Chapter 4
Shell Hard Core

What is the hard core of a shell? Isn't it fair to say it is the pearl? If you are successful in mastering the Carabao shell hard core stuff you will hold the pearl of Carabao in your hand.

The *Carabao* shell provides a menu driven interactive graphical user interface. A main goal of the *Carabao* rapid prototyping approach is to build quickly powerful shells based on available building blocks. It is worth to have a separate chapter which is devoted solely to shell construction, and the reader who seriously wants to improve his skills of rapid shell prototyping is strongly advised to read this chapter with utmost attention. We have already a good understanding of *Carabao* shell functionality from a user's view, and we got some impression how a rapid prototyped shell looks like, but frankly speaking, we still do not have a solid understanding of the powerful possibilities of shell prototyping by utilizing *Carabao* methods and *Carabao* building blocks. This is the topic of this chapter.

Some Aspects of the Shell

Let us list some aspects of a *Carabao* shells on overview level.

- Both the construction of a shell as well as the callback handling is usually contained in one single m-file based method (at least for top-level control). Such a method which is capable to construct a shell is called a *shell* method. Any shell method should be well organized, in order to provide as much as possible overview and readability.

- Besides the default shell, which should be provided for every *Carabao* class, the class can provide more shells. Once a shell object (the object associated with the shell) is initialized that object must carry all information of shell reconstruction, e.g. for cloning, rebuilding or loading from .mat file.
- A *Carabao* shell contains exactly one shell object. This object can be a non-container object or a container object. In case of a container object the object can contain any number of child objects in its children list.
- A shell may have dynamic menus or not. Dynamic menus appear dynamically after object selection, depending on the selected object class. If a shell has no dynamic menus it is called a static shell. Dynamic shells can accept any object of class Carabao or derived from *Carabao* by pasting. Static shells can only accept pasted objects of a single class (the same class as the shell object's class).
- Shells may provide parts of a menu which can be used by other shells as building blocks.
- A shell building block may be used to refresh the outlook of a menu, e.g. updating varying menu entries or menu item check marks.
- A shell building block may contain hidden menu entry headers, which when used by other shells will turn automatically visible once a sub-menu item is added.
- Shells are assigned with settings which are independent of the shell object. This concept allows replacing the shell object while the shell settings are maintained.
- If an object is pulled from the shell the shell settings are temporarily stored in the work property of the pulled object and are accessible as so called options. Options might be changed during callback processing. This does not affect the shell settings.
- A shell method consists usually of plenty of local functions which are all contained in the method's m-file. Some of these functions are building blocks, others are callback functions, and finally there are

auxillary functions. According to Carabao philosophy both building blocks and callbacks have exactly one input argument (the shell object) and one output argument.

- Both building block and callback functions can be 'published' as 'managed functions'. In this case they are also accessible from outside the shell method.
- When an object is saved to a .mat file, also the options are saved, thus indirectly also the shell settings are saved, which allows a reconstruction of the shell during object load from .mat file.

A Simple Shell

Let us now construct a simple shell step by step based on our *Espresso* class. Our *Simple Shell* is not of big practical use but at this stage our major aim is to learn about the most important techniques and aspects of shell programming, which are:

- object initialization
- menu construction
- check & choice menus
- utilizing a menu building block
- providing a menu building block
- shell settings and object options
- object arguments
- callback handling
- local function dispatching
- refreshing the shell's screen
- rebuilding the shell's menu

Since these are quite a lot of topics we need about 40 code lines to demonstrate all of these aspects. The shell method shall have the name *simple*, and our *Simple Shell* shall only be associated with a non-container object. We are not interested on specific object data, but we want to have the possibility of doing some graphics, which we can modify by changing two shell settings (color, grid) which we will introduce. Here is the code.

```
function oo = simple(o,varargin) % simple shell(v1c/@espresso/simple.m)
   [func,oo] = manage(o,varargin,@Shell,@File,@Play,@Plot);
   oo = func(oo);              % dispatch to local function
end
%==========================================================================
% Shell Setup
%==========================================================================
function o = Shell(o)                    % Shell Setup
   o = Init(o);                          % initialize object
   o = menu(o,'Begin');                  % begin menu setup
   oo = File(o);                         % add File menu
   oo = Play(o);                         % add Play menu
   o = menu(o,'End');                    % end menu setup
end
function o = Init(o)                     % Object Initializing
   o = dynamic(o,false);                 % provide as a static shell
   o = launch(o,mfilename);              % setup launch function
   o = provide(o,'par.title','Simple Shell');
   o = provide(o,'par.comment',{'A simple shell to play'});
   o = refresh(o,{@menu,'About'});       % provide refresh callback
end
%==========================================================================
% File Menu
%==========================================================================
function oo = File(o)                    % File Menu
   oo = menu(o,'File');                  % add File menu
end
%==========================================================================
% Play Menu
%==========================================================================
function oo = Play(o)                    % Play Menu
   setting(o,{'play.grid'},false);       % no grid by default
   setting(o,{'play.color'},'r');        % default setting for color

   oo = mhead(o,'Play');                 % add Play menu header
   ooo = mitem(oo,'Sin',{@Plot,'sin'});% add Sin menu item
   ooo = mitem(oo,'Cos',{@Plot,'cos'});% add Cos menu item
   ooo = mitem(oo,'-');                  % add separator
   ooo = mitem(oo,'Grid',{},'play.grid');
   check(ooo,{});                        % add check functionality
   ooo = mitem(oo,'Color',{},'play.color');
   choices = {{'Red','r'},{'Green','g'},{'Blue','b'}};
   choice(ooo,choices,{});               % add choice functionality
end
function o = Plot(o)                     % Plot Callback
   mode = arg(o,1);                      % 1st arg: plot mode
   refresh(o,{@Plot,mode});              % update refresh callback
   col = opt(o,{'play.color','k'});      % get color option (default 'k')
   cls(o);                               % clear screen
   fct = eval(['@',mode]);               % make function handle
   plot(0:0.1:10,fct(0:0.1:10),col);     % plot sin or cos curve
   if opt(o,{'play.grid',0})             % grid option set?
      grid on;                           % show grid
   end
end
```

The code is organized as a main function *simple* and some local functions (*Shell*, *Init*, *File, Play* and *Plot*). Be aware that we use lower case letters for all method and global function names, and begin all local function names with upper case letters. This makes it easy to look at the right location for a function definition.

Before we study the semantics of the code we just focus on the code structure. What is the common structure of all those functions? Correct – all those functions have exactly one input argument and one output argument, and this structure is fully complying with *Carabao* philosophy. Passing a single object to a local function and returning a single object has two good reasons.

- First, local functions may call other *Carabao* methods, which require a *Carabao* object as the first input argument (like `cls(o)` in *Plot*). Then it is good to have already a *Carabao* object at hands.
- Second, the object carries information which might be used in local functions at deeper levels. E.g. this is the case with *Init* where method *provide* needs to check internally the value of *o.par.title* before it can change the parameter value. Other information that is used at deeper levels are graphics handles, shell settings which are passed as object options, and object arguments.

Imagine that the shell provides plenty of settings. If we need some of these settings at deeper function calling level we have to pass those settings as function arguments from function to function. And if we suddenly use an additional setting at deep calling level we need to change plenty of function interfaces. If, however, those settings are passed as object options (carried with the *work* property) we only have to pass one single argument per function call, and there is never a need to change an existing function interface. Consider an interface for a MATLAB® standard callback.

```
function Foo(object,event,arg1,arg2,…) % standard callback interface
```

MATLAB® callback philosophy is based on the handle graphics system, and the first two arguments which are passed to a callback are graphics handles which allow access to the parameters stored in the graphics system. The *Carabao* philosophy is different. *Carabao* callbacks receive only one single object, which is the shell object (pulled from the shell). Any further required information can be found in the object properties.

```
function o = Foo(o)                 % CARABAO callback interface
   object = o.work.object;          % get handle of graphics object
   event = o.work.event;            % get event structure
   arg1 = arg(o,1);                 % callback argument #1
   arg2 = arg(o,2);                 % callback argument #2
     :     :
end
```

If we are not interested on the object handle or other stuff (what is in most times the case) we leave this ballast away.

```
function o = Foo(o)                 % CARABAO callback interface
   mode = arg(o,1);                 % only interested on 1st argument
     :     :
end
```

Launching the Simple Shell

We stay with version *play/v1c*. Since we will modify the *Espresso* class interface all existing class instances must be cleared[1].

```
>> clear classes     % clear all classes & class instances
```

Now create a new empty m-file with the MATLAB® editor and insert the code for method *simple*. Save the file to *play/v1c/@espresso/simple.m*. After that we can launch our *Simple Shell*.

```
>> o=espresso('simple');    % construct ESPRESSO object (type 'simple')
>> launch(o);               % launch simple ESPRESSO shell
```

How does the *launch* method know that it has to consult our new method *simple* for shell launch? The reader may remember what has been told about the *type* property in chapter 2.

- *type*: determines data interpretation, and determines the default method for launching a shell

All right, that's the explanation! When we launch a shell using the *launch* method then by default the object *type* determines the actual shell launch method.

[1] exit and restart MATLAB® if `>> clear classes` does not work for some reasons

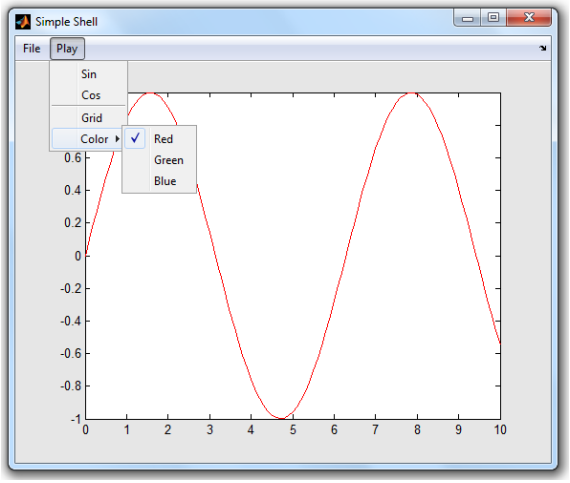

Fig. 4.1 – *Simple Shell displaying red sine curve*

If we made no errors a shell as shown in fig. 4.1 will pop up. With *Play>Sin* or *Play>Cos* a sine or cosine curve can be plotted. The *Simple Shell* provides two custom shell settings, one for grid visibility, and one for plot color. Shell settings can usually be changed via the menu. There is a menu item *Play>Grid* which allows to turn the grid on or off, and with *Play>Color* we can select a color for the plot. Note that the graphics immediately refreshes if any of the shell settings like *Play>Grid* or *Play>Color* is changed, and note the changing check marks in the roll down menus indicating the current choice of shell settings.

Explore also the *File* menu items. All functions are working properly, *File>Rebuild* and *File>Clone*, will preserve the current graphics and menu settings, and if we save the shell to a .mat file and re-open the shell from file we will see the same graphics and menu settings as they have been at saving time. Isn't that awesome?

Object Initializing

Before the shell figure is being opened and the menu is constructed the shell object (the object assigned with the shell) has to be initialized properly. For better overview all initializing stuff has been put into local function *Init*, which is called at the beginning of shell setup.

```
function o = Init(o)                    % Object Initializing
   o = dynamic(o,false);                % provide as a static shell
   o = launch(o,mfilename);             % setup launch function

   o = provide(o,'par.title','Simple Shell');
   o = provide(o,'par.comment',{'A simple shell to play'});
   o = refresh(o,{@menu,'About'});      % provide refresh callback
end
```

Note that all code lines of the *Init* function modify some object properties, that's why the return value of each method call must be re-assigned to the object variable *o*. There are at least 4 initializing tasks for a shell object

1) setting up the shell as static or dynamic
2) setting up the launch function
3) providing a title and comment
4) setting up a refresh callback

Omitting one of these tasks does not lead to a crash, although some weird shell behavior against the intuition could occur. The first command advises the shell to be a static shell (disable dynamic behavior).

```
o = dynamic(o,false);               % provide as a static shell
```

A static shell can only accept pasted objects of one single class (the shell object's class - in our case class `'espresso'`). We could also omit this line since static behavior is the default and our shell has anyway no copy & paste functionality (no *Edit* menu).

The next line defines the launch function explicitly. Note that the built-in function *mfilename* evaluates in our context to the string `'simple'`, thus the following lines are semantically equivalent.

```
o = launch(o,mfilename);            % setup launch function
o = launch(o,'simple');             % setup launch function
```

Why is this necessary? If we set the object type to `'simple'` then the *launch* function knows already by default which shell method to use for launch. But if we use the command

```
>> simple(o);                       % launch a simple shell
```

to launch our *Simple Shell* then the *File>Clone* and *File>Rebuild* functions would sometimes fail, since the reconstructed shell would depend on the object type. For this reason it is good style to setup the launch function explicitly during object initialization.

The remaining initializing tasks have to be implemented as conditional operations. This is very important to understand. It means that an initializing operation has always to check whether the actual value of a parameter or work variable is empty (not initialized), and only in this case the value can be changed (initialized).

Why is this important? Consider the *File>Clone* function, the *File/Rebuild* function or loading a shell from a .mat file. All these functions call the *launch* method and pass the already initialized shell object as an argument, which will be further passed through the initializing procedure of local function *Init*. Thus it should be clear that in a second step no internal object information must be changed, otherwise we would not get exact clones.

The method *provide* is designated for this special kind of task as by special nature of this method a property field will only be changed if the actual value is empty.

```
oo = provide(o,'par.title','Simple Shell');
oo = provide(oo,'par.comment',{'A simple shell to play'});
```

If no object *title* or *comment* is provided we should always provide defaults for these two parameters. Note that *title* is a character string while *comment* is a list of character strings. Also the *refresh* callback setup

```
oo = refresh(oo,{@menu,'About'});  % setup refresh callback
```

uses a special syntax (2 input args, 1 output arg) which updates the proper control option for the refresh callback only if the actual value is empty. Note that the list `{'menu','About'}` (also called a 'call list') denotes a refresh callback. You might expect a function handle `@menu` as the first list element and not a character literal `'menu'`, but this is not a typo – have a little patience and we will come back to this detail.

Shell Setup

We can study now the actual shell setup. This is done in Local function *Shell*.

```
function o = Shell(o)                  % Shell Setup
   o = Init(o);                        % initialize object
   o = menu(o,'Begin');                % begin menu setup
   oo = File(o);                       % add File menu
   oo = Play(o);                       % add Play menu
   o = menu(o,'End');                  % end menu setup
end
```

The first line of *Shell* calls the *Init* function we discussed in the previous section and which makes sure that the shell object is being initialized properly. Be aware that the initialized object is returned by *Init* and has to be re-assigned to object variable *o*. The rest of the code lines construct the shell. They are enclosed with statements

```
   o = menu(o,'Begin');                % begin menu setup
     :       :
   o = menu(o,'End');                  % end menu setup
```

which consult the *Carabao* method *menu,* a library of menu building blocks. The building block *Begin* is called at begin, and building block *End* is called at the end of menu construction. Building block *Begin* is responsible for the following tasks

- opening a new figure
- disabling the standard figure menu bar
- setting the figure title
- initializing shell settings based on object's options
- pushing the object into the figure
- initializing some control settings
- finally pulling the object from the shell, to refresh work variables

Building block *End* is mainly responsible for invocation of the *refresh* method in order to reconstruct a proper screen graphics after completed menu construction. In our case `menu(o,'End')` will trigger *refresh* to invoke the refresh callback that has been provided in the *Init* function, which will display the object information (title and comment) at the end of the menu setup.

In between the two blocks *Begin* and *End* there are the code lines which actually construct the menu. The first one

```
oo = File(o);    % add File menu
```

calls local function *simple>File* which is responsible for the construction of the File menu. We can investigate the implementation of *File*.

```
function oo = File(o)                    % File Menu
   oo = menu(o,'File');                  % add File menu
end
```

Obviously the *File* menu construction is further delegated to the *Carabao menu* method that provides a building block *menu>File* (which can be called by `oo = menu(o,'File');`) doing the actual work. This is a good example of how *Carabao* menu building blocks can be utilized[2]. The second code line calls local function simple/Play in order to construct the *Play* menu. We study the details later.

```
oo = Play(o);  % add Play menu
```

And that's it at this level. It is good programming style to provide only those code lines in the *Shell* setup function that describe the top level tasks (*Init*, *Begin*, *File, Play, End*) and leave the detailed menu construction to sub-functions like *File* and *Play*. The reader might be curious and eager to study the details of local function *Play*, but before it makes sense to do this we need to have a solid understanding of shell *settings*, object *options* and object *arguments*.

There is the final question how local function *Shell* is being invoked. When we launch the *Simple* shell using a command like `>>launch(espresso('simple'));` the *launch* method extracts *simple* to be the method for shell launch and invokes a command like

```
simple(o);  % launch a Simple shell
```

which causes execution of the *simple* method's main function

```
function oo = simple(o,varargin) % simple shell(v1c/@espresso/simple.m)
   [func,oo] = manage(o,varargin,@Shell,@File,@Play,@Plot);
   oo = func(oo);                % dispatch to local function
end
```

[2] remember also that the *Carabao menu* method is a library of menu building blocks

The first code line calls method *manage* with the variable input argument list (*varargin*) of *simple*, followed by a list of function handles (*@Shell*, *@File*, …). Since in our case *varargin* is empty method *manage* is advised to return the first function handle of above mentioned list, which is *@Shell*, being assigned to variable *func*. Thus the following line

```
oo = func(oo);              % dispatch to local function
```

will do nothing else but calling local function *Shell*, which answers our original question, how *Shell* is exactly being invoked.

Shell Settings

We mentioned already that a shell holds two kinds of information. The first kind of information is stored in the properties of the shell object. The other kind of information is known as shell settings. Shell settings are independently stored from the shell object in the figure's user data[3].

To demonstrate the usage of shell settings close all figures and launch a *Simple* Shell.

```
>> o=espresso('simple');  % construct a 'simple' typed espresso object
>> launch(o);             % launch a simple shell
```

The entire shell settings are stored as a structure. Using method setting we can display all shell settings.

```
>> setting(o) % display settings
    control: [1x1 struct]
       play: [1x1 struct]
```

There are *control* settings which are internal settings of the shell (e.g. the *refresh* callback is stored in the *control* settings). And there are custom settings. E.g *play* (which is a structure) contains all custom settings of our *Simple Shell*. We can easily access parts of the settings by providing a tag (like `'play'`) as second argument of method *setting*.

[3] a special data structure is used for the figure's user data which is called the *shelf*

```
>> play = setting(o,'play')    % retrieve settings 'play'
play =
     grid: 0
    color: 'r'
```

It is good programming style to put all settings which are related to a single roll down menu into a single structure with the same (or similar) name as the roll down menu. This means in our case that the *grid* and *color* settings related to the roll down menu *Play* have the name tags *play.grid* and *play.color.* We should expect a change of these settings when we select a related menu item. Click on *Play>Color>Blue* (watch the change of the check mark in the menu) and observe the actual values of setting *play*. The *play.color* setting has now the value `'b'` (blue).

```
>> play = setting(o,'play')    % retrieve settings 'play'
play =
     grid: 0
    color: 'b'
```

Individual shell settings can easily be changed. The following commands change the shell settings. Note, however, that the check marks of the menu, indicating the current choice of shell settings, will not be updated.

```
>> setting(o,'play.color','g');  % change setting 'play.color' to 'g'
>> setting(o,'play.grid',1);     % change setting 'play.grid' to 1
```

We could rebuild the shell menu by selecting *File>Rebuild* but with *Carabao* method *rebuild* we are able to rebuild the menu structure from program level. The related *Carabao* method *refresh* refreshes the shell's screen graphics.

```
>> rebuild(o)  % rebuild the shell's menu structure
>> refresh(o)  % refresh the shell's screen graphics
```

When the *rebuild* method refreshes the menu structure the following things happen:

- The shell method (in our example: *simple*) is invoked internally with a special control option activated to indicate that the menu structure has to be rebuilt. This special control option forces library function *menu>Begin* to go ahead with the current assigned figure, and not to open another new figure.
- The shell method delegates the rebuilding tasks to the menu building blocks which are responsible to rebuild the sub menus for which they are responsible.

Whenever a menu building block is called (like the library building block *menu>File* or our local function *simple>Play*) it must be able to rebuild an existing sub menu and actualize the graphical menu outfit, like check marks, enable/disable functionality or dynamic menu entries like we saw in the *Select>Objects* or *Gallery* menu. It must be avoided that a sub-menu is being duplicated. This sounds a bit challenging, but with *Carabao* methods *mhead* and *mitem* this task is a piece of cake, as we will see later.

A key requirement for a menu building block is that the related local function does not change shell settings which are already initialized. As it is good *Carabao* programming style to initialize shell settings related to a sub-menu in the function which constructs the sub-menu (as it is the case with *simple>Play)* there is a common need to initialize a shell setting conditionally (only if it is currently not initialized, i.e. the current value is empty). The following calling syntax of *setting* does exactly this job.

```
setting(o,{tag},value);        % conditional init (if not initialized)
```

It has equivalent effect as the following code lines have.

```
current = setting(o,tag);      % get current setting
if isempty(current)            % is current setting not initialized?
   setting(o,tag,value);       % initialize setting
end
```

Object Options

To change a setting it is usual to use the *setting* method, but it is very unusual to use *setting* if a setting's value needs to be retrieved.

```
>> setting(o,'play.color','g')       % very usual!
>> setting(o,{'play.color'},'g')     % also very usual!
>> color = setting(o,'play.color')   % very unusual! (only for debug)
```

Once an object is pulled from the shell (and this happens before each callback execution) the settings are copied to the object's options and can be accessed via the *opt* method. Close all figures, launch a *Simple Shell*, pull the object from the shell and study the options (similar to our study of the shell settings). We get the same results.

```
>> o=espresso('simple');  % construct a 'simple' typed espresso object
>> launch(o);             % launch a simple shell
```

```
>> o=pull(o);                % pull object from shell
>> opt(o)                    % display object's options
    control: [1x1 struct]
       play: [1x1 struct]
>> play = opt(o,'play')      % retrieve option 'play'
play =
     grid: 0
    color: 'r'
```

Why do we need this indirect step of options to retrieve shell settings. The reason is that *Carabao* philosophy is much about building block utilization and building block construction. Consider our Simple Shell.

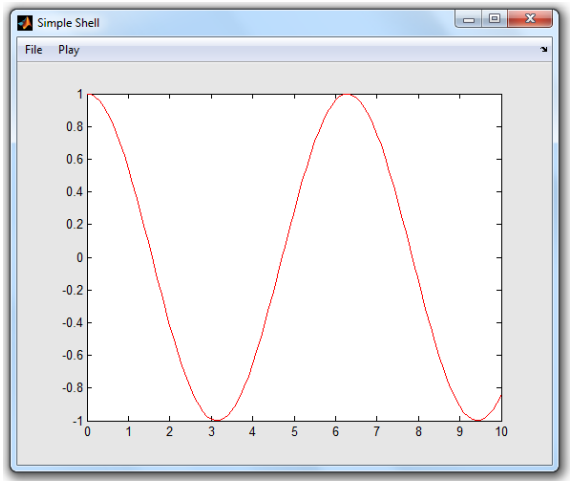

Fig.4.2 – A red cosine curve without grid appears

The main nature of the plot callback is the 'callback nature', displaying a sine or cosine plot on the shell's screen, when the *Play>Sin* or *Play>Cos* menu item is selected. But we can utilize *simple>Plot* also as a building block. Try the following.

```
>> simple(o,'Plot','cos');  % plot a red cosine curve, no grid
```

A red cosine curve appear's on the shell's screen with deactivated grid (fig. 4.2). Why a cosine curve? Because we provided plot mode `'cos'` as the 3rd command line argument. And why red and why no grid? Because the current shell settings are

```
>> setting(o,'play')    % retrieve settings 'play'
ans =
     grid: 0
    color: 'r'
```

which have been passed during the *pull* process (`>> o=pull(o)`) to the object's options, and we will see later in detail that the local *simple>Plot* function sets the graphics attributes according to the object's options.

```
>> opt(o,'play')                % retrieve options 'play'
ans =
     grid: 0
    color: 'r'
```

This means that we can change the outlook of the graphics by changing the object's options. Calling *Plot* again with modified options displays a green cosine curve with grid (fig. 4.3).

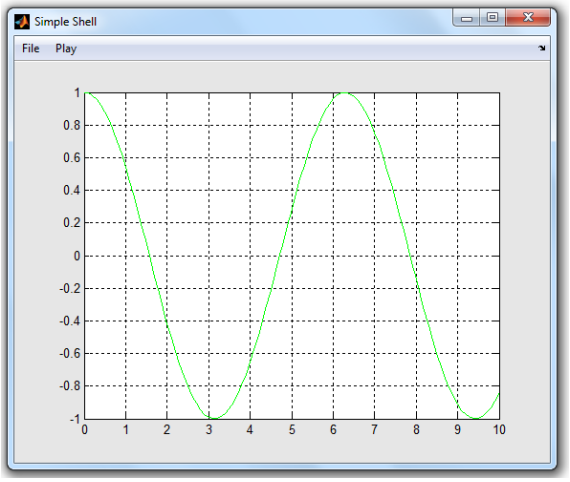

Fig. 4.3 – A green cosine curve with grid appears

```
>> o = opt(o,'play.color','g')   % set 'play.color' option to 'g'
>> o = opt(o,'play.grid',1)      % set 'play.grid' option to 1
>> simple(o,'Plot','cos');       % plot a green cosine curve with grid
```

In another typical example a shell might provide different sets of filter parameters for different kind of analysis tasks. A common *plot* method would be implemented to work with filter options, which are provided as `opt(o,'filter')`. If analysis#1 is activated the according callback would copy the specific filter settings #1 to the actual filter options. Some code part of analysis#1 might look as follows.

```
filter = opt(o,'analysis1.filter');   % filter parameter set #1
o = opt(o,'filter',filter);           % copy to common filter options
plot(o,mode);                         % call common plot method
```

The code part for analysis#2 would do a similar task copying filter parameter settings #2 to the common filter options.

```
filter = opt(o,'analysis2.filter');  % filter parameter set #2
o = opt(o,'filter',filter);          % copy to common filter options
plot(o,mode);                        % call common plot method
```

Note that the shell settings are not affected when the object's options are changed. Using options we get a much easier possibility to implement a flexible building block *plot* than working directly with shell settings.

Another good example for the power of the option concept is demonstrated by the *basket* method which has been shown in the previous chapter. For shells which entertain a basket several operations (like the plot method) are usually be implemented to operate on the *basket* list, and the scheme of operation for such kind of methods is as follows:

```
list = basket(o);          % get list of objects in basket
for (i=1:length(list))     % with all objects in the basket …
   oo = list{i};           % get i-th basket object
      :   :                % perform required operations
end
```

The *basket* method returns a list of basket objects, and at this point the basket is just an abstract construct which collects certain objects of the shell according to some rules which are controlled by specific shell settings. Thus the actual behavior of the plot method depends in general from those shell settings. If, however, there is a need for setting up the basket collection criteria in a specific way (not depending on the shell settings) this can be easily done by overwriting those options which control the basket collection mechanism. Exactly this possibility has been utilized for ensemble analysis of the Espresso shell (*Analyse>Ensemble*).

```
function o = Ensemble(o)                % Ensemble Analysis
     :     :
   o = opt(o,'basket.collect','*');     % all objects of the shell
   o = opt(o,'basket.type','shell');    % only 'shell' typed objects
   list = basket(o);
     :     :
end
```

The responsible code is found in local function *espresso/plot>Ensemble*, where the *basket.collect* option is set to value `'*'` (collect from all objects of the shell object …) and the *basket.type* option is assigned with the value `'shell'` (… but restrict collection on `'shell'` typed objects). When the list of basket objects is retrieved (`list = basket(o);`) the *basket* method does only care about the object options and will

provide a list of all `'shell'` typed objects of the shell object, no matter what the shell settings currently are. This was actually the idea behind the implementation.

With the *basket* method we have a wonderful example how the combination of shell settings and object options supports the flexibility of the *basket* building block in a powerful and effective way.

The most important things about options have been told now, but let me present two special forms of calling syntax for the *opt* method. The first form reads the option value and replaces the value with the default value if it is empty.

```
value = opt(o,{tag,default});        % get option, use default if empty
```

This form is useful if we write a building block and the building block cannot trust that the option is initialized[4], thus preventing a crash by providing a default value. See the equivalent statements.

```
value = opt(o,tag);                  % get option value
if isempty(value)                    % is option not initialized?
   value = default;                  % use default value if empty
end
```

The second form is important during object initialization. Remember that in general during the initialization procedure of a shell object a property may only be initialized, if the current value is empty (not initialized). The syntax

```
o = opt(o,{tag},value);              % set option only if empty
```

does exactly this kind of job. The command is equivalent to the following code lines:

```
current = opt(o,tag);                % get current option value
if isempty(current)                  % is option not initialized?
   o = opt(o,tag,value);             % initialize option
end
```

[4] Note that `opt(o,'xyz')` returns empty ([]) even option `'xyz'` has never been provided

Object Arguments

The last of the three concepts we have to cover are object *arguments*. Why do we need object arguments. Remember that the local function *simple>Plot* is both a callback function and a building block. Remember that we could pass a mode argument the (`'sin'` or `'cos'`) to *simple>Plot*.

```
>> simple(o,'Plot','sin');  % plot a sine curve
>> simple(o,'Plot','cos');  % plot a cosine curve
```

The function interface of *Plot*, however, has only one input argument (object *o*, in compliance with *Carabao* philosophy).

```
function o = Plot(o)   % Plot Callback
   mode = arg(o,1);    % 1st arg: plot mode
     :      :
end
```

As a conclusion the function arguments have to be passed as part of some object property, and in fact the so called 'object arguments' are carried by the *arg* field of the work property. Construct an object and provide a list of three arguments.

```
>> o=espresso;                  % construct an object
>> o=arg(o,{pi,@foo,'sin'})     % set object arguments (list of 3 args)
ESPRESSO object
 MASTER Properties:
  tag: espresso
  type: shell
  par: []
  data: {}
 WORK Property:
    arg: {[3.1416]  [@foo]  'sin'}
```

Method *arg* manages all access to the object arguments. We can retrieve the whole argument list and request the number of provided arguments.

```
>> arg(o,0)                     % retrieve list of object arguments
ans =
    [3.1416]    @foo    'sin'
 >> arg(o,inf)                  % how many args? (info about arg count)
ans =
     3
```

Arguments can be retrieved by index, and *arg* does not crash if the index exceeds the number of provided arguments.

```
>> arg1=arg(o,1)                % retrieve 1st object argument
```

```
arg1 =
    3.1416
>> arg2=arg(o,2)                % retrieve 2nd object argument
arg2 =
    @foo
>> arg3=arg(o,3)                % retrieve 3rd object argument
ans =
    sin
>> arg4=arg(o,4)                % retrieve 4th arg (does not crash)
ans =
     []
```

Finally you should be aware that when arguments are passed to a function via object arguments it never cares about number of arguments, so this technique is a natural way to deal with variable argument lists. In this sense you should also note that there is an alternative way to invoke the local *simple>Plot* method. The following command

```
>> o = arg(o,{'Plot','sin'});   % arg list for plotting a sine curve
>> simple(o);                   % plot a sine curve
```

plots a sine curve, and applying our learning from last section we can construct a command that plots a magenta colored cosine curve (not providing the best readability, but effective).

```
>> simple(arg(opt(o,'play.color','m'),{'Plot','cos'}));
```

The Plot Callback

Equipped with a good understanding of object *options* and object *arguments* we can study now the implementation of the *simple>Plot* callback function.

```
function o = Plot(o)                    % Plot Callback
   mode = arg(o,1);                     % 1st arg: plot mode
   refresh(o,{@Plot,mode});             % update refresh callback
   col = opt(o,{'play.color','k'});     % get color option (default 'k')
   cls(o);                              % clear screen
   fct = eval(['@',mode]);              % make function handle
   plot(0:0.1:10,fct(0:0.1:10),col);    % plot sin or cos curve
   if opt(o,{'play.grid',0})            % grid option set?
      grid on;                          % show grid
   end
end
```

The first line accesses the first object argument and assigns it to variable *mode*. Remember that *mode* carries a function name like `'sin'` or `'cos'`. In the next line a

refresh callback is installed with function handle *@Plot* and one callback argument (*mode*). We will later come back to the refresh mechanism in detail.

The next statement retrieves the plot color from the options and assigns it to variable *col*. Remember that each *Carabao* callback is processed by pulling first the object from the shell and invoking then the callback function by passing the (pulled) object. The *pull*–process makes sure that the shell settings are copied to the object options. This ensures finally that variable *col* carries a copy of the shell setting *play.color* (unless someone provided the color option in a different way, as we did it for demonstration in the previous section). Note also that the chosen syntax

```
col = opt(o,{'play.color','k'}); % get color option (default 'k')
```

would provide a default value `'k'` (black) if the option were not initialized. After clearing the shell's screen (`cls(o)`) we have to construct a function handle from *mode*. Assume now that *mode* has the value `'sin'`. Then the command `fct=eval(['@',mode])` means effectively `fct=eval('@sin')`, which results in the function handle *@sin*. The function handle will be used in the following plot command to draw a sine or cosine curve. Finally the grid is drawn if the grid option is set. All in all with the right understanding of object options and arguments: no rocket science!

Menu Construction

In a previous section I advised the reader to suspend the study of local function *simple>Play.* Now it is time to resume this study, which will tell us how menus are effectively constructed using *Carabao* methods.

```
function oo = Play(o)                    % Play Menu
   setting(o,{'play.grid'},false);       % no grid by default
   setting(o,{'play.color'},'r');        % default setting for color
   oo = mhead(o,'Play');                 % add Play menu header
   ooo = mitem(oo,'Sin',{@Plot,'sin'});% add Sin menu item
   ooo = mitem(oo,'Cos',{@Plot,'cos'});% add Cos menu item
   ooo = mitem(oo,'-');                  % add separator
   ooo = mitem(oo,'Grid',{},'play.grid');
   check(ooo,{});                        % add check functionality
   ooo = mitem(oo,'Color',{},'play.color');
   choices = {{'Red','r'},{'Green','g'},{'Blue','b'}};
   choice(ooo,choices,{});               % add choice functionality
end
```

The function starts with two code lines for the conditional initialization of our custom settings *play.grid* and *play.color*. As already mentioned, it is good *Carabao* programming style to initialize shell settings related to a sub-menu at the beginning of the function which constructs the sub-menu. This is especially important if we write a building block for a sub-menu, as it ensures that the building block does also the proper initialization of the related shell settings.

For the initialization of shell settings the same rule applies as for object initialization: a shell setting is only allowed to be initialized if it is not yet initialized (if the current value is empty). Otherwise a cloned shell would never lead to an exact clone. The conditional calling syntax of the *setting* method does exactly this kind of job.

In the following the menu items are constructed. The construction sequence starts as follows.

```
oo = mhead(o,'Play');                % add Plot menu header
ooo = mitem(oo,'Sin',{@Plot,'sin'}); % add Sin menu item
ooo = mitem(oo,'Cos',{@Plot,'cos'}); % add Cos menu item
```

Some of the readers might be surprised now that there are no code lines which are calling the built-in function *uimenu*, the MATLAB® work horse for menu item construction. For the construction of the *Play* menu of our *Simple Shell* some code would be expected to look like

```
LB = 'Label'; CB = 'Callback';          % short hands
men = uimenu(gcf,LB,'Play');            % add Plot menu header item
itm = uimenu(men,LB,'Sin',CB,{@Plot,'sin'});  % add Sin menu item
itm = uimenu(men,LB,'Cos',CB,{@Plot,'cos'});  % add Cos menu item
```

But hold on. Using the standard MATLAB® style our callback functions would have to provide the following function interface.

```
function Plot(object,event,mode)    % MATLAB style callback interface
```

But this kind of callback interface is not complying to the Carabao philosophy which requires us to use callback function with a single argument – the shell object. For this reason we need to redirect the callback flow to a wrapper callback method *master*

which pulls the object from the shell and packs all relevant information into the object. In addition a method *call* would be helpful to create properly formed callbacks[5].

```
LB = 'Label'; CB = 'Callback';              % short hands
men = uimenu(gcf,LB,'Play');                % add Plot menu header item
callback = call(o,class(o),{@Plot,'sin'});  % construct callback
itm = uimenu(men,LB,'Sin',CB,callback);     % add Sin menu item
callback = call(o,class(o),{@Plot,'cos'});  % construct callback
itm = uimenu(men,LB,'Cos',CB,{@Plot,'cos'});% add Cos menu item
```

I know, this is hard stuff and not easy to follow. But it is not important that you are able to follow all these details. The most important thing is that you are able to recognize how ugly the required code for the menu construction would look and how difficult the code would be to read, spoiled with short hands and technical stuff which does not represent the essentials. Now look on the following code based on *Carabao* methods *mhead* and *mitem*.

```
oo = mhead(o,'Play');                  % add Plot menu header
ooo = mitem(oo,'Sin',{@Plot,'sin'});   % add Sin menu item
ooo = mitem(oo,'Cos',{@Plot,'cos'});   % add Cos menu item
```

Every single chunk of this code is significant, free of ballast, and also without detailed explanation the meaning of the code can be catched intuitively: A menu is to be constructed with menu header *Play* and two sub-menu items *Sin* and *Cos*. The submenu items are provided with callbacks to function *Plot* which (in compliance with Carabao philosophy) needs to be invoked with a single object pulled from the shell that carries one object argument `'sin'` (or `'cos'`).

Menu Item Construction

Let us study now the details of menu item construction. The code in *simple>Play* uses Carabao methods *mhead* and *mitem*, but to catch the principle let us study a simpler way of menu construction, using only *mitem*[6]. The simplified version of local function

[5] if the reader is now totally confused I just propose to relax and continue reading

[6] the code using *mitem* instead of *mhead* is still capable for a perfect initial construction of the menu, although the rebuild capability got lost to refresh the check marks of the actual shell setting selection

Play does also not provide initializing code for shell settings, and there are no menu items for change of grid or color settings.

```
function oo = Play(o)                      % Simpel Play Menu
   oo = mitem(o,'Play');                   % add Play menu header
   ooo = mitem(oo,'Sin',{@Plot,'sin'}); % add Sin menu item
   ooo = mitem(oo,'Cos',{@Plot,'cos'}); % add Cos menu item
end
```

The meaning of the input arguments for method *mitem* is as follows[7].

```
oo = mitem(o,label,callback,userdata);
```

Note that with each call to *mitem* an object is passed as the first argument. The object holds a menu item handle to a parent graphics object, which is stored in *work* property *o.work.mitem*.

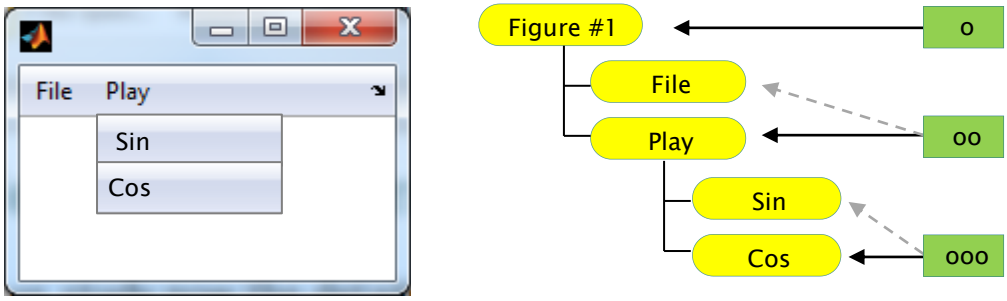

Fig. 4.4 – sample menu and related graphics object hierarchy

If a new menu item is to be constructed a graphical object will be created which is attached as a child object to the parent object. The handle of the child object is stored in a copy *oo* of the parent object *o,* while *oo* is returned by method *mitem*. A similar process is repeated with *oo* and *ooo* (see fig. 4.4).

Let's get hands-on in order to get in touch with the details! Close all figures (*File>Exit*) and initialize a shell object.

```
>> o=provide(espresso,'par.title','My Simple Shell');
>> o=refresh(o,{@menu,'About'});        % setup refresh callback
```

Now pop up an empty shell figure.

```
>> o=menu(o,'Begin');                   % popup an empty shell figure
```

[7] there are even more input arguments possible for *mitem* (consult `>> help carabao/mitem`)

This command calls local function *menu>Begin* [8]. A new figure titled 'My Simple Shell' with an empty menu bar pops up. Let us construct the three menu items of our simplified *Play* menu.

```
>> oo=mitem(o,'Play');                              % add Play menu header
>> ooo=mitem(oo,'Sin',{@simple,'Plot','sin'});     % add Sin menu item
>> ooo=mitem(oo,'Cos',{@simple,'Plot','cos'});     % add Cos menu item
```

The second and third line provides also a callback list. Since we invoke the commands from the MATLAB® command line we have no possibility to create the local function handle *@Plot* but with the provided callback list it is possible to make the callback running, even we build the menu from the MATLAB® command interpreter.

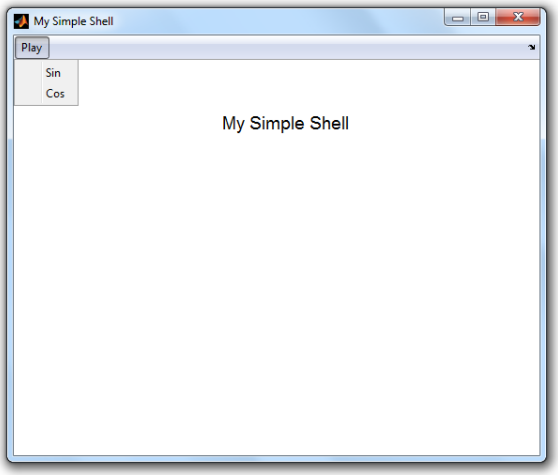

Fig. 4.5 – Play menu construction

Let us complete the setup of our tiny menu with the command

```
>> o=menu(o,'End');              % end menu setup
```

The refresh callback which we prepared during object initialization gets activated and displays the message 'My Simple Shell' on the cleared screen (fig. 4.5). The menu has one roll-down menu header *Play* with two menu items *Sin* and *Cos*. When we click on those menu items a black sine or cosine curve is plotted on the shell's screen. The callbacks seem to run, and we should not be irritated about the black color: Up to now we did not provide any shell setting *play.color*, thus the plot function receives the default value `'k'` (black) when it retrieves the uninitialized option.

[8] actually: *menu@carabao>Begin*

Check & Choice

In a next step we will add a menu item with check functionality. Even we invoked the concluding `menu(o,'End')` statement it is not forbidden to continue menu construction. In this sense let us first add a separator to our *Play* menu, before we continue menu construction.

```
ooo = mitem(oo,'-');                          % add separator
```

In order to construct a menu item with check functionality we have to provide an initialized shell setting on which the check menu item can operate. The following command uses the proper syntax, since the shell setting *play.grid* may only receive the initial value 0 if the setting has not been initialized before (does not exist or has empty value).

```
>> setting(o,{'play.grid'},false);          % no grid by default
```

Initializing a shell setting which is related to a menu is usually done in the local function which is responsible for that menu. Once the shell setting is initialized a check menu item can be setup as follows.

```
>> ooo = mitem(oo,'Grid',{},'play.grid');
>> check(ooo,{});                                % add check functionality
```

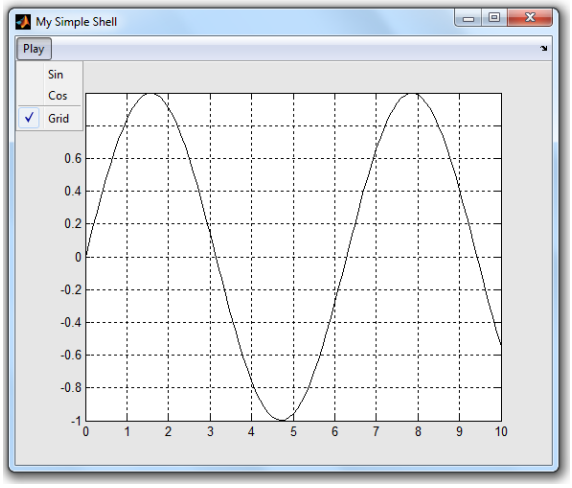

Fig. 4.6 – Tiny menu allowing to switch grid on/off

The first line adds a menu item to the parent item which is handled by object *oo* (our *Play* menu item), provided with label `'Grid'`, empty callback and user data `'play.grid'`. Note that the user data refers to the shell setting to be controlled. The *mitem* call returns *ooo* as a copy of object *oo,* holding the graphics handle to the new created menu item. The second command `check(ooo,{});` adds the required check functionality. After entering those commands we can actually plot a sine or cosine function and switch the grid on or off by clicking *Play>Grid* (fig. 4.6).

After each click on Play>Grid the screen refreshes immediately, and this is because of the special syntax we used, providing an empty list as second input argument. The following options for refresh are possible.

```
check(o)                    % no refresh callback
check(o,{@MyRefresh})       % use custom refresh callback
check(o,{@refresh})         % use standard refresh callback
check(o,{})                 % same as above (short form)
```

Be aware that any refresh callback must comply to the *Carabao* convention having one input argument and one output argument.

```
function o = MyRefresh(o)   % custom refresh callback
   :    :
end
```

As a conclusion we see that the construction of a check menu item is very simple. What about the choice menu functionality. Again we need a shell setting to be controlled by the choice menu, and the same rules regarding conditional initializing apply also here.

```
setting(o,{'play.color'},'r');      % default setting for color
```

Like with the check menu item the construction of a choice menu requires also two steps, the first step is to construct a header menu item for the choice menu, and the second step is to add actually the choice functionality.

```
>> ooo = mitem(oo,'Color',{},'play.color');
>> choices = {{'Red','r'},{'Green','g'},{'Blue','b'}};
>> choice(ooo,choices,{});          % add choice functionality
```

After assigning a list of choices to variable *choices*, which is actually a list of pairs comprising a menu label and a related value for the shell setting, the *choice* method

passes the *choices* list as second input argument. The third input argument defines the refresh functionality in the same way as with the *check* method. Let us define a second choice menu which controls an integer setting *play.level*.

```
>> setting(o,{'play.level'},1);              % default setting for level
>> ooo = mitem(oo,'Level',{},'play.level');
>> choice(ooo,[0:5, 10],{});
```

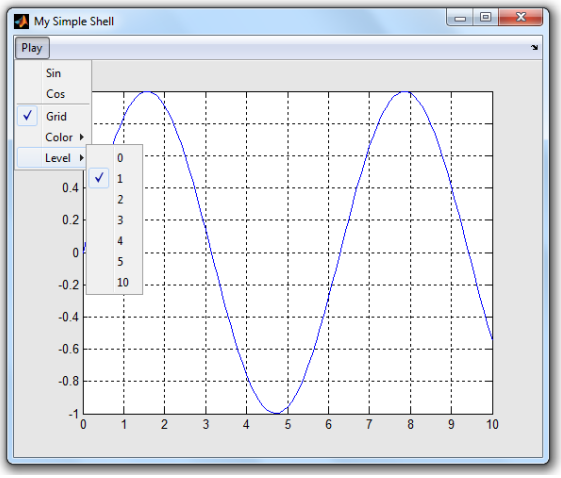

Fig 4.7: Tiny menu allowing to select a plot color

After entering all these commands we get a menu that allows us e.g. to plot a blue sine curve, and it supports a *Play>Level* sub-menu for the choice of the *play.level* setting (fig 4.7), even the current *Espresso* methods do not make use of *play.level*.

Menu Rebuilding

We made a slight simplification, using *mitem* (instead of *mhead*) for construction of the *Play* menu header item and got the following code for *simple>Play*.

```
function oo = Play(o)                    % Play Menu
   setting(o,{'play.grid'},false);       % no grid by default
   setting(o,{'play.color'},'r');        % default setting for color
   oo = mitem(o,'Play');                 % add Play menu header
   ooo = mitem(oo,'Sin',{@Plot,'sin'});% add Sin menu item
   ooo = mitem(oo,'Cos',{@Plot,'cos'});% add Cos menu item
   ooo = mitem(oo,'-');                  % add separator
   ooo = mitem(oo,'Grid',{},'play.grid');
   check(ooo,{});                        % add check functionality
   ooo = mitem(oo,'Color',{},'play.color');
   choices = {{'Red','r'},{'Green','g'},{'Blue','b'}};
   choice(ooo,choices,{});               % add choice functionality
end
```

Paste this code into simple.m, save the m-file and open a *Simple* shell. We can plot sine and cosine curves, change the plot color and turn the grid on and off – everything works fine. Even if we save the shell to .mat file and open the shell from that file, or we clone the shell – everything works fine.

But click now multiple times on *File>Rebuild*. With each selection of the *Rebuild* function we get another copy of menu *Play* (fig. 4.8).

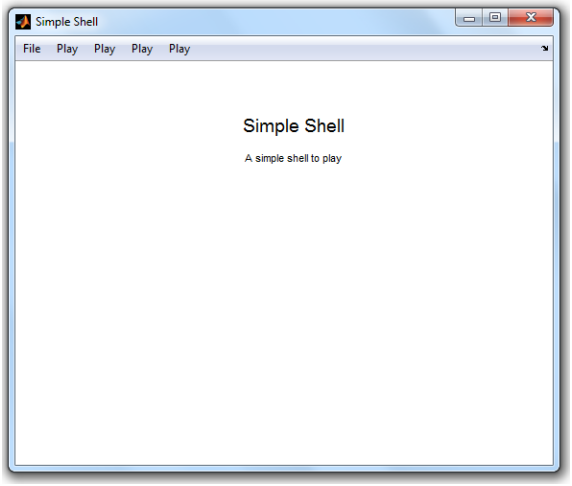

Fig 4.8 – each click on *File>Rebuild* adds a copy of menu *Play*

The same effect appears if we invoke method *rebuild*.

```
>> rebuild(espresso);     % rebuild menu structure
```

What happens? The rebuild process is triggered by the *rebuild* method which causes any shell to build up the menu structure a second time[9]. In order to prevent construction of an additional *Play* menu we use method *mhead* at the beginning of a menu construction sequence.

```
function oo = Play(o)                    % Simple Play Menu
     :     :
   oo = mhead(o,'Play');                 % add Play menu header
   ooo = mitem(oo,'Sin',{@Plot,'sin'}); % add Sin menu item
   ooo = mitem(oo,'Cos',{@Plot,'cos'}); % add Cos menu item
     :     :
end
```

[9] in a special rebuild mode, i.e, `o.work.rebuild == true`

By invoking `oo = mhead(o,'Play');` method *mhead* performs the following tasks:

- Seek for a menu item with label 'Play' in the graphics tree relative to a menu sub-tree with sub-tree root handle given by `o.work.mitem`.
- If the seek was successful assign object variable oo with a duplicate of object o and store the graphics handle of the located menu item in `oo.work.mitem`. Otherwise create a new menu item calling `oo = mitem(o,'Play')`.
- For the graphics object which is referenced by handle `oo.work.mitem` delete all child objects.

With this procedure (and consistent implementation of menu construction) an existing menu item is never being duplicated, and it is ensured that the returned object *oo* references a menu-item which has no sub-tree (fig. 4.9). The remaining construction sequence adds the missing or deleted sub-menu items in accordance to the current shell settings and shell object properties, thus getting to a well updated menu structure.

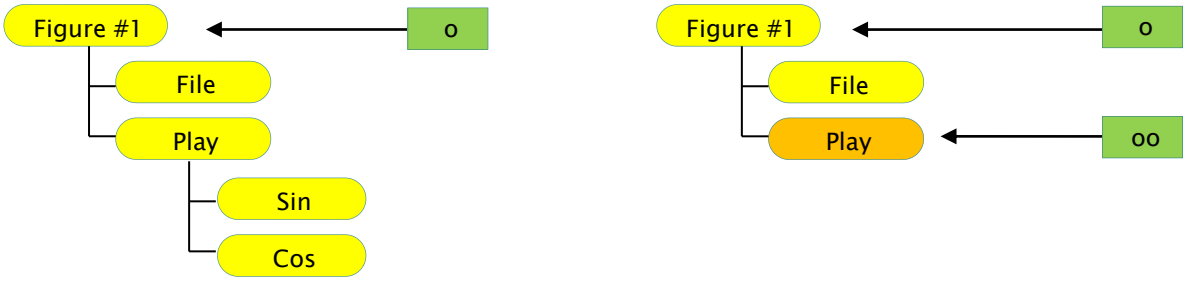

Fig. 4.9 – *mhead* locates/constructs a menu item and deletes the sub-tree

Let us study this process with a simplified menu. Close all figures (*File>Exit*) and begin a simple shell.

```
>> o=provide(espresso,'par.title','My Simple Shell');
>> o=refresh(o,{@menu,'About'});       % setup refresh callback
>> o=menu(o,'Begin');                   % begin an empty shell figure
```

A new figure opens with an empty menu. After providing an initialized shell setting *play.grid* create the menu header item for the *Play* menu.

```
>> setting(o,{'play.grid'},false);      % no grid by default
>> oo = mhead(o,'Play');                % add Play menu header
```

A menu header labeled `'Play'` appears in the menu bar. Method *mhead* has been seeking for an existing menu item labeled `'Play'` but did not find such. Thus it created a new menu item consulting *mitem* method. Let us continue now adding the following menu construction items and finishing the menu construction with *menu>End*.

```
>> ooo = mitem(oo,'Sin',{'simple','Plot','sin'}); % add Sin menu item
>> ooo = mitem(oo,'Cos',{@Plot,'cos'});           % add Cos menu item
>> ooo = mitem(oo,'-');                           % add separator
>> ooo = mitem(oo,'Grid',{},'play.grid');         % add Grid menu item
>> check(ooo,{});                                 % add check funct.
>> o = menu(o,'End');                             % end menu setup
```

The shell refreshes the screen and we get a tiny menu with header *Play*, functional menu items *Sin* and *Cos* and a check menu item *Grid* to switch the grid on and off. By selecting *Play>Sin* and *Play>Grid* the shell presents us a graphics like shown in fig. 4.6, a black sine curve with activated grid. Certainly the shell setting *play.grid* must have the value 1.

```
>> setting(o,'play.grid')       % investigate value of setting
ans =
     1
```

Now clear the value of setting *play.grid* from the MATLAB® command line. We understand already that this will only change the shell setting but will not alter the check mark of menu item *Play>Grid*.

```
>> setting(o,'play.grid',0)     % clear value of setting
```

In order to update the menu structure we would usually consult method *rebuild*. This would not work at the current stage, as we have no procedure which could be called by *rebuild* and would do the actual tasks.

So let us simulate the rebuild process. Be aware that we must repeat every command that has been invoked before with the only difference that work property field *o.work.rebuild* is set to value *true* to indicate a rebuild process. Re-invoke *menu>Begin*.

```
>> o.work.rebuild = true;       % indicates a rebuild process
>> o=menu(o,'Begin');           % ignore: begin an empty shell figure
```

Since *Begin* recognizes that we are now in a rebuild mode it ignores more or less its usual task to open a new figure and to follow up with some initializing stuff. Nothing happens – and nothing has to happen! We must now continue each step from before.

```
>> setting(o,{'play.grid'},false);      % no grid by default
>> oo = mhead(o,'Play');                % add Play menu header
```

Still everything seems to look as before, but wait a moment. If we investigate menu *Play* then all sub menu items of *Play* have been deleted. Let us understand exactly what happened after *mhead* has been invoked a second time: In a first step method *mhead* seeked for a menu item labeled *Play*, which has been located successfully. Thus the creation of another menu item *Play* (using `oo = mitem(o,'Play');` – the second step) has been suppressed. In the third step all child menu items of menu item *Play* have been deleted. That is why we see the same menu header *Play* (and no duplicated one) but with the whole submenu deleted.

What has to happen in the following is straight forward. We have to invoke the same commands as before, until we end with `o = menu(o,'End');`

```
>> ooo = mitem(oo,'Sin',{'simple','Plot','sin'}); % add Sin menu item
>> ooo = mitem(oo,'Cos',{@Plot,'cos'});           % add Cos menu item
>> ooo = mitem(oo,'-');                           % add separator
>> ooo = mitem(oo,'Grid',{},'play.grid');         % add Grid menu item
>> check(ooo,{});                                 % add check funct.
>> o = menu(o,'End');                             % end menu setup
```

This sequence of commands will rebuild the menu structure and when the *Grid* item is being rebuilt the *check* method will prevent to provide a check mark according to the cleared status of shell setting *play.grid*. Once we reach at the end of menu setup local function *menu>End* will refresh the shell's screen which will cause to plot a black sine curve without grid.

The menu rebuild process has been performed perfectly and the refresh of the shell's screen has been executed due to our satisfaction. Finally we got a good understanding why there's a need for two menu item construction methods, *mhead* and *mitem*, and we also understand now how simple it is in the *Carabao* world to rebuild a menu structure according to given shell settings.

The only thing missing is to change back the construction of menu item *Play* using *mhead* in *simple.m*. After that I would say: Break on Carabao Island!

Fig. 4.10 – Break on Carabao Island!

Lessons Learnt

At least now it should be clear that the *Carabao* architecture provides much more than replacing MATLAB callback style with *Carabao* callback style, and wrapping MATLAB *uimenu* calls with the *mitem* method.

Carabao provides a consistent concept to manage two different kinds of informations which define an actual menu structure of a shell's menu: shell settings and object properties. By pulling an object from the shell the shell settings are packed into object options such that the whole information managed by the shell is available in the object. As a consequence launching such an object will reconstruct both the shell's menu structure and refreshing the shell's screen in order to get the same visual outlook as it has been at time of *pull*.

Much of the *Carabao* shell construction concept deals with enabling easy-to-read code for rebuilding menu structures if shell settings are changed programmatically. Furthermore *Carabao* menus are usually built by using menu building blocks. It is easy to utilize a menu building block, as it is easy to provide a menu building block which can be utilized by other users.

In this chapter we met a lot of new *Carabao* methods. Here is a summary of the most important aspects of those methods.

- A shell object can be replaced with the *push* method, while the shell settings will be maintained.

```
push(o);  % replace shell object while keeping shell settings
```

- Once the shell object is pulled from the shell, the shell settings are stored as options in the object's work property. The pulled object contains all information to reconstruct a shell including the shell's outfit and current graphics. A shell can be completely reconstructed from such object using method *launch*.

```
o = pull(carabao);  % pull object from shell (copy settings to opts)
launch(o);          % reconstruct shell (as it has been during pull)
```

- A shell setting can easily be created or conditionally initialized, i.e its value will only be changed if it has not been initialized (or has empty value).

```
setting(o,bag);                  % init settings
setting(o,'play.grid',value);    % set value of 'play.grid'
setting(o,{'play.grid'},value); % conditional setting of 'play.grid'
```

The value of a shell setting can be retrieved directly or with default value, i.e. a default value is returned if the setting has empty value.

```
bag = setting(o);                           % get all settings
value = setting(o,'play.grid');             % get setting 'play.grid'
value = setting(o,{'play.grid',default}); % get default if empty
```

- Shell settings are copied to object options during *pull*. Object options can be accessed in a similar way to shell settings.

```
o = opt(o,bag);                          % init options
o = opt(o,'play.grid',value);            % set option 'play.grid'
o = opt(o,{'play.grid'},value);          % conditional option setting
bag = opt(o);                            % get all options
value = opt(o,'play.grid');              % get option 'play.grid'
value = opt(o,{'play.grid',default}); % get default if empty
```

- Objects may have arguments which can be accessed by method *arg*.

```
o = arg(o,{arg1,arg2,…});              % set list of object arguments
list = arg(o,0);                       % get list of object arguments
argi = arg(o,i);                       % get i-th object argument
n = arg(o,inf);                        % number of object arguments
```

- Any object's property or property field can be conditionally initialized using method *provide*, i.e it will be changed only if it is not yet initialized (or empty).

```
o = provide(o,'par.title','Simple Shell');
o = provide(o,'dat.x',0:0.1:10);
o = provide(o,'type','shell');
o = provide(o,'work.var.a',pi);
```

- Usually any object class derived from *Carabao* provides at least one shell method, which by default should have the name *shell*. The shell method opens a figure and constructs and manages the shell menu. It is possible to provide more than one shell method. Besides of the launch command a shell can be launched using the related shell method.

```
shell(o);              % launch default shell
simple(o);             % launch simple shell
```

- After pulling an object from a shell the *launch* method always knows which of the shell methods has to be used in order to re-launch the shell. To make this scenario happen the launch function name must be setup during shell object initialization.

```
o = launch(o,'simple');      % setup launch function -> 'simple'
o = launch(o,mfilename);     % setup launch function -> mfilename
```

- When a shell object is initialized it must be told whether the shell should behave as a static or dynamic shell (default is static). As during the initialization process the shell's figure does not yet exist, the object options have to be set properly. For the static/dynamic behavior of the shell this shall be done using method *dynamic*. Using the syntax of the following calls the related control option will only be changed if it has not yet been initialized (conditionally).

```
o = dynamic(o,false);        % provide as a static shell
o = dynamic(o,true);         % provide as a dynamic shell
```

- Another task to be done during shell object initialization is setting up the refresh callback. As with other initialization tasks the following syntax provides a conditional setup of the refresh callback control option (only set if not yet initialized).

```
o = refresh(o,{@menu,'About'});     % provide refresh callback
```

  There are other calling forms for refresh. One is used to update (change) the refresh callback during execution of a menu callback in order to execute this

callback again when a shell refresh is invoked. The calling syntax forces an unconditional update of shell control settings (*control.refresh* and *control.class*)

```
refresh(o,{@Plot,mode});            % provide Plot refresh callback
refresh(o,{@simple,'Plot',mode});   % altern. way to setup refresh

refresh(o);                         % invoke refresh of graphics
```

- There is a method *manage* which is used to dispatch control to a local function if the local function's name (character string) is passed as an object argument. This leads to a common pattern for methods with dispatching properties. Typical examples are shell methods or plot methods.

```
function oo = simple(o,varargin)     % simple Espresso shell
   [func,oo] = manage(o,varargin,@Shell,@File,@Play,@Plot);
   oo = func(oo);                    % dispatch to local function
end
function oo = plot(o,varargin)       % Espresso Plot Method
   [func,oo] = manage(o,varargin,@Scatter,@Xstream,@Ystream,…);
   oo = func(oo);
end
```

A call `simple(o,'Plot','sin')` will pass the variable input argument list `{'Plot','sin'}` to method *manage* which will return function handle *@Plot* based on the first list argument `'Plot'`, which allows dispatching of program control to local function *simple>Plot*. In addition the rest of the argument list `{'sin'}` is passed as object argument list to output object *oo*. For an empty *varargin* list method *manage* returns (by default) the first function handle (*@Shell* or *@Scatter*) listed in the input arguments.

- All local functions which have function handles passed to method *manage* (like *@Shell, @File, @Play*, *@Plot*) are called published local functions, since they can be accessed from outside of the method with the following calling syntax.

```
simple(o,'Shell');       % invoke simple>Shell
simple(o);               % invoke simple>Shell (by default)
simple(o,'File');        % invoke simple>File
simple(o,'Plot','sin'); % invoke simple>Plot, with arg list {'sin'}
plot(o,'Scatter');       % invoke plot>Scatter
plot(o);                 % invoke plot>Scatter (by default)
plot(o,'Xstream');       % invoke plot>Xstream
```

- A menu item is constructed (unconditionally) with method *mitem*. It uses the input object *o* to reference the parent menu item (graphics handle stored in `o.work.mitem`) and returns an output object *oo* as a copy of input object *o,* referencing the created menu item (graphics handle stored in `oo.work.mitem`).

```
oo = mitem(o,'Stuff');                % 'Stuff' menu item
oo = mitem(o,'Sin',{@Plot,'sin'});    % 'Sin' menu item with callback
oo = mitem(o,'-');                    % separator
oo = mitem(o,'Grid',{},'play.grid'); % 'Grid' menu item w. user data
```

  Before menu items can be added the object must be initialized for menu item construction. This kind of initialization is not necessary after call of *menu>Begin* since *Begin* performs the initialization already.

```
oo = mitem(o);                        % init for menu construction
```

- An alternative method *mhead* is used to construct the menu header item for 'rebuildable' menus. It is seeking first for a menu item matching the label, and if found, deleting the whole sub menu tree, otherwise consulting *mitem* to create the menu item which could not be located during seeking. Since in both cases there is no sub-menu for the referenced menu item this method can be used to construct 'rebuildable' menus. Similar to *mitem* the input and output objects are used for referencing the graphical menu items.

```
oo = mitem(o,'Play');      % seek/construct 'Play' menu item
```

- There is a simple approach to construct a menu item with *check* functionality. The approach requires first a shell setting to be (conditionally) initialized, which the check menu item will control. After that *mitem* is used to create a menu item with user data containing the tag of the shell setting to be controlled. Applying method *check* to this menu item sets up the check functionality.

```
setting(o,{'play.grid'},false);      % no grid by default
ooo = mitem(oo,'Grid',{},'play.grid');
check(ooo,{});                        % add check functionality
```

  There are different modes of setting up a refresh callback.

```
check(o)                    % no refresh callback
check(o,{@MyRefresh})       % use custom refresh callback
check(o,{@refresh})         % use standard refresh callback
check(o,{})                 % same as above (short form)
```

- Setting up a choice menu is comparably simple. It also requires to (conditionally) initialize a shell setting to be controlled by the choice menu. After that *mitem* (or *mhead*) is used to create a menu with user data containing the tag of the shell setting to be controlled. After preparing a *choices* list method *choice* is used for completing the choice menu and setting up proper *choice* functionality. In analogy to *check* different callback modes can be setup.

```
setting(o,{'play.color'},'r');      % default setting for color
ooo = mitem(oo,'Color',{},'play.color');
choices = {{'Red','r'},{'Green','g'},{'Blue','b'}};
choice(ooo,choices,{});             % add choice functionality
```

This should be enough for this chapter. We introduced an alternative shell, called the *Simple* shell for the espresso object. Method *simple*, which constructs and controls the shell, consists of about 40 lines of code. We got in touch with a bulk of *Carabao* methods, and each of this methods has its own power, especially if it is used consistently with the other methods. That's why it took us almost 40 pages to describe the use of those methods, roughly the same number as the number of code lines in *simple.m*.

In this chapter we were covering shell hard core stuff. Once you are able to digest this hard core stuff you will have reached the next level of *Carabao* programming skills, which will be an effective basis for rapid prototyping of powerful MATLAB® shells.