# The Tag Property

Let us now discuss the *tag* property. For non-expert programming level the *tag* property is not of special importance and we will not cover the tag property in further sections and chapters. For completeness, however, and for the interested reader, we will present an overview of all essential aspects of the tag property[[1]](#footnote-1) in this section.

The *tag* property holds the class name at the time of object construction. Later on we will learn more about the important concept of *object packing* (into a 'bag' structure), which leads to a data representation that is free of any objects, which is used for mass storage representations. The tag information, holding the class name, enables the Carabao method *construct* to reconstruct any object from a (packed) bag structure.

>> bag=pack(o)

bag =

tag: 'carabao'

type: 'shell'

par: [1x1 struct]

data: {[1x1 struct]}

work: [1x1 struct]

>> o = construct(carabao,bag);

The *tag* property plays also an important role for casted objects which can be re-casted into the original class using the *balance* method. E.g. there is a superclass *caracow* which can be used for demonstration of a cast.

>> cow=caracow(o)

CARACOW object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'Carabao Shell (14-Dec-2015)'

comment: {'tiny Carabao animation shell'}

date: '14-Dec-2015'

data: [1x1 carabao]

WORK Property:

opt: [1x1 struct]

arg: {1x0 cell}

figure: 1

Applying the *caracow* constructor to a carabao object will cast the object to a *caracow* class. Using casting will only change the object class, but will not change the object properties. After a cast the *tag* property ('carabao') will differ from the class name ('caracow'). Such kind of object status is called *unbalanced*. Based on the unchanged *tag* property a common method *balance* can bring any casted object back into balanced state, which will be used internally by some Carabao methods.

>> o=balance(o) % alternatively try: o = construct(carabao,pack(o))

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'Carabao Shell (14-Dec-2015)'

comment: {'tiny Carabao animation shell'}

date: '14-Dec-2015'

data: [1x1 carabao]

WORK Property:

opt: [1x1 struct]

arg: {1x0 cell}

figure: 1

# The Type Property

Let's spend some words on the *type* property. The object *type* is important for two reasons:

1. Object data interpretation depends on the object type. Therefore class methods which operate on object data have always to check the *type* for proper data interpretation.
2. The type determines the default method to launch a shell. We will learn more about shell launching in a separate section

Permanent object information is either stored in the parameter (*par*) property or in the data property. Which rule tells us whether a specific object information is a parameter or data information? The rule is as follows:

1. If the interpretation of the information depends on the object type the information should be stored in the *data* property.
2. If the interpretation of the information does not depend on the object type it should be stored in the *par* (parameter) property.

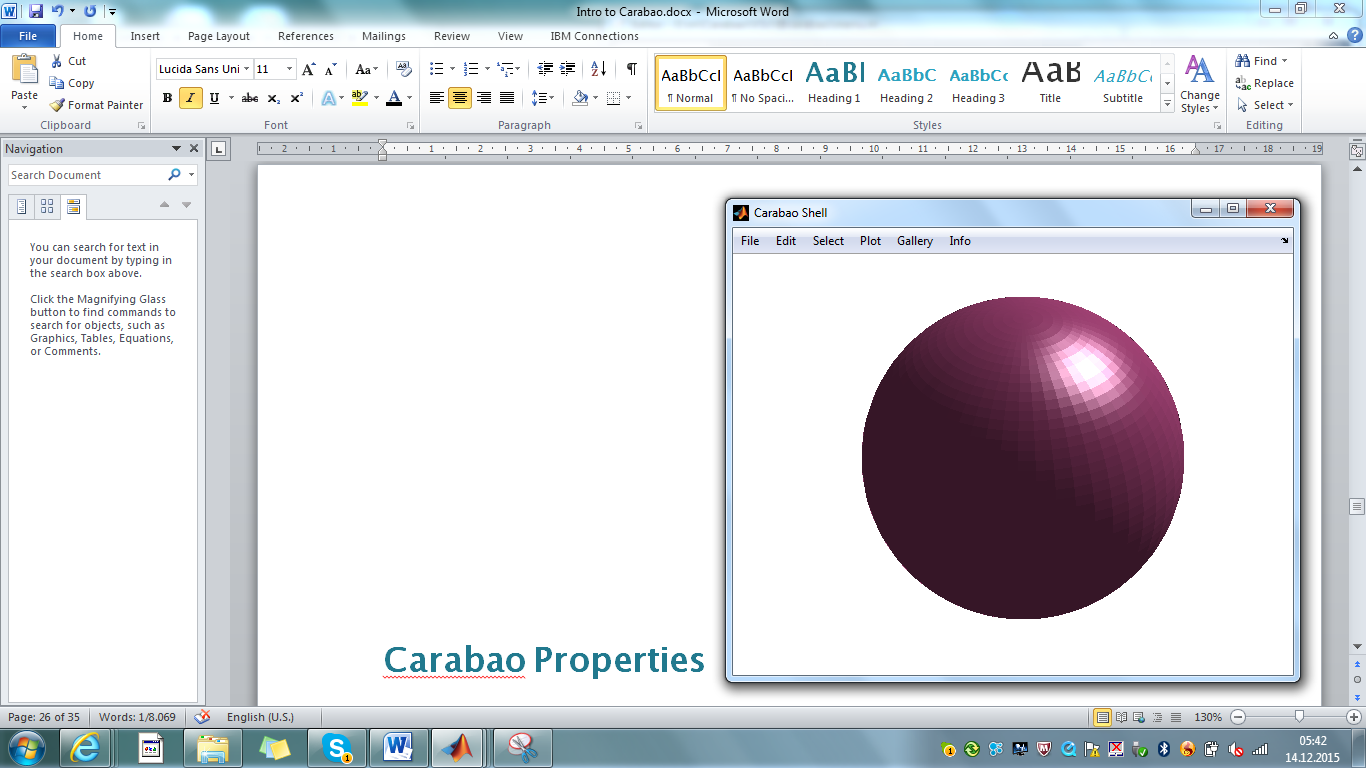


Fig.14 - A ball object, created and displayed with *File>New>Ball*, *Plot>Show*

In the mean time we know already how to study the object internals.

>> o=pull(carabao); oo = o.data{1} % pull package & access child CARABAO object

MASTER Properties:

tag: carabao

type: ball

par:

title: 'Ball (14-Dec-2015 05:56:02)'

color: [0.7000 0.3000 0.5000]

data:

radius: 1.0069

WORK Property:

opt: [1x1 struct]

arg: {}

var: []

While with the previous weird object the type had the value 'weird' the object under study has now type value 'ball', and the supported parameters are again *title* and *color*, which we can assume to have the same interpretation as with the weird object. The difference can be found in the *data* property where in contrast to data fields *t, w, x, y, z* for the weird object we have now the only data field *radius* for the ball object. It is obvious that the two data settings must be interpreted in a different way, and actually the Carabao shell's local *Draw* function must do some dispatching on the object type[[2]](#footnote-2).

function Draw(o,t,idx) % Draw an Object

color = get(o,{'color',[0 0 0]}); % get color (default [0 0 0])

**switch o.type**

case 'ball'

[X,Y,Z] = sphere(50);

Surf(o,X,Y,Z,color);

case 'cube'

[X,Y,Z] = cylinder([0 1 1 0],4);

Z(2,:) = Z(1,:); Z(3,:) = Z(4,:); Z = (Z-0.5)\*sqrt(2);

Surf(o,X,Y,Z,color);

case 'weird'

Weird(o,t,idx); % plot weird object

otherwise

menu(o,'Home');

return

end

end

# The Parameter Property

The parameter property stores all permanent object information which has not type dependent interpretation. As already mentioned it is easy to access a parameter property. It is also easy to add additional parameters, e.g. with the following statements an additional parameter *date* is added to the object parameters, and the *title* parameter is changed in order to contain also date information.

>> o.par.date=datestr(now,'dd-mmm-yyyy');

>> o.par.title = ['My package object from ',o.par.date];

>> o

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'My package object from 15-Dec-2015'

comment: {'tiny Carabao animation shell'}

date: '16-Dec-2015'

data: [1x1 carabao]

WORK Property:

opt: [1x1 struct]

arg: {1x0 cell}

figure: 1

Since the parameter property is always a structure the get/set operations can also be executed on base of symbolic indexing.

>> symbol='title';

>> title=o.par.(symbol);

>> o.par.(symbol)=title;

There is an additonal possibility to access object parameters using the *get/set* methods.

>> title=get(o,'title'); % same as title=o.par.title

>> o=set(o,'title',title); % same as o.par.title=title

Note that for Carabao objects[[3]](#footnote-3) all objects are passed by value, thus *set* returns an updated object which has to be assigned to the object variable *o*. The methods *set* and *get* have some nice extra functionality which are very useful if default values have to be provided when some object parameters are not initialized. The usual get syntax returns an empty value if the object parameter is not available

>> time=get(o,'time') % this would crash: time=o.par.time

time =

[]

and the frequently used syntax >> value=get(o,{symbol,value}) allows to provide a default value if the parameter is either empty or not available.

>> time=get(o,{'time',datestr(now,'HH:MM:SS')})

time =

06:18:42

A similar syntax >> o=set(o,{symbol},value) allows for conditional parameter setting where the parameter update happens only if the actual parameter setting is empty or the parameter does not exist.

>> o=set(o,{'time'},datestr(now,'HH:MM:SS'));

# Pushing and Launching a Carabao Object

see

>> oo.par.color=[1 0 0]; % change color parameter of child object

>> o.data{1}=oo; % store modified child in package

>> push(o); % push package object back into shell

Let us launch another shell for our modified *carabao* object. Last time we used the method *shell* for shell launch.

# References

[1] *Stormy Attaway*: MATLAB® – A Practical Introduction to Programming and Problem Solving (3rd edition); Butterworth-Heinemann, Elsevier Inc. 2013, ISBN: 978-0-12-405876-7

[2] MATLAB® – Object-Oriented Programming – R2015b; Mathworks, online on the internet

1. the non-interested reader can skip this section without loss of essential basics [↑](#footnote-ref-1)
2. the interested reader might look-up the code by invoking >> edit carabao/shell and proceeding to the definition of local function *Draw* [↑](#footnote-ref-2)
3. so called handle objects would not require an update of the object variable in order to change an object property, i.e. set(o,'title',title) would change a property of *o* - see [2] [↑](#footnote-ref-3)