Welcome back everybody and we wish you lots of energy and reflection this year for your master!

Before Xmas we gave you a little task to look into and figure out by today. The task wasn't easy. On top of that, I discovered quite late that one important method to store and retrieve objects has not been explained yet. I am sure it is possible to find it out by yourselves but also hard to get your head around. So, this hand-out shows two methods of collecting object data and accessing it at a later stage for maniputation. The little programme for today includes both methods: *selection sets* and *array storage*.

Selection Sets *(well described in the Help files)*

In AutoCad there are two different ways of collecting objects - into *collections* and into *selection sets*. Today we will look at the selection sets since they are the everyday method to handle collections of objects.
As the name suggests, a selection set is a set of specified objects that you selected. Hence, it is obvious that there should be modes of selecting objects from the drawing. In other words there are various ways of filtering the drawing for objects.
Before one can select anything from the drawing and store that selection into a set, one must declare and dimension a selection set like any other variable:

```
Dim ss As AcadSelectionSet
```

After declaring it, as usual it has to be instantiated into the drawing database with its name:

```
Set ss = ThisDrawing.SelectionSets.add("it")
```

Now you would be ready to use the selection set and populate it with objects from the drawing database.
You must now chose between 4 methods of selection, of which we regularly use two:

```
ss.Select and ss.SelectbyPolygon
```

Each methods requires given parameters as arguments that define the selection mode. The general *.Select* method has 5 modes of filtering for data. In our little programme we use the *acSelectionSetAll* mode, which can be given some filtering parameters (none of them has to be used). We use the third (group code) and fourth (type code):

```
ss.Select acSelectionSetAll, , , gp, dv
```

ss.Select means we generally select objects, using no auxiliary methods like a boundary (SelectByPolygon) or interactively picking (SelectOnScreen). If one specifies the mode - here *acSelectionSetAll* - some arguments must be passed to say what to look for. *gp* is an array of integers with only one element that indicates what general family of objects we are looking for.

```
Dim gp(0) As Integer
gp(0) = 8
```

gp stands for group-code. group codes are coded as DXF codes and indicate to the assembler what kind of object should be looked for in the database. A range of DXF group-codes are

```
0     entity type
2     object name
5     handle
8     layer
10    startpoints/ endpoints
62    colour
```

Within the group, one has to specify the exact description of the type of objects to be selected. One does that with the data-value dv(). Even the data-value must be declared as an array of variants. Variants because it could be any type of data, in our example the name of the layer we would like to select from:

```
Dim dv(0) As Variant
dv(0) = name
```

The value of the data must be absolutely precise otherwise it won't select the right objects. We select object from a layer. The layer name is easy because we specified it ourselves. But say, we are filtering for circles in the drawing. Then one has to define the group-code to be

```
gp(0) = 0
dv(0) = CIRCLE
```

In order to find the right description, you can create an object manually on screen, select it and type *list* on the command line. That will give you the exact name of the object.

It seems tempting to filter for more than one group and type in the arrays but unfortunately AutoCad never fixed that bug so far and it doesn't work yet - so don't even try!

So now we have filtered through all object on a layer called whatever is stored as a string in the variable *name*.
Since the purpose of the sub-procedure *wipe_layer()* is to erase all objects found on a given layer, we have to iterate through the selection set and erase one by one:

```
For i = 0 To ss.count - 1
      ss.Item(i).Delete
Next i
```

If the selection set has collected any objects, its Count should me higher than 0. Infact, if there are no objects in the set, the count will be <empty>.
To access the objects, we use the ***Item(index)*** property that is also valid for other types of collections. All one needs to do is to loop through as many objects there are in the set and access them via the Item() property, and finally indicate what to do with it. We want to delete the items, so we chose the method *.Delete* of the Item() object.

Eventually, to clear up the memory used for the selection set, we delete it after having finished:

```
ss.Delete
```

You could also use the selection set to retrieve the circles and manipulate them. In order to do so, you have to extract and instantiate the circle:

```
set a_circle = ss.Item(i)
```

That way you can investigate and use the circle, manipulate it and delete it again.

As you can see in the friends sub-procedure, *sel_set_del*, you can ask the drawing how many selection sets there are by using the .Count property again:

```
ThisDrawing.SelectionSets.count
```

We do sel_set_del first because if any selection set of the same name existed already in the drawing, you get a run-time error. Thus, we collect all the selection sets in the drawing and delete them:

```
If (ThisDrawing.SelectionSets.count > 0) Then
      ThisDrawing.SelectionSets.Item(0).Delete
End If
```

ObjectID & ObjectIDToObject properties

Another method of collection any kind of object in a light and minimal fashion, is to store the identification numbers that AutoCad attributes to drawing objects. It is a very light and practical method, because one can determine the order in which objects are stored and we don't store all details about the object but just their identification numbers. When later, for the CA for example, one wanted to store and access objects in an orderly fashion, it becomes almost impossible with a selection set, since a two dimensional or three dimensional order becomes more feasable.
So, how to get that identification, store it and get it out later again. Consider out example of the circles in an orthogonal grid. Each circle we draw should be stored directly in an array:

```
nodes(count) = ball.ObjectID
```

That's how easy it is to access the object identification number. Notice two things - firstly, ball must be an AutoCad object and *Set*; and secondly, the ObjectID property is of data type *Long*. That means that your array nodes() must be dimensioned as:

```
Dim nodes() As Long
```

The data type Long occupies 4 bytes of memory and therefore can contain much larger numbers than an integer. The identification numbers of objects are greater than an integer.
Now that you have stored all the object numbers in an array, you can at a later stage retrieve the whole object again with all its data (as the Romans proverb goes: 'pars pro toto', meaning one little part describes the whole). To retrieve the object one has to use the *ObjectIdToObject()* method, which is subject to the drawing, not necessarily to modelspace. But beware, the return type is an object in our case and therefore, the drawing object needs to be *Set* again:

```
Set obj = ThisDrawing.ObjectIdToObject(ObjectID)
```

The ObjectID in brackets is one of the Longs that we have stored in the array nodes(). Therefore, we could retrieve one of those objects by asking for say the second circle:

```
Set winner = ThisDrawing.ObjectIdToObject(nodes(2))
```

Winner has to be declared as an AcadCircle first:

```
Dim winner As AcadCircle
```

Now, you have retrieved the original second circle and can access all its methods and properties. In our little programme we scale it and take out the next circle.

... and finally some (bubble) sorting: