MATLAB Classes (2)

Set/Get Methods

- These are methods used to access the properties: `set` for assignment, and `get` for queries.

- We need to code such functions only when we want to do some additional processing (such as argument checking or some additional computation) during such accesses.

  - Example: For class "Fraction", we may want to ensure that the denominator is not zero at an assignment.

Set/Get Methods

- Using `set` functions:
  - Syntax of `set` function header:

    ```
    obj = set.PropertyName(obj, value)
    ```

  - Syntax of the calling statement:

    ```
    object.PropertyName = value
    ```

- Using `get` functions:
  - Syntax of `get` function header:

    ```
    value = get.PropertyName(obj)
    ```

  - Syntax of the calling statement:

    ```
    object.PropertyName
    ```

Set/Get Example: Fraction

```
classdef Fraction
  …
  methods
    …
    function v = get.den(f) % unnecessary here
      v = f.den;
    end
    function f = set.den(f, d)
      if isscalar(d) && d ~= 0
        f.den = d;
      else
        error('Input error!');
      end
    end
    …
  end
end
```

Access Control

- Two main types of access controls (for properties):
  - `GetAccess`: public (default), protected, private
  - `SetAccess`: public (default), protected, private, immutable
- Put properties with different access control in separate property blocks. Syntax (example):

```
properties (SetAccess = private)
   list of properties
end
```

Function Overloading

- As mentioned before, MATLAB does not provide function overloading (multiple functions of the same name and different argument lists).
- For a function name, you can only have one function.
- The desired effect of overloading (different processing depending on different argument lists) has to be handled in the function by checking the number/types of arguments.
  - Use `nargin` and `nargout` to check numbers of arguments.
  - Use `isa` (or other `is*` functions) to check the types of input arguments.

Object Arrays

- Normally we want to be able to handle arrays of objects of the same class.
- Normal array operations still work, such as transpose, concatenation, sub-array, etc.
  - Many such operations / functions can be overloaded as well. Do so with care though.
- Some expressions used in structure arrays, such as the syntax `[var.field]`, are applicable to object arrays and useful for elementwise operations.
- The methods of the class need to be able to handle arrays.

Object Arrays Example: Fraction

```
function obj = Fraction(n, d)
  if nargin == 0
    obj.num = 0;  obj.den = 1;
  elseif nargin == 1
    obj(1,numel(n)) = Fraction; % pre-allocation
    for ii = 1:numel(n)
      obj(ii).num = n(ii);
      obj(ii).den = 1;
    end
    obj = reshape(obj, size(n));
  elseif all(size(n) == size(d)) && all(d(:) ~= 0)
    obj(1,numel(n)) = Fraction; % pre-allocation
    for ii = 1:numel(n);
      obj(ii).num = n(ii);
      obj(ii).den = d(ii);
    end
    obj = reshape(obj, size(n));
  else
    error('Input error!');
  end
end
```

Object Arrays Example: Fraction

```
function r = value(f)
  r = reshape([f.num] ./ [f.den], size(f));
end
function f = plus(f1, f2)
  if isscalar(f1)
    f = reshape(Fraction([f1.num]*[f2.den] + ...
       [f2.num]*[f1.den], [f1.den]*[f2.den]), size(f2));
  elseif isscalar(f2)
    f = reshape(Fraction([f1.num]*[f2.den] + ...
       [f2.num]*[f1.den], [f1.den]*[f2.den]), size(f1));
  elseif all(size(f1) == size(f2))
    f = reshape(Fraction([f1.num].*[f2.den] + ...
       [f2.num].*[f1.den], [f1.den].*[f2.den]), ...
       size(f1));
  end
end
function disp(f)
  fprintf('%d/%d\n', [[f.num]; [f.den]]);
end
```

Additional Topics

Several topics that we do not cover:

- Inheritance
- More class/property/method attributes (such as `abstract`, `static`, etc.)
- Handle classes
- Events, messaging, callbacks.
- Saving and loading objects.
- More …