^2 + x - 3$ and $p_2(x) = 2x^3 - x + 9$. Use the polynomial class to

- compute the polynomial $p_3(x)$ defined as $p_3(x) = p_1(x) + p_2(x)$
- compute the polynomial $p_4(x)$ defined as $p_4(x) = p_1(x) - p_2(x)$

Assignment: `polynomial`

- Construct simple example to check implementation of `mtimes` and `mpower`

Assignment: `polynomial`

Define the piecewise cubic polynomial

$$p(x) = \begin{cases} x^3 - 6x + 2 & \text{for } x \in [-1, 0] \\ x^3 + x^2 + 2 & \text{for } x \in [0, 1] \end{cases}$$

- compute the derivative of $p(x)$ (the fact that it does not exist at 0 should not cause problems)
- compute the definite integral of $p(x)$ over $[-1, 1]$

Handle Class

- Handle class is a *reference* to data and methods (similar to graphics handles as references to graphics objects)
- In contrast to value classes, *handle* classes enable you to create an object that more than one function can share
- Declare class a *handle* class by having it inherit from the handle `superclass`
- Similar to handle classes,
  - the first argument of all methods must be the class instance itself
  - methods are invoked identically

```
classdef  dsg_elem_def < handle
    properties
    end
end
```

Handle Class

- Unlike value classes, a method in a handle class can *modify* properties of the class instance
- Removes need for instantiating new objects and returning them in methods
  - A method can simply modify the properties of the instance in place
  - Does not necessarily require an output

Example: `dsg_elem_def`

- See Homework 3 handout for details
- Demo