OOP in Matlab

Table of Contents

Alban, pmxal9@nottingham.ac.uk

Foreword

Scientists would gain to learn software development practices.

Best Practices for Scientific Computing, PLoS Biology 2014. http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745

Scientists typically develop their own software for these purposes because doing so requires substantial domain-specific knowledge. As a result, recent studies have found that scientists typically spend 30% or more of their time developing software. However, 90% or more of them are primarily self-taught, and *therefore lack exposure to basic software development practices such as writing maintainable code, using version control and issue trackers, code reviews, unit testing, and task automation.*

[...]

A large body of research has shown that code reviews are the most cost-effective way of finding bugs in code. They are also a good way to spread knowledge and good practices around a team. In projects with shifting membership, such as most academic labs, code reviews help ensure that critical knowledge isn't lost when a student or postdoc leaves the lab.

- Look at some best practice tips,
- Show your code (fellow PhDs, supervisor, random guy, student).

Prepare workspace

This code is just preparing my workspace.

```
clear;          % Clear workspace
cPath = fullfile('/
Users','pmaal','Dropbox','Nottingham','repos','phd','Seminars','MatlabOOP');
cd(cPath);      % Change directory
```

Preintroduction

Structures in MATLAB®

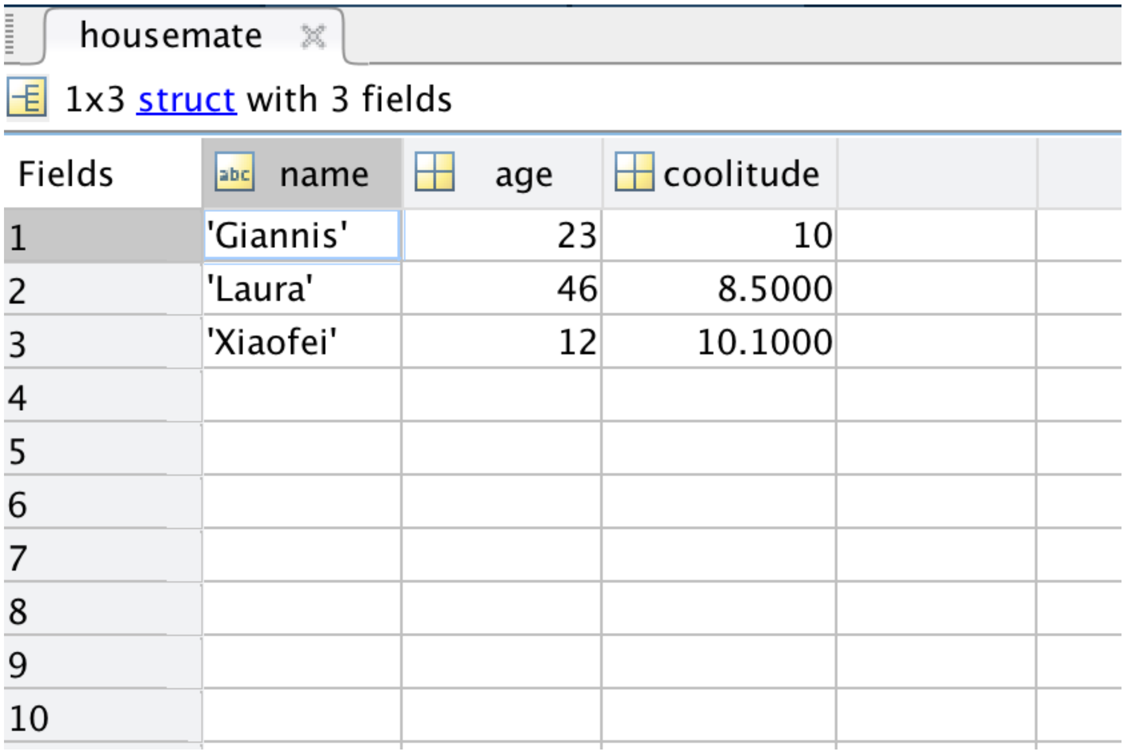
A structure array is a data type that groups related data using data containers called fields. Each field can contain any type of data. Access data in a structure using dot notation of the form structName.fieldName.

Ex: To capture all infos regarding my classes this term, I'll use struct:

```
thisTerm.nbCoursesIDemonstrate = 2;
thisTerm.listOfCourses = 'G11MAN G59ZZZ';
disp( thisTerm );

    nbCoursesIDemonstrate: 2
            listOfCourses: 'G11MAN G59ZZZ'


sameInfo = struct('nbCoursesIDemonstrate', 2,'listOfCourses', 'G11AAA
 G59ZZ');
disp( sameInfo );

    nbCoursesIDemonstrate: 2
            listOfCourses: 'G11AAA G59ZZ'
```

Ex: If I need a list of structures with the same fields: structure array.

```
housemate(1).name = 'Giannis';
housemate(1).age = 23;
housemate(1).coolitude = 10;
housemate(2).name = 'Laura';
housemate(2).age = 46;
housemate(2).coolitude = 8.5;
housemate(3).name = 'Xiaofei';
housemate(3).age = 12;
housemate(3).coolitude = 10.1;
```



1x3 struct with 3 fields

Fields	name	age	coolitude
1	'Giannis'	23	10
2	'Laura'	46	8.5000
3	'Xiaofei'	12	10.1000
4			
5			
6			
7			
8			
9			
10			

Average coolness of my housemates:

```
mean([housemate.coolitude])
```

```
ans =

    9.5333
```

As easily: nested structures

```
C.M.A = ones(3);
C.M.M.A = zeros(4,1);
C.M.M.B = eye(4);
C.M.C = magic(5);
```

What is the value of C.M.M.B(6)?

Be wary of using 'struct':

```
thisTerm.nbCoursesIDemonstrate = 2;
thisTerm.listOfCourses = {'G11AAA','G59ZZ'};
disp( thisTerm );
```

```
    nbCoursesIDemonstrate: 2
```

```
            listOfCourses: {'G11AAA'  'G59ZZ'}
```

```
notSameInfo = struct('nbCoursesIDemonstrate', 2,'listOfCourses',
 {'G11AAA','G59ZZ'});
disp( notSameInfo );
```

```
1x2 struct array with fields:

    nbCoursesIDemonstrate
    listOfCourses
```

Also possible to initialise a structure array:

```
structArray(15).name = 'NoName';
structArray(16).age = 'noAge';
```

What is length(structArray)?

Open structArray

Workspaces in MATLAB®

The **base workspace** stores variables that you create at the command line. This includes any variables that scripts create, assuming that you run the script from the command line or from the Editor. Variables in the base workspace exist until you clear them or end your MATLAB® session.

Functions do not use the base workspace. Every function has its own **function workspace**. Each function workspace is separate from the base workspace and all other workspaces to protect the integrity of the data. Even local functions in a common file have their own workspaces. Variables specific to a function workspace are called local variables. Typically, local variables do not remain in memory from one function call to the next.

Questions

```
aud = audience('University of Nottingham');
aud.nbPeople = 300;
```

Who already knew what a structure was?

```
aud.nbPplKnowStructures = 85;
```

Who knows what an object is? (in OOP sense)

```
aud.nbPplKnowObjects = 25;
```

Who knows what inheritance is?

```
aud.nbPplKnowInheritance = 20;
```

Who has used OOP in MATLAB?

```
aud.nbPplUsedOOP = 1;
```

Who has written classes in MATLAB?

```
aud.nbPplWrittenClasses = 1;
```

```
disp( aud );
```

```
  audience with properties:

                      time: '28-Oct-2015 21:47:33'
                  nbPeople: 300
                    isnice: 1
                     venue: 'University of Nottingham'
          nbPplKnowObjects: 25
       nbPplKnowStructures: 85
      nbPplKnowInheritance: 20
             nbPplUsedOOP: 1
       nbPplWrittenClasses: 1
```

So far, aud looks like a structure. What would the following do?

aud.time = 42;

Introduction: OOP & MATLAB®

Mathworks introduced in 2008a an entirely new OO framework and syntax, bringing it more in line with Java, python, C++, etc.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. The first OOP languages come from the 60's.

MATLAB® is a multi-paradigm programming language: Interactive command line, scripts and functions, object-oriented programming.

(What) OOP: Main concepts

At a first approximation, objects in Matlab are just structs. Here are some important differences:

- Both structs and objects store data in named fields, (called properties in objects) but objects also encapsulate the operations you can perform on them, (called methods).
- Structs can be created with whatever fields you like but all objects of the same class have the same properties. The values for these properties will differ.
- Structs are created with the struct() function, objects use constructors.

Everything has a class

In MATLAB®, every value is assigned to a class (predefined or user-defined).

```
a = 7;
b = 'some text';
st.Name = 'Nancy';
st.Age = 64;
whos
```

```
  Name                Size                Bytes  Class        Attributes
  C                 1x1                1488  struct
  a                 1x1                   8  double
  ans               1x1                   8  double
  aud               1x1                 112  audience
  b                 1x9                  18  char
  cPath             1x60                120  char
  housemate         1x3                1286  struct
  notSameInfo       1x2                 614  struct
  sameInfo          1x1                 384  struct
  st                1x1                 370  struct
  structArray       1x16                614  struct
  thisTerm          1x1                 606  struct
```

Let's look at a class that has something more

```
c = nottinghamClass();  % nottinghamClass is a class,
disp( c );              % c is an instance of this class.
```

```
  nottinghamClass with properties:

         time: '28-Oct-2015 21:47:33'
    path2logo: './notts.jpg'
         logo: []
```

We can as easily create an array of this class.

```
doubleArray(6) = 10;              % Creates double array filled with
 default values (0) and a 10.
structArray(6).myfield = 'a';     % Creates structure array, default
 value '', and a 'a'.
classArray(6) = audience();       % Creates audience array
arrayfun( @(ind)classArray(ind).setNbPeople(ind^2), 1:6 ) % No output
```

Open structArray, Open classArray. Note the difference in visualisation in the variables' viewer.

```
disp( structArray );
disp( classArray );
```

```
1x16 struct array with fields:

    name
    age
    myfield

  1x6 audience array with properties:

    time
    nbPeople
    isnice
    venue
    nbPplKnowObjects
    nbPplKnowStructures
    nbPplKnowInheritance
    nbPplUsedOOP
    nbPplWrittenClasses
```

Check we can get all values from a given field like with structures:

```
disp({structArray.myfield})

  Columns 1 through 13

    []    []    []    []    []    'a'    []    []    []    []    []
 []    []

  Columns 14 through 16

    []    []    []


disp({classArray.time})

  Columns 1 through 3

    '28-Oct-2015 21:4...'    '28-Oct-2015 21:4...'    '28-Oct-2015
 21:4...'

  Columns 4 through 6

    '28-Oct-2015 21:4...'    '28-Oct-2015 21:4...'    '28-Oct-2015
 21:4...'


disp([classArray.nbPeople])

     1     4     9    16    25    36
```

Look at the (non-hidden) methods of our nottinghamClass object

```
methods( c )


Methods for class nottinghamClass:

nottinghamClass  saveasjpg        todaysLogo

Call "methods('handle')" for methods of nottinghamClass inherited from
 handle.
```

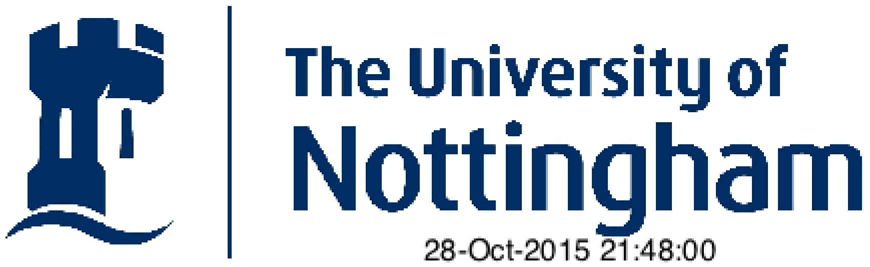
and let's look at the image it contains. Calling 'c.todaysLogo' is the same as calling 'todaysLogo(c)'

```
c.todaysLogo;
```

The University of
Nottingham
28-Oct-2015 21:47:33

Check we can change the time

```
c.time = datestr( now + 0.0003 );
c.todaysLogo;
c.saveasjpg;
```



```
close( c.logo );
```

Dynamic referencing

Just like with structures, we can use the syntax myClass.(varContainingFieldName) to access the values from a field.

```
propName = 'time';
myStruct.(propName) = datestr( now );
fprintf('myStruct.%s contains ''%s''\n' ,propName,myStruct.
(propName));
fprintf('c.%s contains ''%s''\n'        ,propName,c.(propName));
```

```
myStruct.time contains '28-Oct-2015 21:47:36'
c.time contains '28-Oct-2015 21:48:00'
```

(Why) Good Programming

Good practice with classes:

- MATLAB is increasingly using OOP.
- Always use case sensitive method names in your MATLAB® code (always).
- Choose the attributes that fit your use (ex: private to simplify what you can do in a script).

- Think about your hierarchy of classes before starting to code (will diminish coding errors and reduce restructuring).
- Choose the level of asbtraction you want to work in (ex: c2d rather than c2d_char in example to come).
- Resulting code easily extendable and reusable.
- Replace spaghetti code by more robust and maintainable code.
- Documentation provides a simple test to evaluate how clear your comments are.

(How) Write classes in MATLAB®

In file myClassSyntax.m, accessible to MATLAB's path:

```
classdef myClassSyntax < superClass1 & superClass2
    properties ( Attribute1 = value1, Attribute2 = value2 )
    end

    methods ( Attribute1 = value1, Attribute2 = value2 )
        function obj = myClassSyntax( arg1,arg2, arg3 )
             % Constructor method
             % The constructor can take any number of arguments
 specifying
             % initial values for the properties for instance, and
 must return
             % one argument, the constructed object.
        end

        function out = funcName( obj, arg1 )
            % A method called funcName
        end
    end

    events (Attribute1 = value1, Attribute2 = value2 )
    end
end
end
```

Let's look at a working examples: nottinghamClass.m

nottinghamClass.m:

```
classdef nottinghamClass < handle
    % Subclass of handle made for today's seminar

    % Properties (optional: with their default values)
    properties
        % Time of creation (first word disappeared in doc because
 property name!)
        time = datestr( now )
        % Local path to logo
        path2logo = fullfile('.','notts.jpg')

        % A figure with the logo
        logo = []
    end

    % One method block for each unique set of attribute
specifications.
    % Unlike in Java or C++, you must pass an object of the class
    %   explicitly to the method.

    % Classes can define a special method to create objects, called a
constructor.
    % Constructor methods enable you to validate and assign property
values.
    % Here is a constructor for the nottinghamClass class:
    methods
        function obj = nottinghamClass( varargin )
            % Constructor method checking number of inputs
            if nargin == 2 || nargin > 3
                error('Dude, no more than one input.');
            elseif nargin == 3
                disp('Secret: 3 arguments is actually fine. It
contains:');
                disp( varargin{3} );
            end
        end
    end

    methods ( Sealed=true )
        % Sealed because I do not plan to change those methods in
subclasses

        function todaysLogo( obj )
            % Add time to Notts' logo
            A = imread( obj.path2logo );
            if isempty( obj.logo )
                obj.logo = figure;
            end
            image( A );
            axis equal
            axis off
            hold on
            text(1000,590,obj.time,'FontSize',15);
        end

        function saveasjpg( obj )
            % Save it in current directory
            if isempty( obj.logo )
                error('Run obj.todaysLogo before calling saveasjpg.');
            end
            saveas( obj.logo, sprintf('notts_%s.jpg', obj.time) );
        end
    end

end
```

we can look at its doc.

Commenting in classes

Best practice is to document interfaces and reasons, not implementations, and in classes, all properties (what do they contain/what are they for) and methods (inputs/outputs, what they do, examples of I/O). MATLAB®'s doc will do the rest.

For example: doc nottinghamSubclass is not perfect. Let's look at doc nottinghamClass (note property 'time')

Blocks: classdef, properties, methods, events, enumeration

```
properties( c )


Properties for class nottinghamClass:

    time
    path2logo
    logo


methods( c )


Methods for class nottinghamClass:

nottinghamClass  saveasjpg        todaysLogo

Call "methods('handle')" for methods of nottinghamClass inherited from
 handle.


properties( aud )


Properties for class audience:

    time
    nbPeople
    isnice
    venue
    nbPplKnowObjects
    nbPplKnowStructures
    nbPplKnowInheritance
    nbPplUsedOOP
    nbPplWrittenClasses


methods( aud )


Methods for class audience:

audience     plus         setNbPeople

Call "methods('handle')" for methods of audience inherited from
 handle.
```

Practical example

that shows the flexibility in working with classes: example use of class: myClass.m

```
methods(myClass('1'))


Methods for class myClass:

a2b          b2c          c2d          myClass      one2another

Call "methods('handle')" for methods of myClass inherited from handle.
```

Superclasses and subclasses

MATLAB® supports multiple inheritance, i.e. subclasses with multiple superclasses. Care must be taken that naming conflicts do not occur.

```
classdef nottinghamSubclass < nottinghamClass & matlab.System &
 myAbstractClass
    % nottinghamSubclass is a subclass of nottinghamClass and of
    % matlab.System, so it inherits all properties and methods from
    % nottinghamClass and matlab.System


    properties ( SetAccess = private )
        % Those properties are read-only for the user:
        % only changeable via methods able to (ex: setNewProp)
        myNewProperty@char          = 'initialValue'
        % This property can't be changed
        myImmutableProp@char        = datestr( now )
        myNewPropertyRestr@char     = 'initialValue'
    end

    properties ( GetAccess = private )
        privateGetAccess = 'privateGetAccess contains: You can''t see
 me unless you call getPrivAcc.';
```

```
    end

    methods
        function set( obj, propName, value )
            % Overwrite set (defined in superclass)
            switch propName
                case 'myImmutableProp'
                    error('Don''t even try to change
myImmutableProp');
                case 'myNewProperty'
                    obj.setNewProp(value);
            end
        end

        function setNewProp( obj, value )
            % Only way to change property myNewProperty
            obj.myNewProperty = value;
        end

        function getPrivAcc( obj )
            disp( obj.privateGetAccess );
        end
    end

    % To demonstrate that private methods can be called by public ones
    methods ( Access='public' )
        function callPrivateMethod( obj, varargin )
            obj.myPrivateMethod('fromCallPrivateMethod');
        end
    end

    methods ( Access='private' )
        function myPrivateMethod( ~, str )
            fprintf(1, 'myPrivateMethod(obj, %s)\n', str );
        end
    end

    methods % Implement the abstract method from myAbstractClass.m
        function [a,b] = myAbstractMethod( obj, arg )
            a = arg;
            b = obj.myNewProperty;
        end
    end

    methods (Static)
        function myNewMethod
            fprintf(1,'I''m the new method, only available to objects
of class nottinghamSubclass.\n');
            fprintf(1,'As a static method, I''m callable from the
class itself: nottinghamSubclass.myNewMethod();\n');
            fprintf(1,'I can be useful: Time is %s.\n',
datestr( now ));
        end
    end
```

```
end
```

Visual represenstation of class hierarchy: !open html/PMTK2-class-diag-models-may09.png

For example, an instance of class nottinghamSubclass looks like this:

```
n = nottinghamSubclass;
disp(n);
```

```
  System: nottinghamSubclass

  Properties:
         myNewProperty: 'initialValue'
       myImmutableProp: '28-Oct-2015 21:47:36'
    myNewPropertyRestr: 'initialValue'
                  time: '28-Oct-2015 21:47:33'
             path2logo: './notts.jpg'
                  logo: []
```

What is the difference between those two classes?

Exercise: Guess final value of wH.d and noH.d:

```
wH = myClass('initialValue');
wH.a2b;
wH.b2c;
wH.c2d;
noH = myClassNoHandle( 'initialValue' );
noH.a = 'initialValue' ;
noH.a2b;
noH.b2c;
noH.c2d;
```

wH.d, noH.d

About superclass 'handle'

Two Copy Behaviors There are two fundamental kinds of MATLAB®? classes?handles and values.

- **Value classes** create objects that behave like ordinary MATLAB variables with respect to copy operations. Copies are independent values. Operations that you perform on one object do not affect copies of that object.

- **Handle classes** create objects that behave as references. A handle, and all copies of this handle, refer to the same object. When you create a handle object, you copy the handle, but not the data referenced by the object's properties. Any operations you perform on a handle object are visible from all handles that reference that object.

Equivalent in C: Call by reference / Call by value

The handle class is the superclass for all classes that follow handle semantics. A handle is a reference to an object. If you copy an object's handle variable, MATLAB? copies only the handle. Both the original and copy refer to the same object. For example, if a function modifies a handle object passed as an input argument, the modification affects the original input object. In contrast, nonhandle objects (that is, value objects) are not references. Functions must return modified value objects to change the object outside of the function's workspace.

See MATLAB's documentation:

Comparing handle and value classes -- Which kind of class to use

Handles are everywhere in MATLAB®

We use 'f' as the number 1 (it's the only reference we, as users, need) but f contains much more information:

```
f = figure(1);
get(f);
close(f)
```

```
            Alphamap: [1x64 double]
        BeingDeleted: 'off'
          BusyAction: 'queue'
       ButtonDownFcn: ''
            Children: []
            Clipping: 'on'
     CloseRequestFcn: 'closereq'
               Color: [0.9400 0.9400 0.9400]
            Colormap: [64x3 double]
           CreateFcn: ''
         CurrentAxes: []
    CurrentCharacter: ''
       CurrentObject: []
        CurrentPoint: [0 0]
           DeleteFcn: ''
        DockControls: 'on'
            FileName: ''
   GraphicsSmoothing: 'on'
    HandleVisibility: 'on'
       IntegerHandle: 'on'
       Interruptible: 'on'
      InvertHardcopy: 'on'
         KeyPressFcn: ''
       KeyReleaseFcn: ''
             MenuBar: 'figure'
                Name: ''
            NextPlot: 'add'
              Number: 1
         NumberTitle: 'on'
    PaperOrientation: 'portrait'
       PaperPosition: [0.2500 2.5000 8 6]
   PaperPositionMode: 'manual'
           PaperSize: [8.5000 11]
           PaperType: 'usletter'
          PaperUnits: 'inches'
              Parent: [1x1 Root]
             Pointer: 'arrow'
   PointerShapeCData: [16x16 double]
 PointerShapeHotSpot: [1 1]
```

```
                 Position: [560 528 560 420]
                 Renderer: 'opengl'
             RendererMode: 'auto'
                   Resize: 'on'
            SelectionType: 'normal'
           SizeChangedFcn: ''
                      Tag: ''
                  ToolBar: 'auto'
                     Type: 'figure'
            UIContextMenu: []
                    Units: 'pixels'
                 UserData: []
                  Visible: 'on'
      WindowButtonDownFcn: ''
    WindowButtonMotionFcn: ''
        WindowButtonUpFcn: ''
        WindowKeyPressFcn: ''
      WindowKeyReleaseFcn: ''
     WindowScrollWheelFcn: ''
              WindowStyle: 'normal'
                 XDisplay: 'Quartz'
```

Cf doc figure; figure(h) does one of the following:

- If h is the handle or the Number property value of an existing figure, then figure(h) makes that existing figure the current figure, makes it visible, and moves it on top of all other figures on the screen. The current figure is the target for graphics output.
- If h is not the handle and is not the Number property value of an existing figure, but is an integer, then figure(h) creates a figure object and assigns its Number property the value h.
- If h is not the handle to a figure and is not a positive integer, then MATLAB® returns an error.

```
try
    % Suspense...
    figure(0);
catch ME
    disp(ME);
    disp(ME.message);
end
```

```
  MException with properties:

    identifier: 'MATLAB:hgbuiltins:figure:InputMustBeAFigureHandle'
       message: 'Single input must be an existing figure handle or a
 scala...'
         cause: {0x1 cell}
         stack: [5x1 struct]

Single input must be an existing figure handle or a scalar integer
 from 1 to 2147483646
```

For fun: a Mathworks tool to explore the graph of some objects' (graphics) properties: Handle Graphics Object Properties

Class folders

There's another way to write classes, using class folders. Here is what is contained in the file haveYouSeenThisMethod.m:

```
classdef haveYouSeenThisMethod < handle
    properties
    end
end
```

```
hsm = haveYouSeenThisMethod();
methods(hsm)
```

```
Methods for class haveYouSeenThisMethod:

haveYouSeenThisMethod  methodCcallingB
methodA

Call "methods('handle')" for methods of haveYouSeenThisMethod
 inherited from handle.
```

```
hsm.methodA
```

```
ans =

from MethodA.m
```

Calling 'hsm.privMethodB' would give an error message: 'Cannot Acces...'.

But we can call this private method from a public method

```
hsm.methodCcallingB
```

```
ans =

Obtained from methodCcallingB.m: from privMethodB.m
```

There are two ways to specify classes with respect to files and folders:

- Create a single, self-contained class definition file in a path folder or a class folder
- Define a class in multiple files, which requires you to use a class folder (@classname) inside a path folder

Ways to organise your folder

methodA, privMethodB and methodCcallingB are contained in separate files. The other attributes we've seen (Hidden, Transient, ...) can't be attributed to these methods! Let's look at the three files:

```
function out = methodA( obj )
    out = 'from MethodA.m';
end
```

```
function out = privMethodB( obj )
    out = 'from privMethodB.m';
end
```

```
function out = methodCcallingB( obj )
    out = sprintf('Obtained from methodCcallingB.m:
 %s',obj.privMethodB);
end
```

Useful if your method is getting too big or if it's easier to work on separate files (version control, testing,...) When defining a class in a class folders, you can't specify the attributes of methods having their own file. If you want a method to be Hidden, this method will have to be implemented in the classdef file.

Packages

It is also possible to group classes using packages. If you need to work with files located in folders at various locations in your computer, instead of using 'cd' 'addpath' or even copy files, consider managing them with packages. Let's look into +myPackage:

```
what myPackage;
```

```
MATLAB Code files in folder /Users/pmaal/Dropbox/Nottingham/repos/phd/
Seminars/MatlabOOP/+myPackage

functionFromPackage

Classes in folder /Users/pmaal/Dropbox/Nottingham/repos/phd/Seminars/
MatlabOOP/+myPackage

classFromPackage
```

Calling a class from the package won't work because this is only visible from within the package

```
classFromPackage;
```

Parent folder of +myPackage/ has to be on the MATLAB® path so that you can call the package and use the functions and classes within

```
a = myPackage.classFromPackage;
methods( a );
```

```
Methods for class myPackage.classFromPackage:

classFromPackage  methodA           methodB

Call "methods('handle')" for methods of myPackage.classFromPackage
 inherited from handle.
```

```
a.methodA
myPackage.functionFromPackage('Just a test');
```

```
Just a test
```

If there are packages within packages, the parent of the top-level package folder must be on the MATLAB® path!!!

Just like in python, we can now import packages and use classes defined within directly:

- import only one class from packag or
- import all classes using syntax pkg.*

```
import myPackage.classFromPackage;
import myPackage.*;
a = classFromPackage;
functionFromPackage('Just another test');
```

```
Just another test
```

Clear base workspace from all imported packages

```
clear import;
```

To see what is in current folder: code files (functions, scripts or classes), class folders, packages, mat-files:

```
what;
```

```
MATLAB Code files in the current folder /Users/pmaal/Dropbox/
Nottingham/repos/phd/Seminars/MatlabOOP

MatlabOOPrun        myClassNoHandle     polymA
SyntaxColors        myClassSyntax       polymB
audience            myCreditor          showHousemates
myAbstractClass     myTest              students
myClass             nottinghamClass
myClassEvent        nottinghamSubclass
myClassExample      polym

Classes in the current folder /Users/pmaal/Dropbox/Nottingham/repos/
phd/Seminars/MatlabOOP

haveYouSeenThisMethod

Packages in the current folder /Users/pmaal/Dropbox/Nottingham/repos/
phd/Seminars/MatlabOOP
```

```
myPackage
```

Class attributes

All classes support the attributes listed in the following. Attributes enable you to modify the behavior of class. Attribute values apply to the class defined within the classdef block.

```
classdef (Attribute1 = value1, Attribute2 = value2 ) ClassName
   ...
end
```

- **Abstract** (default = false) If specified as true, this class is an abstract class (cannot be instantiated).
- **ConstructOnLoad** (default = false) If true, MATLAB? calls the class constructor when loading an object from a MAT-file. Therefore, you must implement the constructor so it can be called with no arguments without producing an error.
- **HandleCompatible** default = false) If specified as true, this class can be used as a superclass for handle classes. All handle classes are HandleCompatible by definition.
- **Hidden** (default = false) If true, this class does not appear in the output of the superclasses or help functions.
- **Sealed** (default = false) If true, this class can not be subclassed.
- **AllowedSubclasses** List classes that can subclass this class.
- **InferiorClasses** Use this attribute to establish a precedence relationship among classes.

Class attributes are not inherited, so superclass attributes do not affect subclasses.

For attributes values that are logical true or false, class definitions can specify attribute values using expressions.

Properties attributes

Here is a partial list of the properties attributes. See documentation for complete descriptions.

```
classdef (...
        properties (Attrib1=value1, Attrib2=value2, Attrib3=val3)
        end
end
```

- **AbortSet** If true, MATLAB? does not set the property value if the new value is the same as the current value. MATLAB does not call the property set method, if one exists.
- **Abstract** If true, the property has no implementation, but a concrete subclass must redefine this property without Abstract being set to true.

- **Access** public - unrestricted access; protected - access from class or subclasses; private - access by class members only (not subclasses).
- **Constant** Set to true if you want only one value for this property in all instances of the class.
- **GetAccess** Same values as 'Access'.
- **GetObservable** If true, and it is a handle class property, then you can create listeners for access to this property.
- **Hidden** Determines if the property should be shown in a property list (e.g., Property Inspector, call to set or get, etc.).
- **NonCopyable** Indicate if property value should be copied when object is copied.
- **SetAccess** Has additional 'immutable' - property can be set only in the constructor.
- **SetObservable** If true, and it is a handle class property, then you can create listeners for access to this property.
- **Transient** If true, property value is not saved when object is saved to a file.

There are also some hidden and undocumented properties attributes such as Description or DetailedDescription. They are read as errors in your code but are perfectly usable (as of 2015a) and the error code disappears if you end to property definition by %#ok<ATUNK>

Example of private access

n.myImmutableProp can't be changed. Let's try.

```
try
    n.privateGetAccess;
catch ME
    disp(ME);
    n.getPrivAcc;
end
```

```
  MException with properties:

    identifier: 'MATLAB:class:GetProhibited'
       message: 'You cannot get the 'privateGetAccess' property of
 'nottin...'
         cause: {}
         stack: [5x1 struct]

privateGetAccess contains: You can't see me unless you call
 getPrivAcc.
```

Handle: any change on m will change n.

```
m = n;
try
    m.time = 'We can easily change property "time" because it''s
 public (Access=public)';
    fprintf(1,'n.time now contains: %s\n', n.time);
```

```
catch ME,  disp(ME);
end

n.time now contains: We can easily change property "time" because it's
 public (Access=public)

try
    m.myNewProperty = 'but not as easily myNewProperty because it''s
 read-only (SetAccess=private)';
catch ME,  disp(ME);
end

  MException with properties:

    identifier: 'MATLAB:class:SetProhibited'
       message: 'You cannot set the read-only property 'myNewProperty'
 of ...'
         cause: {}
         stack: [5x1 struct]


try
    m.set('myNewProperty', 'The method "set" is defined, and can
 normally change the private properties');
catch ME,  disp(ME);
end
fprintf(1,'%s\n',m.myNewProperty);

The method "set" is defined, and can normally change the private
 properties

try
    m.set('myImmutableProp', 'But I can''t use it on
 myImmutableProp!!!');
catch ME,  disp(ME);
end

  MException with properties:

    identifier: ''
       message: 'Don't even try to change myImmutableProp'
         cause: {0x1 cell}
         stack: [6x1 struct]


try
    m.myImmutableProp = 'So, m.myImmutableProp is not changeable,
 whatever you do.';
catch ME,  disp(ME);
end

  MException with properties:

    identifier: 'MATLAB:class:SetProhibited'
       message: 'You cannot set the read-only property
 'myImmutableProp' o...'
```

```
        cause: {}
        stack: [5x1 struct]
```

Methods attributes

Method Attribute: Abstract If (Abstract=true), the method has to be implementated in subclasses.

myAbstractMethod has to be implemented in nottinghamSubclass, otherwise it inherits the abstract class and can't be instanciated.

Abstract methods must be implemented by subclasses and as such their inclusion in a superclass acts as a kind of contract, enforcing interface consistency

```
[n1, n2] = n.myAbstractMethod('arg');
```

Method Attribute: Access, GetAccess, SetAccess Determines what code can call this method. 4 options:

- Access='public' Unrestricted Access
- Access='protected' Access from methods in class or subclasses
- Access='private' Access by class methods only
- Access={?class1,?class2,...}) Access by specified classes

GetAccess and SetAccess do the same for get and set methods, repectively.

```
try
    n.myPrivateMethod('Trying to use private method directly: fails.
\n');
catch
    n.callPrivateMethod('Calling the private method through public
 one: works.\n');
end
```

```
myPrivateMethod(obj, fromCallPrivateMethod)
```

Method Attribute: Hidden If set to true, the methods's names not included in methods(classInstance) and ismethod returns false for this method name. Ex: Open myClass

```
mc = myClass( 1 );
methods( mc )  % c2d_* not meant to be seen by user


Methods for class myClass:

a2b           b2c           c2d           myClass       one2another

Call "methods('handle')" for methods of myClass inherited from handle.
```

Method Attribute: Sealed If true, the method cannot be redefined in a subclass.

Attempting to define a method with the same name in a subclass causes an error. Ex: Open nottinghamClass

Method Attribute: Static Static methods are methods that are associated with a class as opposed to instances of that class.

Static methods are useful when you have methods are are thematically related to the class but which do not use any information particular to specific instances of that class.

```
n.myNewMethod;

I'm the new method, only available to objects of class
 nottinghamSubclass.
As a static method, I'm callable from the class itself:
 nottinghamSubclass.myNewMethod();
I can be useful: Time is 28-Oct-2015 21:47:36.

nottinghamSubclass.myNewMethod();

I'm the new method, only available to objects of class
 nottinghamSubclass.
As a static method, I'm callable from the class itself:
 nottinghamSubclass.myNewMethod();
I can be useful: Time is 28-Oct-2015 21:47:36.
```

Advanced MATLAB®: Events and listeners

Events are notices that objects broadcast in response to something that happens, such as a property value changing or a user interaction with an application program. Listeners execute functions when notified that the event of interest occurs. Use events to communicate things that happen to objects, and respond to these events by executing the listener's callback function.

The syntax is as we would guess:

```
classdef ClassName < handle
   ...
   events
      EventName
   end
   ...
end
```

Let's try the following. First, openopen myClassEvent

```
myself = myClassEvent( 'Alban', 100 );
myself.receivedMoney(10);
myCreditor1 = myCreditor('Steve',myself,1100);
myself.spendMoney(1100);
myCreditor2 = myCreditor('Chris',myself,25);
myself.spendMoney(36);
myself.spendMoney(200);
```

When using events and listeners:

- Only handle classes can define events and listeners.

- Call the handle notify method to trigger the event. The event notification broadcasts the named event to all listeners registered for this event.
- Use the handle addlistener method to associate a listener with an object that will be the source of the event.
- When adding a listener, pass a function handle for the listener callback function using a syntax such as the following:

```
addlistener(eventObject,'EventName',@functionName) % for an ordinary
addlistener(eventObject,'EventName',@Obj.methodName) % for a method o
addlistener(eventObject,'EventName',@ClassName.methodName) % for a st
addlistener(eventObject,'EventName',@PackageName.functionName) % for
```

- Listener callback functions must define at least two input arguments ? the event source object handle and the event data.
- You can modify the data passed to each listener callback by subclassing the event.EventData class.

Define and Trigger Events ('notify' method) To define an event, declare a name for the event in an events block. Trigger the event using the handle class notify method. Only classes derived from the handle class can define events.

Listen for Events ('addlistener' method) Any number of objects can listen to the StateChange event. When notify executes, MATLAB® calls all registered listener callbacks. MATLAB® passes the handle of the object generating the event and event data to the callback functions. To create a listener, use the addlistener method of the handle class.

Polymorphism

Polymorphism is the ability to treat all the children of a common parent as if they were all instances of that parent. It has two important aspects, as follows:

1. All objects that are children of a parent class must be treated collectively as if they were all instances of the parent,
2. Individually, the system must reach the specific child methods when called for.

In MATLAB®, we will use the superclass matlab.mixin.Heterogeneous

```
p = [polymA, polymB, polymA, polymA];
disp(p);                    % Collectively treated as instances of the
 parent
```

```
  1x4 heterogeneous polym (polymA, polymB) array with properties:

    propPolym
```

Use method from class polym (no polymorphism)

```
arrayfun(@(ind)ind.sealMeth, p)
```

```
ans =
     0     0     0     0
```

Use method from class polymA or B (polymorphism)

```
arrayfun(@(ind)ind.meth, p) % Individually treated as instances of the
 child
```

```
ans =

     1     2     1     1
```

```
arrayfun(@(ind)ind.a, p)
```

```
ans =

    10    20    10    10
```

MATLAB® in excruciating depth

Let's count the number of m-files, packages... in MATLAB® internals. A lot of classes, a lot of packages, a lot of weird files (such as .ver, .vcxproj, .traineddata, .policy,...)

```
Number of folders:       11503
Number of packages:      2108
Number of class folders: 5912
Number of private files: 7343
Number of .m:            28418
Number of .mat:          1671
Number of weird files:   30082
```

Let's change the desktop's title: Change the editor's title by running html/setTitleEd.

(When) When to use classes

1. When, fundamentally, you are developping a specific object (model,...)
2. When you need some constraints (some properties being char, ...)
3. When you do this kind of things a2b, b2c, c2d... successively.
4. When you want to call many small functions but don't want as many files.
5. When you are working with what feels like levels of abstraction (inheritance) in particular if you need to hsm.methodCcallingBadd methods (newClass < class1 & class2).
6. More generally, when you feel your code should be simple than what it is currently. Feeling also refered to as the KISS principle (*Keep it Simple, Silly*).
7. When you need to keep track of an event (addlistener).

8. Think about polymorphism when you need to apply different things without specifying in your script the level of abstraction (ex: in writeScript).

Overload MATLAB® functions and MATLAB® operators

Classes can implement existing functionality, such as addition, by defining a method with the same name as the existing MATLAB® function. For example, suppose you want to add two audience objects. It makes sense to add the values of the nbPplKnowObjects, nbPplKnowStructures, ... properties of each object. Works because user-defined classes have a higher precedence than built-in classes.

Open audience.m

```
a_tomorrow = audience('Paris');
a_tomorrow.nbPplKnowObjects     = 14;
disp( aud + a_tomorrow );  % same as plus(aud, a_tomorrow);
```

```
  audience with properties:

                  time: '28-Oct-2015 21:47:33'
              nbPeople: 300
                isnice: 1
                 venue: 'University of Nottingham_Paris'
      nbPplKnowObjects: 39
   nbPplKnowStructures: 85
  nbPplKnowInheritance: 20
          nbPplUsedOOP: 1
   nbPplWrittenClasses: 1
```

Question: would you overload some operators in your work?

A few more things with OOP

Seeing all the information you want

Simply overload the function 'disp' to output exactly what you want to see.

The trick to use the built-in function 'disp' within your overloaded 'disp' is to call builtin('disp', ~).

Ex: I have an object called writeScript that contains a handle of an object HTKCodeGenerator. When displaying, writeScript, the built-in disp is applied on both:

Open writeScript.

Nested functions

You can use nested function in a class' method as you're used to.

Using all variables types

Within your methods, you can use the 3 types of variables: local, persistent, global. In most cases, using properties or attributes is more natural when working within an object.

Find memory allocation for objects

Trick to get size of the objects: A handle typically takes 112 Bytes. To see how much memory it really takes, a trick is to convert into struct:

```
warning off 'MATLAB:structOnObject' % To get rid of warning message.
c_struct = struct( c );
warning on 'MATLAB:structOnObject'  % Error message back on.
disp( c );
whos( 'c','c_struct' );
```

```
  nottinghamClass with properties:

         time: '28-Oct-2015 21:48:00'
    path2logo: './notts.jpg'
         logo: [1x1 Figure]

  Name              Size            Bytes  Class              Attributes

  c                 1x1               112  nottinghamClass
  c_struct          1x1               694  struct
```

Tab completion

Tab complementation works for classes, properties and methods. Try it.

Enumeration class

Enumeration classes are limited to a list of given values. Each statement in an enumeration block is the name of an enumeration member, optionally followed by an argument list. If the enumeration class defines a constructor, MATLAB calls the constructor to create the enumerated instances.

MATLAB provides a default constructor for all enumeration classes that do not explicitly define a constructor. The default constructor creates an instance of the enumeration class: * Using no input arguments, if the enumeration member defines no input arguments. * Using the input arguments defined in the enumeration class for that member.

```
enumeration students
```

```
Enumeration members for class 'students':

    Alban
    Toby
    Dave
    Oscar
    Joel
    Steaders
```

```
a = students.Toby;
disp(a.isgoodstudent);
```

```
He's the best!
```

```
enumeration SyntaxColors


Enumeration members for class 'SyntaxColors':

    Error
    Comment
    Keyword
    String


b = SyntaxColors.Comment;
disp(b);

    Comment


disp(b.RGB);

     0     1     0
```

A few constraints on enumeration classes

- Enumeration classes are implicitly Sealed.
- Only MATLAB® can call enumeration class constructors to create the fixed set of members.
- The properties of value-based enumeration classes are immutable.

Example from my own work

Live presentation of my work. Cd to MAP_/ Open HTKCodeGenerator.

OOP Drawbacks

- Verbosity. Can be a good thing too.
- Harder to learn than doing script with functions.
- Not widely spread.

Thought experiment

For the next 1 minute, please think about how you can transform the code you are currently working on to adapt it with your own classes.

Questions?

Take home messages

- Think first about the object you work on rather than the operations you want to do on it (ie: create and use the appropriate class).

- Think of your work as building a package (+) using appropriate levels of abstraction (@).
- Complexity is your ennemy: if it makes your code easier to read/change/share/..., then make those 3 more methods.
- Matlab offers a lot of possiblities: Once you use OOP, it becomes a proper programming language with good computational abilities.

References

Mathworks documentation and webinars, Yair's book Accelerating Matlab Performance (2014) and website undocumentedmatlab.com, and a few others:

- http://www.mathworks.com/help/pdf_doc/matlab/matlab_oop.pdf
- http://fr.mathworks.com/videos/programming-with-matlab-86354.html?form_seq=conf1092&elqsid=1445531855668&potential_use=Education&country_code=GB
- http://fr.mathworks.com/help/matlab/matlab_prog/copying-objects.html
- http://fr.mathworks.com/help/matlab/matlab_oop/method-attributes.html
- http://fr.mathworks.com/help/matlab/ref/handle-class.html
- https://yagtom.googlecode.com/svn/trunk/html/objectoriented.html
- http://undocumentedmatlab.com/
- http://www.dms489.com/Concepts_book/onlinechapters/Chapter18.pdf
- https://en.wikipedia.org/wiki/Object-oriented_programming
- Best Practices for Scientific Computing, PLoS Biology http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745

Published with MATLAB® R2015a