Matlab for PhD students – Advanced Topics 6 – WS 2010/11
"Object oriented programming"

General consideration

- General ideas of object oriented programming (in any language):
    - Objects: Data that belongs together is grouped into a single object. (Same idea as for structures and cell arrays: No need to carry around several variables.)
    - Classes: Objects belong to classes. Classes define the common properties of all objects.
    - Methods: Functions are specifically designed for classes of objects. The user can use these functions without knowing how they are implemented (higher level programming).
    - Overloading: Different classes can use methods with the same name but different function, e.g. the commands `plot` or `disp` can do different things for objects of different types.
    - Inheritance: Classes can have hierarchies, in which sub-classes inherit properties and methods of super-classes.
    - Control flow: In contrast to traditional procedural programming, the sequence of steps to solve a problem is not the central point in object oriented programming. Here, objects interact in a network of methods, properties and events. Therefore, e.g. flow charts do not work very well for object oriented programming. However, it is possible to combine classical control flow with the use of objects.

- Matlab classes:
    - In Matlab, every data value is assigned to a class. (I have referred to classes as “data types” in this course, because this is the more general name, used in all kinds of languages.)
    - A class is a definition that specifies certain characteristics that all instances of the class share. These characteristics are determined by the properties and methods (and some classes also events) that define the class and the values of attributes that modify the behavior of each of these class components.
    - Class definitions describe how objects of the class are created and destroyed, what data the objects contain, and how you can manipulate this data.
    - You can create your own classes. Since the Matlab release 2008a this is possible in a relatively convenient way, making it possible to use object oriented programming techniques like in other modern languages. (Older Matlab versions required very complicated things to create objects. Their use is strongly discouraged!)
    - The User Guide in the Matlab help provides a very good introduction and detailed explanations about object oriented programming in Matlab.

- Objects:
    - A class is like a template for the creation of a specific instance of the class, which is called an object. This object contains actual data for a particular entity that is represented by the class.

- Objects are not just passive data containers. Objects actively manage the data contained by allowing only certain operations to be performed, by hiding data that does not need to be public, and by preventing external clients from misusing data by performing operations for which the object was not designed. Objects can even control what happens when they are destroyed.

- Class hierarchies:
  - It sometimes makes sense to define a new class in terms of existing classes. This enables you to reuse the designs and techniques in a new class that represents a similar entity.
  - You accomplish this reuse by creating a sub-class. A sub-class defines objects that are a subset of those defined by the super-class.
  - A sub-class inherits properties and methods of the super-class.
  - A sub-class is more specific than its super-class and might add new properties, methods, and events to those inherited from the super-class.
  - A big advantage of inheritance is that code is re-used very efficiently. This reduces the efforts to write, test and modify similar code several times for slightly different applications. E.g. it is easy to apply global changes.

- Overloading:
  - It is convenient for the user to use well-known names of functions (e.g. plot, length, disp…) also for your new objects.
  - However, these functions often need to do different (or at least more specific) things than the standard functions defined for double precision floating point matrices.
  - Overloading means that a function name that is already defined for other data types is re-used as method of a new class. Matlab will distinguish between the previously existing (maybe built-in) function and the new class method depending on the input arguments passed to the function.

- Advantages of objects compared to structures in Matlab:
  - Objects have their own methods.
  - Users cannot accidentally misspell a field name without getting an error. In a structure, a misspelled field name simply adds a new field, whereas an object returns an error if the property is not defined for objects of a class.
  - A class can validate individual field values when assigned, including class (data type) or value.
  - A class can restrict access to fields, for example, allowing a particular field to be read, but not changed.
  - A class is easy to identify. A class has a name so that you can identify objects with the `whos` and `class` functions and the Workspace browser. (A variable is not just some structure, but an instance of your particular class.) The class name makes it easy to refer to records with a meaningful name.
  - A class is easy to reuse. Once you have defined the class, you can easily extend it with subclasses that add new properties.

Nomenclature
Matlab classes use the following words to describe different parts of a class definition and related concepts:

- Class definition: Description of what is common to every instance of a class.
- Properties: Data storage for class instances
- Methods: Special functions that implement operations that are usually performed only on instances of the class
- Events: Messages that are defined by classes and broadcast by class instances when some specific actions occur
- Attributes: Values that modify the behavior of properties, methods, events and classes
- Listeners: Objects that respond to a specific event by executing a callback function when the event notice is broadcast
- Objects: Instances of classes which contain actual data values stored in the objects' properties
- Sub-classes: Classes that are derived from other classes and that inherit the methods, properties and events from those classes (sub-classes facilitate the reuse of code defined in the super-class from which they are derived)
- Super-classes: Classes that are used as a basis for the creation of more specifically defined classes (i.e., sub-classes)
- Packages: Folders that define a scope for class and function naming (will not be discussed in this course).

Definition of a class

- Header:
  - A function definition file must start with the keyword `classdef`
  - The header line defines the name of the new class, e.g.
    ```
    classdef TimeSig
    ```
  - The file must have the same name as the class, e.g. TimeSig.m
  - Sub-classes of existing classes are specified by < e.g.
    ```
    classdef SoundSig < TimeSig
    ```
  - The header line should be followed by comment text about the class. This text can be displayed as help text.
  - It is possible to specify class attributes in the classdef header. They modify the behavior of the class, e.g. a class with attribute Hidden=true does not appear in the output of Matlab commands or tools that display class names. (For more information see class attributes in the help).

- Definition of properties:
  - The first block of code in the object definition file defines which data the objects of the class contain.
  - The block starts with the keyword `properties` and ends with the keyword `end`
  - The names of the properties (corresponding to the names of the fields of a structure) are specified in this block.
  - Default values can be assigned to the properties in the function definition block.
  - Example:
    ```
    properties
          vData             % measurement data
    ```

```
        SampRate        % sampling rate
        sName           % name of the time series
        TimeUnit='s';   % unit used for time axis,
                        % default value is ,s' (seconds)
    end
```

- <u>Property attributes:</u> Attributes enable you to modify the behavior of properties.
- Matlab supports several attributes. See help text for property attributes for a complete list. Some important attributes are:

```
Abstract      % Logical value. Default: false
              % If true, the property has no implementation,
              % but a concrete subclass must redefine this
              % property without Abstract being set to true.
SetAccess     % Enumeration value. Default: public
              % Restricts the access from where an attribute
              % can be set. Possible values:
              % public – unrestricted access
              % protected – access from class or derived
              %        classes
              % private – access by class members only
GetAccess     % Same as SetAcces, but for reading property
              % values.
Hidden        % Logical value. Default: false
              % Determines whether the property should be
              % shown in a property list
Dependent     % see below
```

- Attribute values apply to all properties defined within the properties block that specifies the non-default values.

```
properties (SetAccess=protected)
   ImportantNumbers=[pi eps i];     % define a protected
                               % property and set default values
end
```

- <u>Dependent properties:</u> The property attribute `Dependent` is probably the most important one. The value of a dependent property cannot be assigned directly and is not stored in the object. Instead, it is calculated by a class method whenever the dependent property is requested (e.g. to display the object or to calculate with the property).
  - This is a convenient way to keep several properties up-to-date when they depend on a property that changes values when you work with the object.
  - Dependent properties are defined by a block started with the keyword `properties` with the logical attribute `Dependent` set to `true`.

    ```
    Properties (Dependent=true)
    ```
  - Example:

    ```
    properties (Dependent=true)
          vTimeAxis  % time axis in seconds, compute from
                     % sampling rate and length of the data
                     % vector with method get.vTimeAxis(obj)
    end
    ```
  - If a property is marked to be dependent, an error will be issued if a user tries to set it directly. If you want to have a specific error message, you can define also set property method (see below) for the dependent property.

- Definition of methods:
  - The second mandatory block of code in the object definition file defines the functions used to work with objects of the class.
  - The block starts with the keyword `methods` and ends with the keyword `end`
  - Methods are defined as functions which have objects of the class as input and / or output arguments. The functions can have any number of additional input and output arguments. It is possible to use `varargin` and `varargout`.

  - Ordinary methods: Class methods can perform any operations you can express with Matlab code. E.g. you can create and destroy objects, read and write object properties, compare objects, display them as text or graphics, perform calculations based on property values…

  - Constructor method: Each class should at least have a constructor method for creating new objects of the class. The constructor method has the same name as the class.
  - A constructor method can have any number of input arguments and returns an object of the class.
  - The constructor method sets the properties to the values specified in the function call. Dependent properties are usually not set by the constructor method but by specific get property functions (see below).
  - Example:

```
methods
      function obj = TimeSig(sFileName, sSigName)
            % load signal and sampling rate
            [vX fs]       = wavread(sFileName);
            % store in object properties
            obj.vData     = vX;
            obj.SampRate  = fs;
            obj.sName     = sSigName;
      end

      % Usually more functions will follow here…
end
```

  - A TimeSig object can be created e.g. with
    `oSig1 = TimeSig(‚bill_sig1_8kHz.wav’, ‚Signal 1’);`

  - Destructor methods: For many applications it also makes sense to specify a method to destroy an object, usually a function `delete(object)`. Since Matlab will not delete the data contained in an object when the name of the object is used for a variable of a different type, the memory can become overcrowded if you do not use a destructor to clear the memory.

  - Set property methods: A property can be restricted to data of specific data types (classes) or a specific range of values by defining a `set` method for this specific property.
  - A set property function should get (at least) the object and the value for the property as input and return the changed object as output.
  - The name of the function must be `set.PropertyName`

- In the body of the function the desired value is checked to be in the allowed range of values. If it is in the allowed range, the property is set to the new value. Otherwise, an error should be issued.
- The set property function is called every time the user attempts to change the value of the property, including the creation of the object.
- Example:

```
function obj = set.TimeUnit(obj,timeunit)
      if ~ischar(timeunit)  % wrong class
            error('TimeUnit must be a string!')
      elseif ~(strcmpi(timeunit,'ms') ||...
            strcmpi(timeunit,'s') || ...
            strcmpi(timeunit,'min') || ...
            strcmpi(timeunit,'h'))
            % wrong values
                  error(,Unknown value for TimeUnit!')
      else        % set property to new value
                  obj.TimeUnit=timeunit;
      end
end
```

- The value of the property TimeUnit can be changed e.g. with `oSig1.TimeUnit='ms';`

- Get property methods: The value of a dependent property is calculated every time it is needed. This is done by a function `get.PropertyName`
- Get property methods expect (at least) the object as input and give back the value of the dependent property.
- Example:

```
function vTimeAxis = get.vTimeAxis(obj)
    vTimeAxis = [0:length(obj)-1]/obj.SampRate;
end
```

- Overloading methods: You can specify methods for your objects which have the same name as built-in functions (or other Matlab functions), but which are specifically designed for the class.
- It is even possible to overload operators like + by defining a method `plus(a,b)` (see implementing operators for your class in the help).
- Matlab will determine based on the input and output arguments whether to look for the function in the class definition file or on the Matlab search path.
- Example:

```
% class method to compute length (number of time points)
% note that this is different from the Matlab built-in
% length function
function Npoints = length(obj)
      Npoints = size(obj.vData, 1);
End
```

- Method attributes: Methods also have attributes, which modify the behavior of the method – very much like it is the case for properties. For a complete list see help method attributes. Some important attributes are:

```
Abstract      % Logical value. Default: false
              % If true, the method has no implementation,
              % The method has a syntax line that can include
              % arguments which subclasses use when
```

```
                          % implementing the method.
    Access                % Enumeration value. Default: public
                          % Determines what code can call the method.
                          % Possible values:
                          % public – unrestricted access
                          % protected – access from class or sub-classes
                          % private – access by class members only
    Hidden                % Logical value. Default: false
                          % When false, the method name shows in the list
                          % of methods displayed using the methods or
                          % methodsview commands. If set to true, the
                          % method name is not included in these listings
```

  - Method values apply to all methods defined within the methods block that specifies the non-default values.

```
methods (Access = 'private') % Access by class members only
      function m = CalcModulus(td)
         % Over-simplified calculation of Elastic Modulus
         ind = find(td.Strain > 0); % Find nonzero strain
         m = mean(td.Stress(ind)./td.Strain(ind));
      end % CalcModulus
end
```

Work with your class

- Creation of an object: To create an instance of your class, you need to give it a name and to call the constructor method, e.g.
  `oSig1 = TimeSig('bill_sig1_8kHz.wav', 'Signal 1');`
- Access public property values: You can assign values to public properties (properties which have the default attribute value `SetAccess=public`) and read values of public properties (properties which have the default attribute value `GetAccess=public`) by using the dot notation that is also used for structures, e.g.
  ```
  oSig1.TimeUnit='ms';
  a=oSig1.vTimeAxis;
  ```
- Use public methods: You can call public methods (methods which have the default attribute value `access=public`) from your programs to work with your objects (e.g. perform calculations based on property values, compare objects, manipulate objects, create and destroy objects…)

Creating class hierarchies

- Designing class hierarchies: Before you start to write code you should think about all the objects you want to organize. Objects represent real world concepts or things. You should group them according to their similarities and differences in properties (“what and how are the objects?”) and in methods (“what can the objects do and what can be done to the objects?”)
  - Identify a super-class: Super-classes define the properties and methods that are common to all objects.
  - Identify sub-classes: Sub-classes inherit the data and actions from the super-class. In addition, they contain properties and / or methods that are unique to their particular purposes. A sub-class object is a super-class object, but it is a specialized one.

- Example: Represent all members of a university.

  - All members have common properties like date of birth, sex and email address. (Even though each member of the university has a different email address, all have the property to have one.)
  - All members have common methods, e.g. they can enter the university, leave the university, eat mensa food, borrow books in the library etc.
  - However, there are some fundamental differences between students and employees.
  - Students have additional properties like number of semesters studied and matriculation number.
  - Students have additional methods like taking courses, writing exams, or paying administration fees.
  - Employees, however, have additional properties like office phone number, office room number.
  - An additional method of all employees is to receive wages.

- Advantages of class hierarchies:
  - You avoid duplicating code that can be common to all classes.
  - You can add or change subclasses at any time without modifying the super-class or affecting other derived classes.
  - If the super-class changes (for example, all members of the university are assigned a Stud.IP account), then the sub-class automatically picks up these changes.

- Definition of subclasses:
  - To define a class that is a subclass of another class, add the super-class to the classdef line after a < character:

```
classdef classname < superclassname
```

- Inheritance of properties:
  - Usually, a sub-class inherits all properties of its super-classes.
  - You can re-define super-class properties in a sub-class object, if the value of the super-class property `Abstract` attribute is `true.` In this case, the super-class is just requesting that you define a concrete version of this property to ensure a consistent interface.
  - You also can re-define super-class properties in a sub-class object, if the values of the super-class property `SetAccess` and `GetAccess` attributes are `private.` In this case, only the super-class can access the `private` property, so the sub-class is free to re-implement it in any way.

- Inheritance of methods:
  - There are two ways how sub-classes can inherit methods of super-classes:
  - Interface inheritance: Abstract super-classes define the methods and properties that you must implement in the subclasses, but do not provide an implementation, in contrast to implementation inheritance. Abstract super-classes provide a common interface of methods of their sub-classes, but their implementation can be individually different.
  - Implementation inheritance: A method is implemented in the super-class and can be used as it is from within the sub-class without duplicating the code.

- Extending super-class methods: Sub-class methods can call super-class methods of the same name by calling `methodname@superclassname`. This fact enables you to extend a super-class method in a subclass without completely redefining the super-class method.

```
classdef sub < super
   methods
      function foo(obj)
         % preprocessing steps
         foo@super(obj); % Call superclass foo method
        % postprocessing steps
      end
   end
end
```

- Completing super-class methods: A super-class method can define a process that executes in a series of steps using a protected method for each step (`Access` attribute set to `protected`). Subclasses can then create their own versions of the protected methods that implement the individual steps in the process.

```
classdef super
   methods
      function foo(obj)
         step1(obj)
         step2(obj)
         step3(obj)
      end
   end
   methods (Access = protected)
      function step1(obj)
         % superclass version
      end
      ...
   end
end
```

The subclass does not re-implement the `foo` method, it re-implements only the methods that carry out the series of steps (`step1(obj), step2(obj), step3(obj)`). That is, the sub-class can specialize the actions taken by each step, but does not control the order of the steps in the process. When you pass a sub-class object to the super-class `foo` method, Matlab calls the subclass step methods because of the dispatching rules.

```
classdef sub < super
   ...
   methods (Access = protected)
      function step1(obj)
         % subclass version
      end
      ...
   end
end
```

- Subclassing multiple classes:
  - When inheriting from multiple classes, use the `&` character to indicate the combination of the super-classes:

```
classdef classname < super1 & super2
```

    - When you create a subclass derived from multiple classes, the subclass inherits the properties, methods, and events defined by all specified superclasses.
    - If more than one superclass defines a property, method, or event having the same name, there must be an unambiguous resolution to the multiple definitions. You cannot derive a subclass from any two or more classes that define incompatible class members. (See subclassing multple classes in the Matlab help.)

- <u>Some rules:</u>
    - Methods defined in the super-class can operate on sub-class objects.
    - Methods defined in the sub-class cannot operate on super-class objects.
    - A sub-class object is a super-class object. Therefore the function `isa` will return `true` when called with the name of a super-class of an object. However, the function `class` will return the most specific class the object belongs to.

Handle classes and value classes

- Matlab supports two kinds of classes — handle classes and value classes. The kind of class you use depends on the desired behavior of the class instances and what features you want to use.

- <u>Value classes</u>
    - Use value classes to represent entities that do not need to be unique, like numeric values.
    - For example, use a value class to implement a polynomial data type. You can copy a polynomial object and then modify its coefficients to make a different polynomial without affecting the original polynomial.
    - Instances of value classes behave like structures.
    - Value objects are always associated with one workspace or temporary variable and go out of scope when that variable goes out of scope or is cleared. So when a value object is passed to a function it is copied to the function workspace.
    - The built-in classes for numeric values (e.g. double, int32…) are examples for value classes.
    - Value classes are the standard classes that are created with a `classdef` file.

- <u>Handle classes</u>
    - Use a handle class when you want to create a reference to the data contained in an object of the class, and do not want copies of the object to make copies of the object data. So each object of a handle class is unique.
    - For example, use a handle class to implement an object that contains information for a phone book entry. Multiple application programs can access a particular phone book entry, but there can be only one set of underlying data.
    - The reference behavior of handles enables these classes to support features like events, listeners, and dynamic properties.
    - When you copy a handle object, Matlab copies the handle, but not the

data stored in the object properties. The copy refers to the same data as the original handle. If you change a property value on the original object, the copied object reflects the same change.
  - When a handle object is passed to a function as input argument, only the handle is copied to the function workspace. The function can change the underlying data of the object, without having the object as output argument(!).
  - A creator function of a handle object does not return the object but a handle, a unique identifier pointing to the object.
  - To create a handle class you must indicate `< handle` in the header line of the `classdef` file:
    ```
    classdef myClass < handle
    ```
  - `handle` is an abstract super-class (meaning that it is not possible to create an instance of this class directly).
  - Handle objects inherit methods from the `handle` class, e.g.
```
delete(H)       % destroys a handle object
b=isvalid(H)    % true if H is a valid handle
```
  - Properties of handle classes are usually accessed by `set` and `get` methods.
  - Handle graphics are an example for Matlab built-in handle classes (will be discussed next lecture).

- <u>Events and listeners in handle classes</u>
  - <u>Events:</u> Events are notices that handle object broadcasts in response to something that happens, such as a property value changing or a user interaction with an application program. You can use events to report things that happen in your program to other objects, which then can respond to these events by executing the listener's callback function.
  - <u>Definition and triggering of events:</u> Events are defined by a `events` … `end` block between the `properties` and `methods` blocks in the definition of handle classes. Events are triggered from within methods, usually using the function `notify`.
```
classdef SimpleEventClass < handle
% Must be a subclass of handle
   properties
      Prop1 = 0;
   end
   events
      Overflow
   end
   methods
      function set.Prop1(obj,value)
         orgvalue = obj.Prop1;
         obj.Prop1 = value;
            if (obj.Prop1 > 10)
            % Trigger the event using custom event data
   notify(obj,'Overflow',SpecialEventDataClass(orgvalue));
            end
      end
   end
end
```
  - <u>Listeners:</u> Listeners execute functions when notification of the event of interest occurs. The addlistener method of the handle class is used to

create a listener. You also need to define a callback function for the listener to execute when the event is triggered.

```
function sec = setupSEC
   % Create an object and attach the listener
   sec = SimpleEventClass;
   addlistener(sec,'Overflow',@overflowHandler)
   % Define the listener callback function
   function overflowHandler(eventSrc,eventData)
      disp('The value of Prop1 is overflowing!')
      disp(['It''s value was: 'num2str(eventData.OrgValue)])
      disp(['It''s current value: 'num2str(eventSrc.Prop1)])
   end
end
```

Homework:

- Think about your own projects: Would it be useful to define objects with specific properties and methods (and maybe also events)? Can you define super-classes and sub-classes?