```
classdef MyFraction < handle
    properties (SetAccess = private)
        nominator = []
        denominator = []
    end
    methods
        function obj = MyFraction(nom, denom)
            obj.setNominator(nom);
            obj.setDenominator(denom);
        end
        function setNominator(obj, nominator)
            if mod(nominator,1) ~=0
```

an introduction to

object-oriented programming in MATLAB

Jann Paul Mattern

```
            out = obj.nominator/obj.denominator;
        end
        function disp(obj)
            fprintf('   %d/%d\n\n', obj.nominator, obj.denominator)
        end
        function out = plus(obj,obj2)
            if isnumeric(obj)
                out = MyFraction(obj*obj2.denominator+obj2.nominator,obj2.denominator);
            elseif isnumeric(obj2)
                out = MyFraction(obj2*obj.denominator+obj.nominator,obj.denominator);
            elseif isa(obj, 'MyFraction') && isa(obj2, 'MyFraction')
                out = MyFraction(obj.nominator*obj2.denominator+obj2.nominator*obj.denominator, ...
                obj.denominator*obj2.denominator);
            else
                error('Cannot add %s to %s.', class(obj), class(obj2))
            end
```

```
classdef MyFraction < handle
    properties (SetAccess = private)
        nominator = []
        denominator = []
    end
    methods
        function obj = MyFraction(nom, denom)
            obj.setNominator(nom);
            obj.setDenominator(denom);
        end
        function setNominator(obj, nominator)
            if mod(nominator,1) ~=0
                error('Nominator must be an integer.')
            end
            obj.nominator = nominator;
            obj.reduce()
        end
        function setDenominator(obj, denom)
            if denom == 0 || mod(denom,1) ~=0
                error('Denominator must be an integer and nc
            end
            obj.denominator = denom;
            obj.reduce()
        end
        function out = toDouble(obj)
            out = obj.nominator/obj.denominator;
        end
        function disp(obj)
            fprintf('   %d/%d\n\n', obj.nominator, obj.denon
        end
        function out = plus(obj,obj2)
            if isnumeric(obj)
                out = MyFraction(obj*obj2.denominator+obj2.n
            elseif isnumeric(obj2)
                out = MyFraction(obj2*obj.denominator+obj.no
```

In the next 20 (or so) slides we will create a simple matlab class and expand it step by step.

```
matlab code will appear
in boxes like this
```

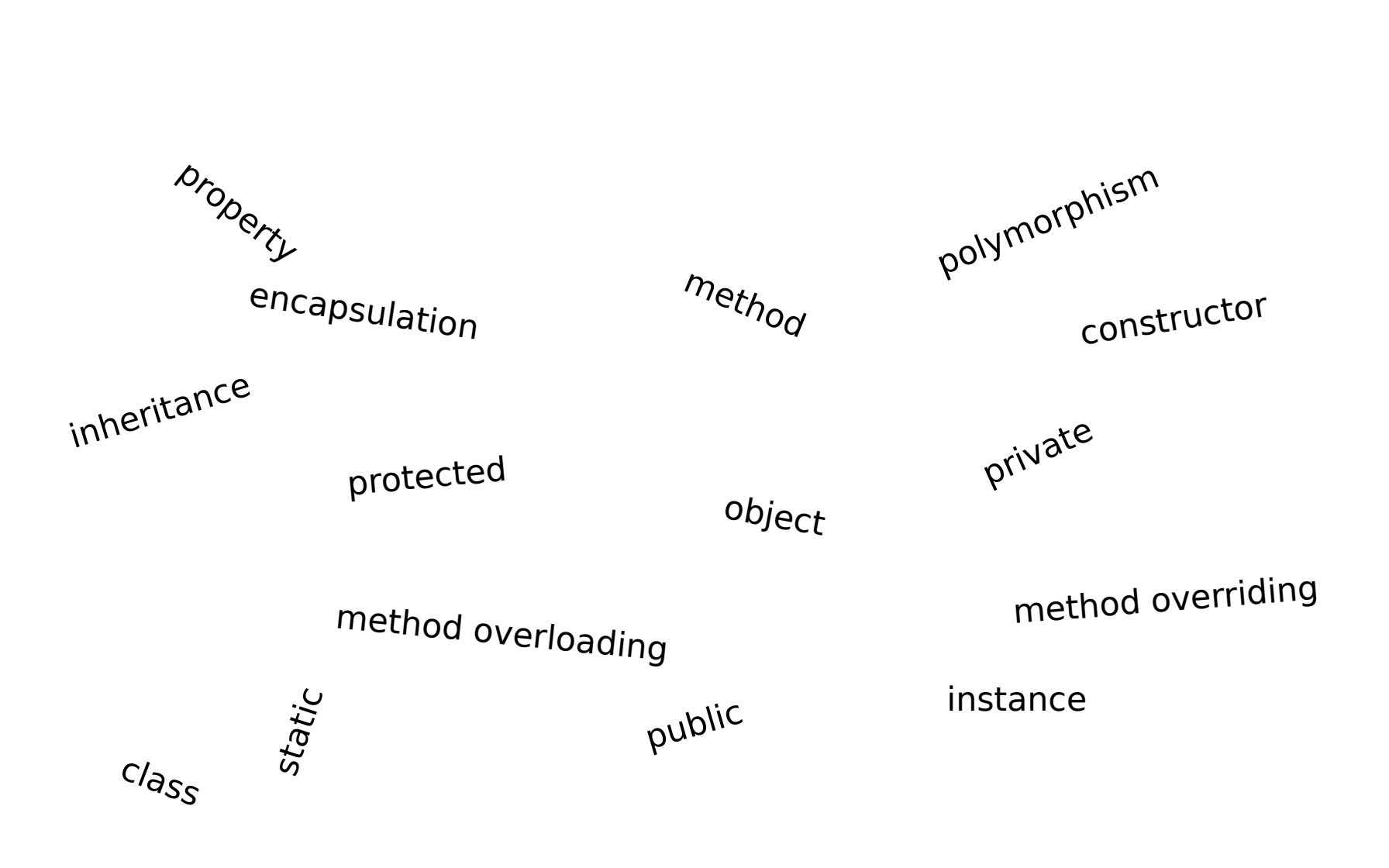
```
>> 1 + 1
ans =
    2
```

The ">>" refers to matlab command line input

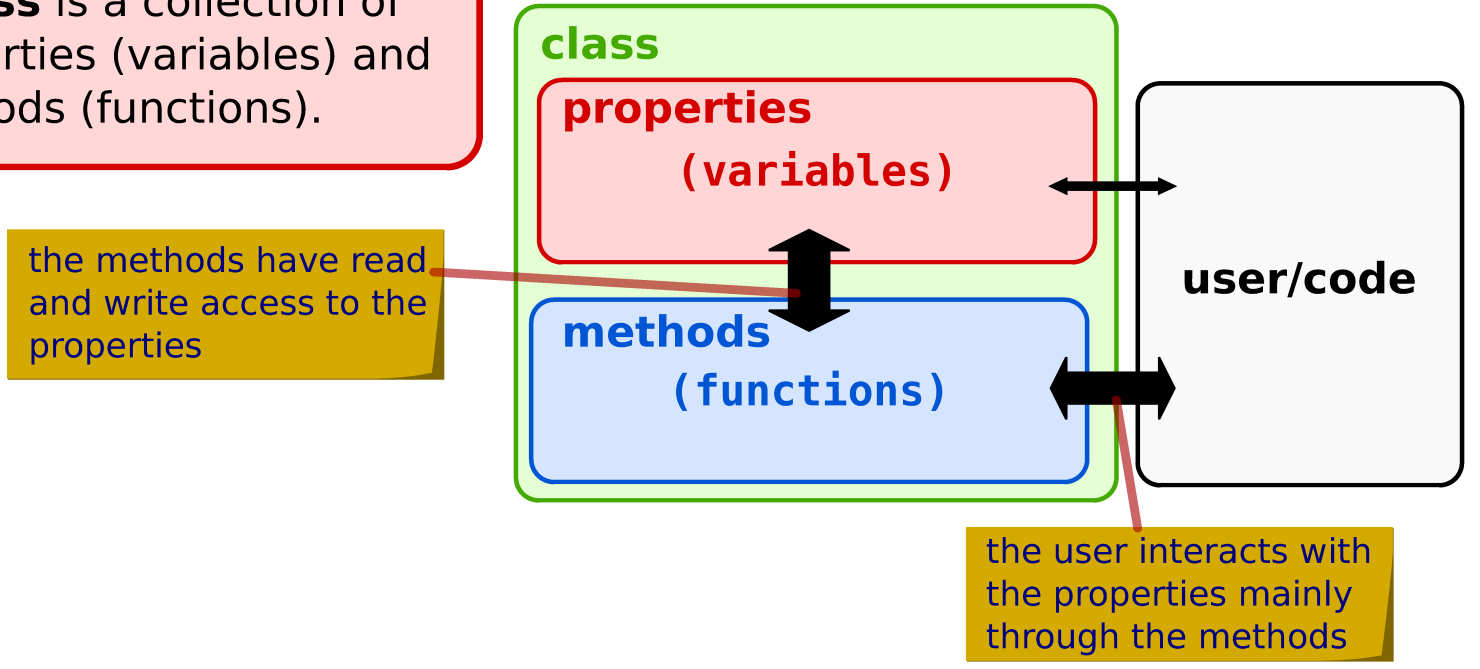
Object-oriented **terminology** such as "class" will appear in boxes like this.

Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. Many modern programming languages now support OOP.

- Wikipedia


property
polymorphism
encapsulation
method
constructor
inheritance
protected
private
object
method overriding
method overloading
static
public
instance
class

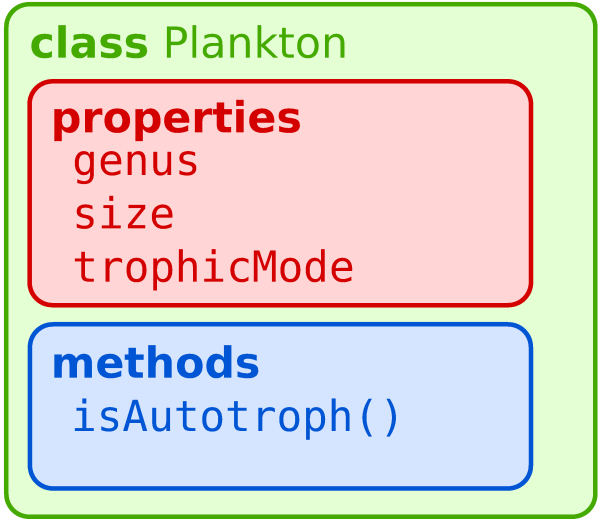
A **class** is a collection of properties (variables) and methods (functions).



A class is a collection of properties (variables) and methods (functions).

A class can be thought of as a blueprint, that describes the characteristics of something.

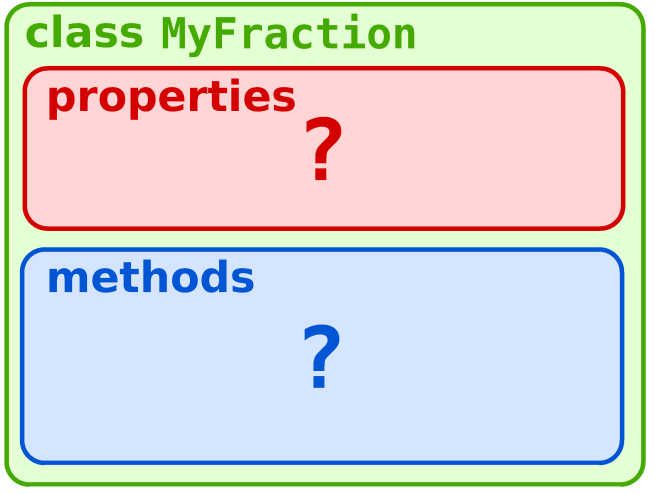
In our code (at runtime) we create **instances** of classes called **objects**.



object (copepod is an instance of Plankton)

```
>> copepod = Plankton();
>> copepod.size = 1;
```

In the following we will design step-by-step a simple matlab class that allows us to use fractions in a convenient way.



Let's say we want to be able to create our own fractions, add or subtract them and reduce them if necessary:

"
```
>> a = 1/2;
>> b = 3/4;
>> a + b
ans =
    5/4
```
"

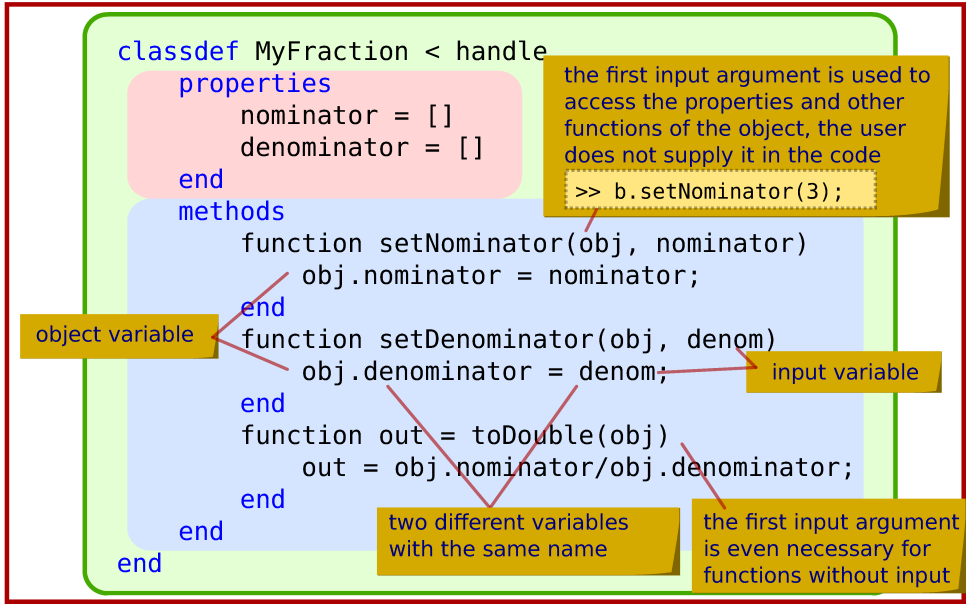
Let's start out simple:

class MyFraction

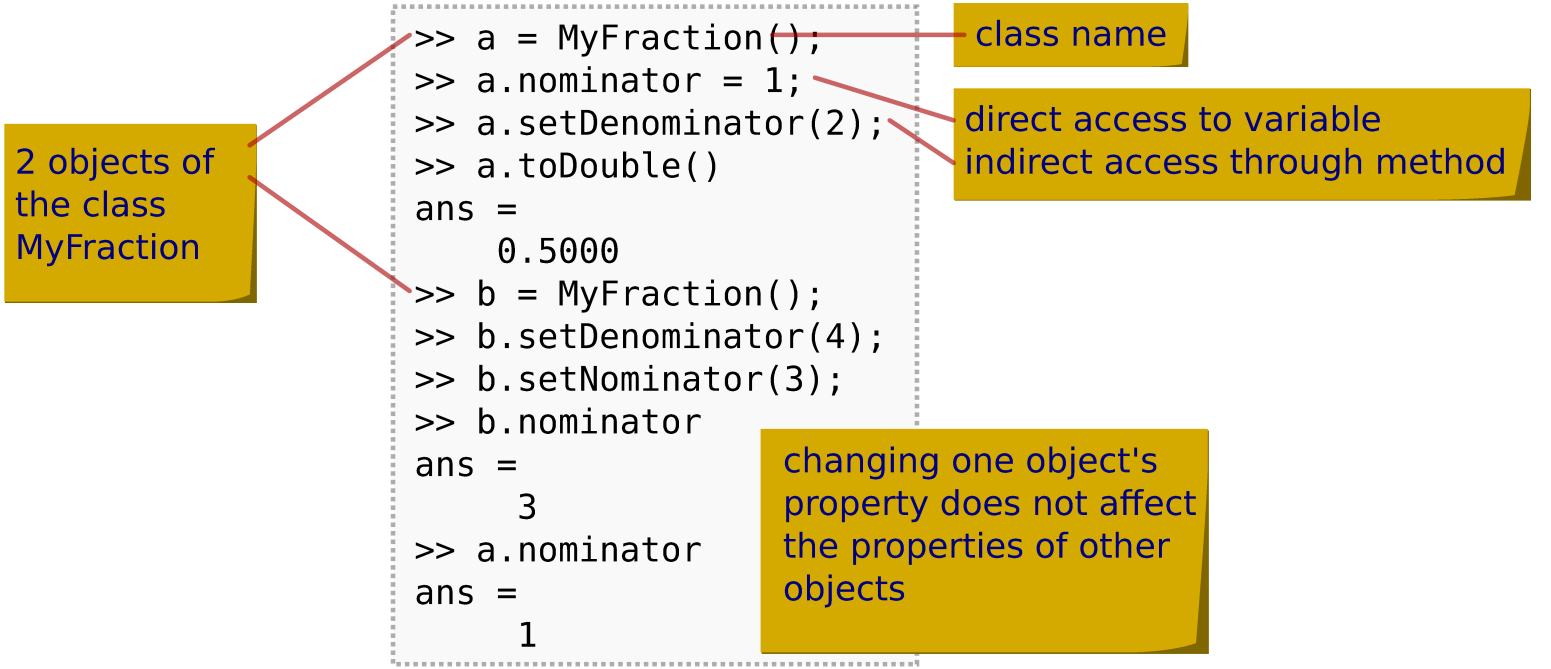
properties
- nominator
- denominator

methods
- setNominator(nom)
- setDenominator(denom)
- out = toDouble()

Our first MyFraction.m file:



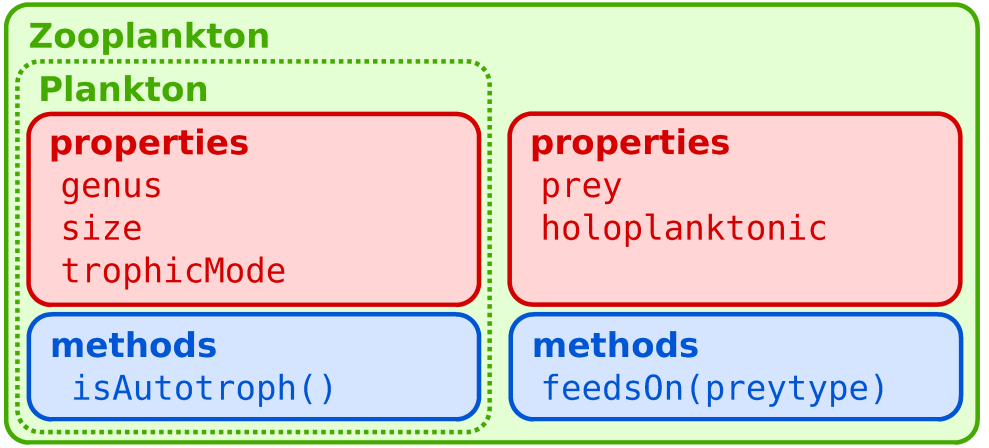
Once we start matlab, we can use MyFraction for some simple stuff:



Inheritance is a fundamental concept of object-oriented programming: A class may inherit the properties and methods of another class.

The class that inherits is called **subclass**, the class a it inherits from is called **superclass**. The subclass contains all the properties and methods of its superclass.

Each subclass can alter its inherited characteristics and add its own.



name of subclass

name of superclass

```
classdef Zooplankton < Plankton
    [...]
end
```

```
classdef MyFraction < handle
```

handle is the name of a superclass matlab provides, it contains some default methods

MyFraction is useless without values for its properties, nominator and denominator, yet setting those values is cumbersome:

```
>> b = MyFraction();
>> b.setDenominator(4);
>> b.setNominator(3);
```

Now we must(!) use the new constructor to create MyFraction objects:

```
>> b = MyFraction(3,4);
```

Constructors are special functions used to intialize the properties of an object.

Let's create a constructor for MyFraction:

```
classdef MyFraction < handle
    properties
        nominator = []
        denominator = []
    end
    methods
        function obj = MyFraction(nom, denom)
            obj.setNominator(nom);
            obj.setDenominator(denom);
        end
        [...]
    end
end
```

constructor:
- has same name as class
- obj is only output argument
- obj is not an input argument

This tutorial will require you to update MyFraction frequently. Required code updates are indicated by this symbol: **→STEP2**

1) uncomment the code for the next step in MyFraction.m and comment the rest

```
...
%%% STEP 2: constructor

% classdef MyFraction < handle
%     ...
%     ...
% end

%%% STEP 1: basic layout

classdef MyFraction < handle
    ...
    ...
end
```

only one classdef block should be uncommented

```
...
%%% STEP 2: constructor

classdef MyFraction < handle
    ...
    ...
end

%%% STEP 1: basic layout

% classdef MyFraction < handle
%     ...
%     ...
% end
```

to comment: Ctrl + R
to uncomment: Ctrl + T

2) save MyFraction.m

3) type "clear classes" in the matlab console

```
>> clear classes
```

Now you are good to try out the updated version of MyFraction

What happens, if we try to execute:

```
>> a.denominator = [1 2 3];
```

```
>> a.nominator = 'hello';
```

```
>> a.setNominator(0.5);
```

```
>> a.setDenominator(0);
```

All of the above statements do not result in an immediate error but because the values get saved as object variables, errors might occur later.
Solution: forbid direct access:

```
>> a.denominator = 0
??? Setting the 'denominator' property of the 'MyFraction'
class is not allowed.
```

and limit indirect access:

```
>> a.setNominator(0.4);
??? Error using ==> MyFraction>MyFraction.setNominator at 15
Nominator must be an integer.
```

```
classdef MyFraction < handle
    properties (SetAccess = private)
        nominator = []
        denominator = []
    end
    methods
        [...]
        function setNominator(obj, nominator)
            if mod(nominator,1) ~=0
                error('Nominator must be an integer.')
            end
            obj.nominator = nominator;
        end
        function setDenominator(obj, denom)
            if denom == 0 || mod(denom,1) ~=0
                error('Denominator must be an integer and not equal to zero.')
            end
            obj.denominator = denom;
        end
  [...]
end
```

restrict set-access to properties to "private", get-access remains "public"

create checks and error messages for invalid inputs

STEP3

Sometimes it is useful to limit the access to methods, too, e.g. to create functions that the user cannot call.
Example:

```
function reduce(...)
```

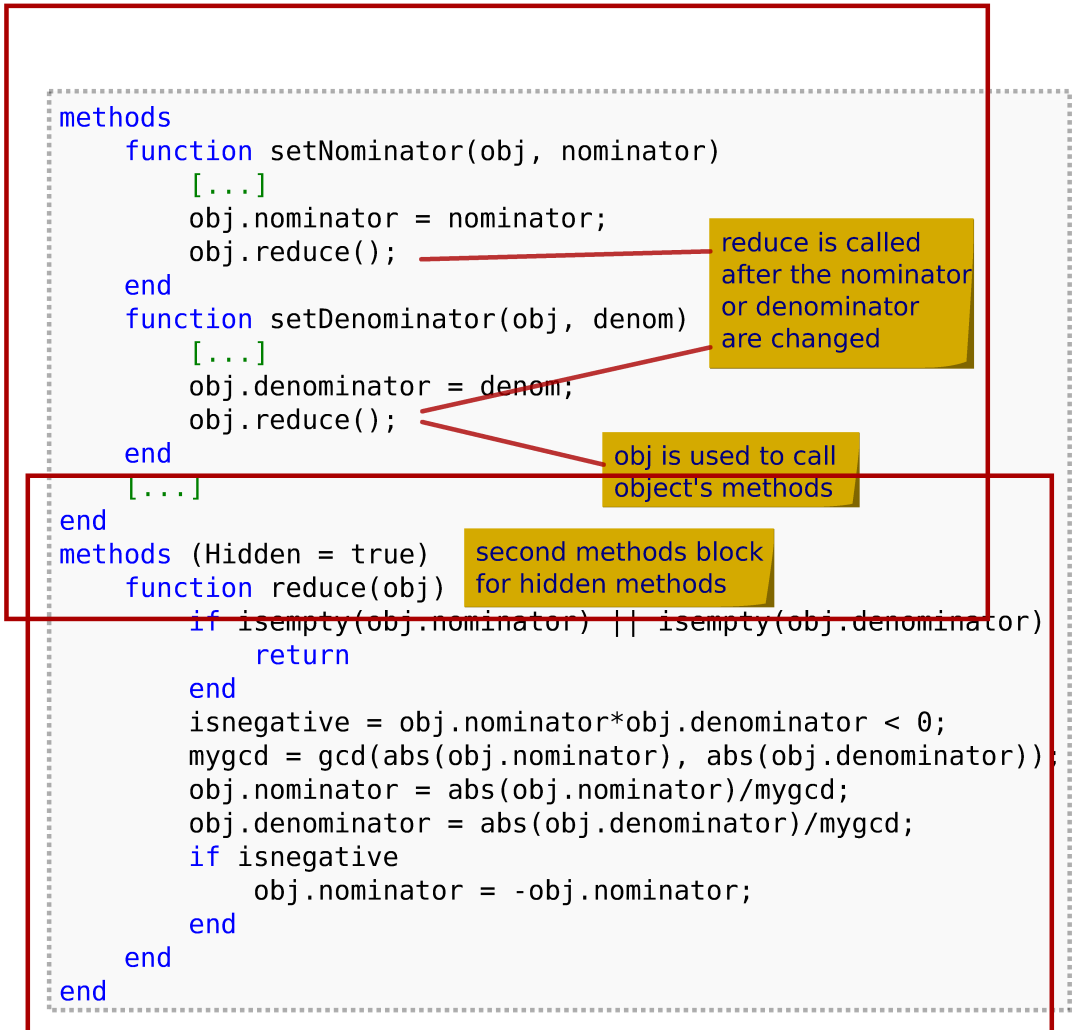
task: reduce the fraction (find greatest common divisor etc.)

➡ it needs to be called only when nominator or denominator are changed

➡ it can be called within `setNominator` and `setDenominator`

➡ it never needs to be called by the user

Encapsulation hides the internal details of a function from the user/code and provides a simpler interface.


methods
function setNominator(obj, nominator)
[...]
obj.nominator = nominator;
obj.reduce();
end
function setDenominator(obj, denom)
[...]
obj.denominator = denom;
obj.reduce();
end
[...]
end
methods (Hidden = true)
function reduce(obj)
if isempty(obj.nominator) || isempty(obj.denominator)
return
end
isnegative = obj.nominator*obj.denominator < 0;
mygcd = gcd(abs(obj.nominator), abs(obj.denominator));
obj.nominator = abs(obj.nominator)/mygcd;
obj.denominator = abs(obj.denominator)/mygcd;
if isnegative
obj.nominator = -obj.nominator;
end
end
end
reduce is called after the nominator or denominator are changed
obj is used to call object's methods
second methods block for hidden methods

After implementing the reduce function:

```
>> a = MyFraction(4,-12);
>> a.nominator
ans =
    -1
>> a.denominator
ans =
     3
```

Now, is there a way to let me see the value of a fraction more conveniently?

The disp function is responsible for displaying each object.
Matlab provides a default disp function.
When we leave out the semicolon, we can see the output of the default disp:

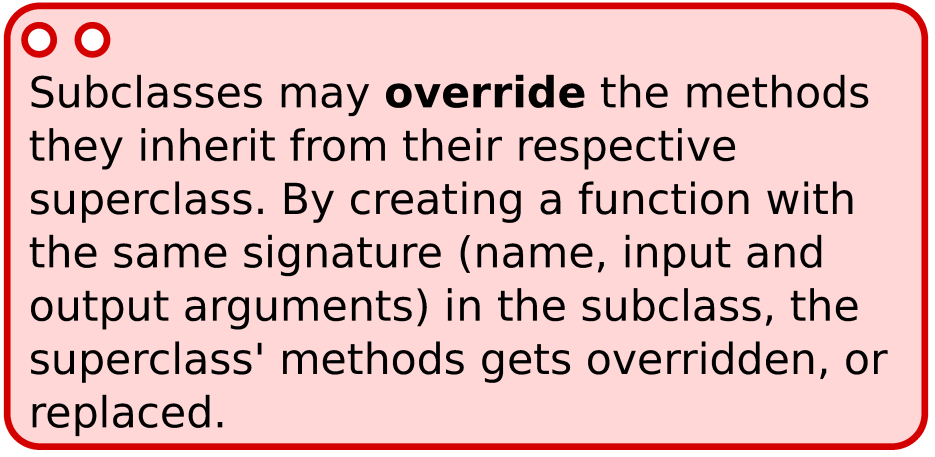
```
>> a = MyFraction(2,4)
a =
  MyFraction handle

  Properties:
      nominator: 1
    denominator: 2

  Methods, Events, Superclasses
```

But we'd rather like to see:

```
>> a = MyFraction(2,4)
a =
   1/2
```



MyFraction's superclass handle supplies a basic (and very general) disp function.
Let's override it...

We just need to know the signature (name, and number of input and output arguments) of a function we want to override; in our case: "`disp(obj)`"

Now, we just add a new function with this signature to our class and let it print the object the way we'd like:

```
classdef MyFraction < handle
    [...]
    methods
        [...]
        function disp(obj)
            fprintf('   %d/%d\n\n', obj.nominator, obj.denominator)
        end
        [...]
    end
    [...]
end
```

Note that we can also override other functions such as size, length etc.

Fractions are easy and safe to create now and they look nice.
But they are practically useless:

```
>> a = MyFraction(1,3);
>> b = MyFraction(1,4);
>> a + b
??? Undefined function or method 'plus' for input arguments of type 'MyFraction'.
```

It seems like a method called plus is missing for "+" to work.

```
>> help plus
 +   Plus.
    X + Y adds matrices X and Y.  X and Y must have the same
    dimensions unless one is a scalar (a 1-by-1 matrix).
    A scalar can be added to anything.

    C = PLUS(A,B) is called for the syntax 'A + B' when A or B is an
    object.
    ...
```

this is the information we are interested in, it tells us about the signature of plus: 2 inputs, 1 output
(the default input "obj" is listed here as "A")

Let's create a plus function for the MyFraction class, we start out simple:

```
    [...]
    methods
        [...]
        function out = plus(obj,obj2)
            out = MyFraction(obj.nominator*obj2.denominator + ...
                obj2.nominator*obj.denominator, ...
                obj.denominator*obj2.denominator);
        end
        function out = uminus(obj)
            out = MyFraction(-obj.nominator, obj.denominator);
        end
        function out = minus(obj,obj2)
            out = obj.plus(-obj2);
        end
        [...]
    end
    [...]
```

note that the name of the input arguments are not important (we could have called them A and B)

Here, we make the important assumption that both input arguments are MyFraction objects

while we are at it, here are some functions for minus and uminus (unary minus; called for "-a")

By creating a plus function for MyFraction, we overload plus.

Overloading methods means creating a new method with an existing name and new input arguments.

```
>> help plus
 +   Plus.
    X + Y adds matrices X and Y.  X and Y must
    have the same dimensions unless one is a
    scalar (a 1-by-1 matrix).
    A scalar can be added to anything.

    C = PLUS(A,B) is called for the syntax
    'A + B' when A or B is an object.

    Overloaded methods:
       ncitem/plus
       timeseries/plus
       lti/plus
       dynamicsys/plus
       distributed/plus
       codistributed/plus
       idmodel/plus
       laurmat/plus
       laurpoly/plus
```

plus already exists for various classes (input arguments)

→STEP6

This is working now:

```
>> a = MyFraction(1,3);
>> b = MyFraction(1,4);
>> a + b
ans =
   7/12
```

But these are not:

```
>> a = MyFraction(1,3);
>> a + 2
??? Attempt to reference field of non-structure array.
[...]
```

here the second input argument to plus is a double (not a MyFraction)

```
>> a = MyFraction(1,3);
>> 4.2 + a
??? Attempt to reference field of non-structure array.
[...]
```

here the first input argument to plus is a double

Solution: checks using the "isnumeric" and "isa" functions:

we can safely assume that the second argument must then be a MyFraction object, otherwise MyFraction's plus function would not be called

```
function out = plus(obj,obj2)
    if isnumeric(obj)
        out = MyFraction(obj*obj2.denominator+obj2.nominator,obj2.denominator);
    elseif isnumeric(obj2)
        out = MyFraction(obj2*obj.denominator+obj.nominator,obj.denominator);
    elseif isa(obj, 'MyFraction') && isa(obj2, 'MyFraction')
        out = MyFraction(obj.nominator*obj2.denominator+...
            obj2.nominator*obj.denominator, obj.denominator*obj2.denominator);
    else
        error('Cannot add %s to %s.', class(obj), class(obj2))
    end
end
```

`class(obj)`	returns the name of the class of obj, e.g. 'MyFraction'
`isa(obj,cname)`	returns true if obj is an object of the class cname, false otherwise
`isnumeric(obj)`	returns true if obj is a numeric (e.g. a double matrix or scalar), false otherwise

→STEP7

```
>> a = MyFraction(1,3);
>> b = MyFraction(1,4);
>> a + b
ans =
   7/12
```

```
>> a = MyFraction(1,3);
>> a + 3
ans =
  10/3
>> 2 + a
ans =
  7/3
```

```
>> a = MyFraction(1,3);
>> a + 'hello'
??? Error using ==> MyFraction>MyFraction.plus at 43
Cannot add MyFraction to char.
```

Using uminus and minus:

```
>> a = MyFraction(1,3);
>> -a + 2 + MyFraction(1,2) - 1
ans =
   7/6
```

We have reached our goal!

Now it is your turn to apply your new skills...