

# Keeping Up With 4D's Latest Technologies — A Worthwhile Endeavor (Why?)

# Ground-shaking announcement v17!

## OBJECT RELATIONAL DATA ACCESS



### Keynote by Thomas Maul

Entirely new programming paradigm

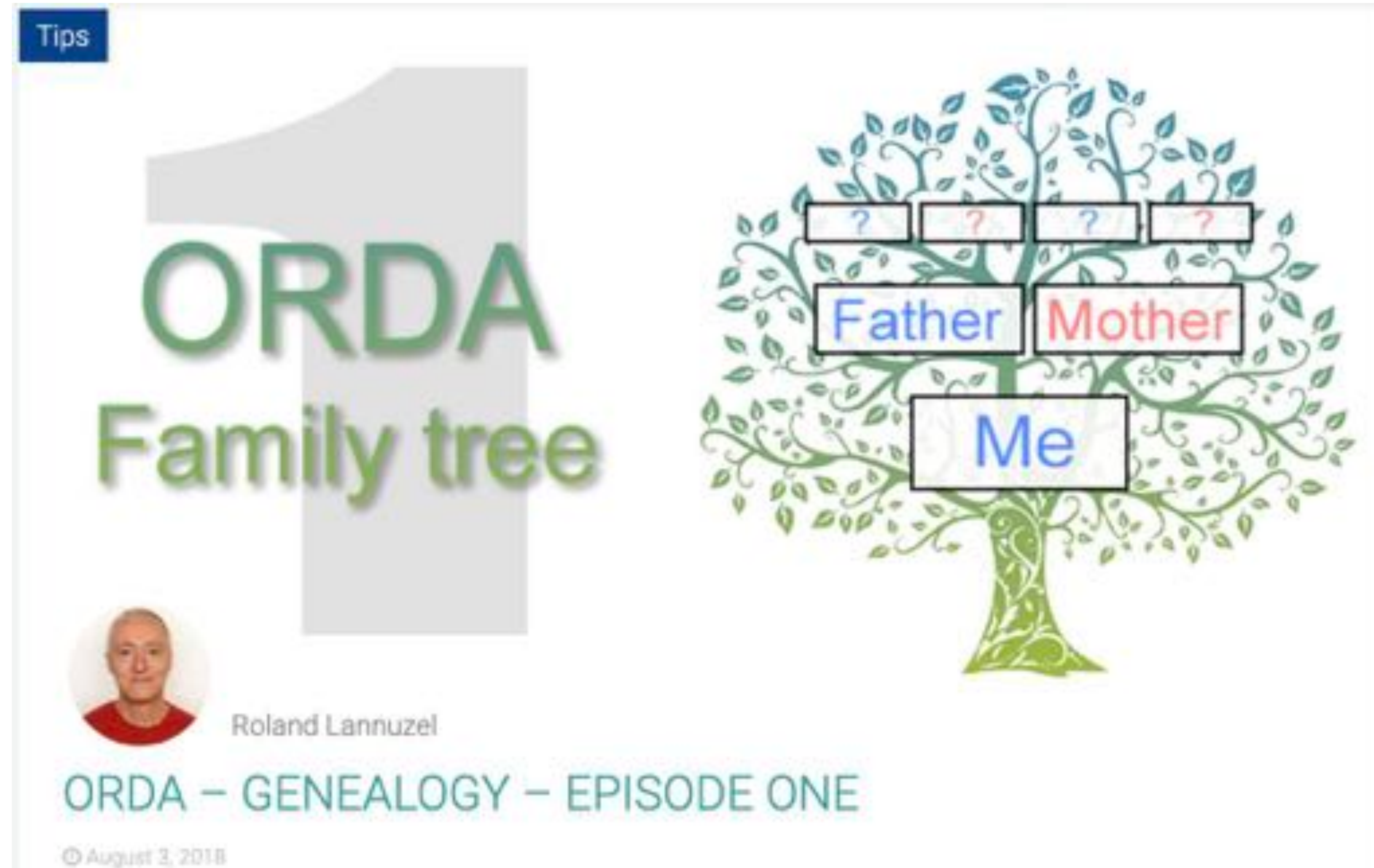
- Entire database is an object!
- Multiple relations between DataClasses
- Entities replace 'current record'
- Entity Selection replaces 'current selection'
- DataSource for list boxes and other objects
- Modern Code Design
- No more 'Push Record' & 'Pop Record'

followed by a reveal and 'plan' presentation:

### Master Class by Laurent Ribardière

“Genealogy” Series of HDI's by  
Roland Lannuzel

# “Landmark” Blog Post Series



- These HDI's helped “light the fire” for me to get deeply into the new features.

# My Reaction ... with excitement...

- Initially ‘experimented’ with v17.0 — designing a personal-interest app.
- Contacted by a company with interest in an app
- Rather than ‘retro-fitting’ an existing app of mine — with the blessing of the customer — I got started designing an entirely new application.
- Salvaged some methods from past ‘toolbox’ and organized into a component I call “Colonel”
- Committed myself to truly acquiring a working knowledge of v17.

# My code base is completely '4D-modern'. It contains no...

- ~~ARRAYS~~ (except for processing 4D commands that require them)
  - Collections of objects replace groups of arrays
- ~~CLASSIC 4D CODE~~ (other than in triggers, where it is *still* necessary)
- ~~PROCESS VARIABLES~~ (I have 3 process variables out of necessity)
  - **Form.** object negates the need for any process variables. As a result, forms can peacefully coexist in the same process without fear of 'data collisions'
- ~~INTERPROCESS VARIABLES~~
  - **Storage.** and **Shared Objects / Collections** negate need for interprocess vars.

# Since, v17, I have:

- **Written 'Colonel' Component**
  - Collection & Object support methods
  - Form Object
  - Text functions ...
- Made extensive use of subforms
  - ListBoxes (LB, LB\_LIST)
  - Editor Button Panel (standard interface for editors)
  - Browser Button Panel (standard interface for list boxes)  
(to me, subforms became useful with **Form.**)
- Eliminated use of process, interprocess variables (**Form.**)
- **No Arrays or Pointers** (unless there is no other way)
- Changed to Project Mode
- DataStore, DataClass, Entity, EntitySelection Classes

**MOTTO: First choice for implementation: CLASSES!**

## WRITTEN CLASSES:

- Handle UI-related functions, like:
  - Combo-boxes
  - Finder widget
  - 'IS Forms'
  - Menus
  - Button Groups

## 'SINGLETON' OBJECTS:

- App
- MenuBar Manager
- Window Manager
- Switchboard (runs in worker process)

## OTHER:

- Document Librarian
- Various Schedulers
- Cache Manager (in progress)

# My Standard Design Policies

- I hate ‘spaghetti code’ — strong commitment to **GENERIC PROGRAMMING**.
- **SIMPLICITY brings in POWER**. If something I am working at starts to feel too complicated, I know that there is something inherently wrong about how I am going about it. Once I get the sense it is ‘simplified’ then it is a much better direction than I was initially taking.
  - On a number of occasions, I have ‘polled’ the 4D dev community through the Forum and invariably have received tips and pointers that greatly simplified the process. THANK YOU!
- **MODULARITY** — strong emphasis on ‘plug-and-play’ programming. If a subsystem is written correctly, it should just ‘plug into’ other parts of the system.

# SIMPLIFICATION THROUGH DATA POLYMORPHISM

## ...BY LEVERAGING OBJECT FIELDS

One of the first 'Subsystems' I built in my new 'Code Base' was intended to clarify objects I would deploy in my code.

So I have built what I call **UI DESIGNER**. Let me demo it for a few minutes.

I (Designer) use it to configure such things as:

System & Contextual Menus; Lists ('choice lists' and 'orda lists'); Data 'Rules'; object construction in various Object fields in my DataClasses.

# HOW I USE UICLASS & UIOBJECTS

**UIClass** contains 'definitions' of object structures in `._Attrs{ }`

**UIObjects** contains 'instances' of **UIClass**.

• Hierarchical structure from the Parent  $\leftrightarrow$  Children relation



Certain classes of **UIObjects** are either directly or 'analysed' to produce items for **Storage** or **Controller Objects**

example: In **Storage** — Lists; in **cs.App.me.MB** — system menu bar, etc.

# UIClass Function — getBlankObj

```
Function getBlankObj($name : Text) : Object // get a BLANK object defined from this class
    $en:=This.getUIClassEntity($name) // get the CLASS entity
    $O:=New object
    For each ($attr; $en.Attrs.col) // process the collection of attributes for this class
        Case of
            : (($attr.attrType="Text") | ($attr.attrType="String") | ($attr.attrType="Alpha"))
                $O[$attr.name]:= ""
            : ($attr.attrType="Boolean")
                $O[$attr.name]:= Bool($attr.default)
            : (($attr.attrType="Long@") | ($attr.attrType="Int@") | ($attr.attrType="Real") | ($attr.attrType="Float"))
                $O[$attr.name]:= 0
            : ($attr.attrType="Time")
                $O[$attr.name]:= Time(0)
            : ($attr.attrType="Date")
                $O[$attr.name]:= Date("")
            : ($attr.attrType="Collection")
                $O[$attr.name]:= New collection // should we recursively go further and create the collection's objects? probably not? — ** CB 08/24/19
            : ($attr.attrType="Object")
                $O[$attr.name]:= New object
        Else
            $O[$attr.name]:= ""
    End case
End for each
```

# UIClass Function — getDefaultObj

```
Function getDefaultObj($name : Text) : Object // get a DEFAULT object defined from this class
    $sen:=This.getUIClassEntity($name) // get the CLASS entity
    $0:=New object
    For each ($attr; $sen.Attrs.col) // process the collection of attributes for this class
        Case of
            : (($attr.attrType="Text") | ($attr.attrType="String") | ($attr.attrType="Alpha"))
                $0[$attr.name]:=String($attr.default)
            : ($attr.attrType="Boolean")
                $0[$attr.name]:=Bool($attr.default)
            : (($attr.attrType="Long@" ) | ($attr.attrType="Int@" ) | ($attr.attrType="Real") | ($attr.attrType="Float"))
                $0[$attr.name]:=Num($attr.default)
            : ($attr.attrType="Time")
                $0[$attr.name]:=Time($attr.default)
            : ($attr.attrType="Date")
                $0[$attr.name]:=Date($attr.default)
            : ($attr.attrType="Collection")
                $0[$attr.name]:=New collection // should we recursively go further and create the collection's objects? probably not? —— **** CB 08/24/19
            : ($attr.attrType="Object")
                $0[$attr.name]:=New object
        Else
            $0[$attr.name]:= ""
    End case
End for each
```

# UIObjects Functions

**Function allOfClass(\$name : Text) : cs.UIObjectsSelection**

**\$ID:=ds.UIClass.getUIClassID(\$name)** // this is the ID of the \$name class

**\$0:=ds.UIObjects.query("Class = :1"; \$ID).orderBy("Name")** // return all the cities, sorted by NAME

**Function lookup(\$class : Text; \$name : Text) : cs.UIObjectsEntity** // look up the record of CLASS \$class, with NAME \$name

**\$ID:=ds.UIClass.getUIClassID(\$class)** // this is the UUID of the \$name class

**\$0:=ds.UIObjects.query("Class = :1 AND Name = :2"; \$ID; \$name).first()** // this should be a single entity

# Sample — How .getDefaultObj() used

IN THE 'JOB ENTITY' CLASS:

```
local Function initConfig($seed : Object) : cs.JobEntity // set up the initial state of a new record. Called by the dataClass .newEntity()  
  UUID_Get(This) // this will ASSIGN a new UUID  
  This.CallDate:=Current date  
  This.CallTime:=Current time  
  This.Status:=ds.UIClass.getDefaultObj("JobStatus")  
  This.Status.isOpen:=True  
  This.Schedule:=New object // ensure we have a 'blank' schedule!  
  This.DailyJobDetails:=New object // a new BLANK object is required for storing the Daily Job Details  
  This.Branch:=Storage.env.company.branch // record the 'branch'  
  OBJ_SetAttrVals(This; $seed) // if $seed is null, then nothing will change  
  $0:=This // return so it can be chained if desired
```

# TRANSITIONING FROM PROJECT METHODS TO USING 4D's STANDARD DATA - RELATED CLASSES

(i.e. DataStore, DataClass, EntitySelection, Entity)

**Example of transition from Project Methods  
to Class Functions:**

**Write an ORDA-Compliant  
replacement for the standard  
4D Classic Command  
APPLY TO SELECTION()**

# APPLY TO SELECTION(Table; Statement) — ORDA- compliant implementation

## AS A PROJECT METHOD **with no error-reporting, just like APPLY TO SELECTION()**

### Basic form, exactly replacing APPLY TO SELECTION

```
// APPLY_FORMULA($es : cs.EntitySelection; $formula : Object)
#DECLARE($es : cs.EntitySelection; $formula : Object)
var $en : cs.Entity
var $obj_Result : Object // result of the .save( )
For each ($en; $es)
    $formula.apply($en)
    $obj_Result:=$en.save()
End for each
```

### Support for optional 'mode' for .save( ) i.e. dk auto merge

```
// APPLY_FORMULA($es; $formula; {$mode} )
#DECLARE($es : cs.EntitySelection; $formula : Object; $mode : Integer)
// Note: If $mode is not given, 4D defaults it to 0
var $en : cs.Entity
var $obj_Result : Object // result of the .save( )
For each ($en; $es)
    $formula.apply($en)
    If ($mode#0)
        $obj_Result:=$en.save($mode)
    Else
        $obj_Result:=$en.save()
    End if
End for each
```

// In 4D 19r4: If (\$mode...) end if can be written using a Ternary ? as:  
\$obj\_Result := (\$mode#0) ? \$en.save(\$mode) : \$en.save()

# APPLY TO SELECTION() — ORDA- compliant implementation

**BUT:**

Our ORDA-compliant replacement to **APPLY TO SELECTION** *operates only on entity selections.*

So would it not be prudent if this were implemented to support the syntax:

```
$es.applyFormula($formula; $mode)
```

Then to apply a formula to a related selection of entities, we could even:

```
$en.relationName.applyFormula($formula; $mode)
```

**WHAT DOES THIS REQUIRE?**

The code must be implemented as a **cs.EntitySelection** - type function for each DataClass

Do we want to copy/paste this code into each Entity selection class? No.

So where can we place the main piece of code?

- the **DataStore** class is a logical repository for data-related functions. **HOW?**

# APPLY TO SELECTION() — ORDA- compliant implementation

## The function in the DataStore Class:

// .applyFormula( ); apply a formula to a selection of entities.

**Function applyFormula**(\$es : cs.EntitySelection; \$formula : Object; \$mode : Integer)

// Note: If \$mode is not given, 4D defaults it to 0

**var** \$en : cs.Entity

**var** \$obj\_Result : Object // result of the .save( )

**For each** (\$en; \$es)

    \$formula.*apply*(\$en)

**If** (\$mode#0)

        \$obj\_Result:=\$en.*save*(\$mode)

**Else**

        \$obj\_Result:=\$en.*save*()

**End if**

**End for each**

## Simple function in each EntitySelection class:

// .applyFormula( ); apply a formula to this selection of entities.

**Function applyFormula**(\$formula : Variant; \$mode : Integer)

*ds.applyFormula*(**This**; \$formula; \$mode)

# APPLY TO SELECTION() — ORDA- compliant implementation

BUT ... We can do better than this!

`$en.save()` returns an object result. We can get our ORDA-compliant ‘replacement’ to **return errors!**

The enhanced error-reporting function in the **DataStore Class**: (no change to EntitySelection functions)

```
// .applyFormula( ); apply a formula to a selection of records. Return with any errors from the save
```

```
// result: { .result:true } if all successful; { .result: false; .errors: collection }
```

**Function applyFormula(\$es : cs.EntitySelection; \$formula : Object; \$mode : Integer) : Object**

```
// note: if $mode is not given, 4D defaults it to 0
```

```
var $en : cs.Entity
```

```
var $obj_Result : Object // result of the .save( )
```

```
var $errors : Collection // any .save( ) that failed are returned in this collection
```

```
$errors:=New collection
```

```
For each ($en; $es)
```

```
    $formula.apply($en)
```

```
    If ($mode#0)
```

```
        $obj_Result:=$en.save($mode)
```

```
    Else
```

```
        $obj_Result:=$en.save()
```

```
    End if
```

```
    If (Not($obj_Result.success)) // if the save failed, tell the caller
```

```
        $obj_Result.en:=$en // append the entity to the error object
```

```
        $errors.push($obj_Result)
```

```
    End if
```

```
End for each
```

```
If ($errors.length=0)
```

```
    $0:=New object("result"; True)
```

```
Else // there were entities that failed the formula application
```

```
    $0:=New object("result"; False; "errors"; $errors) // return the collection of errors, with their messages
```

```
End if
```

# APPLY TO SELECTION() — ORDA- compliant implementation

One additional enhancement: what if we want to **support FORMULA STRINGS** also?

## DataStore Class function:

**Function** applyFormula(\$es : cs.EntitySelection; \$formula : Variant; \$mode : Integer) : Object

// note: if \$mode is not given, 4D defaults it to 0

**If** (Value type(\$formula)=Is text)

    \$formula:=Formula from string(\$formula)

**End if**

**var** \$en : cs.Entity

**var** \$obj\_Result : Object // result of the .save( )

**var** \$errors : Collection // any .save( ) that failed are returned in this collection

\$errors:=New collection

**For each** (\$en; \$es)

    \$formula.apply(\$en)

**If** (\$mode#0)

        \$obj\_Result:=\$en.save(\$mode)

**Else**

        \$obj\_Result:=\$en.save()

**End if**

**If** (Not(\$obj\_Result.success)) // if the save failed, tell the caller

        \$obj\_Result.en:=\$en // append the entity to the error object

        \$errors.push(\$obj\_Result)

**End if**

**End for each**

**If** (\$errors.length=0)

    \$0:=New object("result"; True)

**Else** // there were entities that failed the formula application

    \$0:=New object("result"; False; "errors"; \$errors) // return the collection of errors, with their messages

**End if**

## In each EntitySelection class:

// .applyFormula( ); apply a formula to this selection of entities.

**Function** applyFormula(\$formula : Variant; \$mode : Integer)

    ds.applyFormula(This; \$formula; \$mode)

# APPLY TO SELECTION()

**BUT WHY STOP WITH ENTITY SELECTIONS?**

Isn't this just as useful for **COLLECTIONS**?

**DataStore Class function:**

**Function applyToCollection(\$col : Collection; \$formula : Variant)**

**If (Value type(\$formula)=Is text)**

**\$formula:=Formula from string(\$formula)**

**End if**

**var \$o : Object**

**For each (\$o; \$col)**

**\$formula.*apply*(\$o)**

**End for each**

# APPLY TO SELECTION()

**BUT... Entity Selections are just ‘special collections’. Can we adjust `ds.applyFormula()` to do both? One choice (the one implemented as a sample in the demo **isWidgetsDemoApp**)**

## DataStore Class function:

```
// .applyFormula( ); apply a formula to a selection of entities OR to a collection of objects.
```

```
// for EntitySelection, returns with any errors from the save
```

```
// result: { .result:true } if all successful; { .result: false; .errors: collection }
```

```
Function applyFormula($es : Variant; $formula : Variant; $mode : Integer) : Object
```

```
// note: if $mode is not given, 4D defaults it to 0
```

```
If (Value type($formula)=Is text)
```

```
    $formula:=Formula from string($formula)
```

```
End if
```

```
If (Value type($es)=Is collection)
```

```
    ds._applyToCollection($es; $formula)
```

```
Else
```

```
.... same
```

```
Function _applyToCollection($col : Collection; $formula : Variant)
```

```
    If (Value type($formula)=Is text)
```

```
        $formula:=Formula from string($formula)
```

```
    End if
```

```
    var $o : Object
```

```
    For each ($o; $col)
```

```
        $formula.apply($o)
```

```
    End for each
```

## QUESTION:

Should `.applyFormula()` be a standard 4D function for Entity Selections and Collections?

```
$col.applyFormula($formula)
```

```
$es.applyFormula($formula; $mode)
```

Seems like a natural development, and easy to implement, too!  
But until then, you can build your own...

# SHIFTING FROM 4D CLASSIC Commands to Orda-Compliant Versions.

- Certain 4D Commands are non-ORDA compliant (i.e. return arrays instead of collections, etc.)
- Here are some sample ‘converters’ you can use (and are included in **isWidgets** component)

```
// colGetPropertyNames( OBJECT ) → COLLECTION containing the property names
var $1 : Object // if we are checking out a COLLECTION, then just pass $col[0] assuming it exists
var $0 : Collection // the collection containing the properties names for OBJECT
ARRAY TEXT($props; 0) // how the properties
$0:=New collection // we have an empty collection if there are no objects in it

If ($1#Null) // if we can even see any...
    OB GET PROPERTY NAMES($1; $props)
    ARRAY TO COLLECTION($0; $props) // convert the ARRAY to a collection!
End if
```

# 4D CLASSIC Commands → Object-Versions.

- Get Form Object's Characteristics (i.e. from Objects (Forms) Theme)

**OBJ\_GetObjCharacteristics** ( Object Name ; {\$choice} ) → infoObject, return in an object:  
{ .foreground .background .altBackground .action .left .right .top .bottom .height .width  
.font .fontSize .fontStyle .displayFormat .entryFilter .styleSheet } — depending on \$choice

**\$choice:** these are constants defined using 4D Pop Constants Editor and contained in the isWidgets resources folder.

_OB_All	
_OB_RGB	.foreground, .background, .altBackground
_OB_Action	.action
_OB_Coordinates	.left; .right; .top; .bottom; .height; .width
_OB_Font	.fontFamily; .fontSize; .fontStyle
_OB_Format	.displayFormat
_OB_Filter	.entryFilter
_OB_StyleSheet	.styleSheet
_OB_Spellcheck	.spellcheck
_OB_BorderStyle	.borderStyle
_OB_ContextMenu	.contextMenu
_OB_CornerRadius	.borderRadius
_OB_DragDrop	.dragging; .automaticDrag; .dropping; .automaticDrop
_OB_Enabled	.enabled

_OB_Enterable	.enterable
_OB_Events	.events
_OB_FocusRectInv	.focusRectInvisible
_OB_HelpTip	.helpTip
_OB_Alignment	.horizAlign; .vertAlign
_OB_ListNames	.choiceList; .excludedList; .requiredList
_OB_ListRefs	.choiceListRef; .excludedListRef; .requiredListRef
_OB_MaxMin	.max; .min
_OB_MultiLine	.multilineLine
_OB_Placeholder	.placeholder
_OB_ResizeOpts	.resizeHoriz; .resizeVert
_OB_Shortcut	.shortcut; .modifiers
_OB_StyledText	.styledText

Complimentary method is:

**OBJ\_SET\_OBJ\_CHARACTERISTICS** ( Object Name ; Object ) → set Object characteristics from object's attributes

# 4D CLASSIC Commands → Object-Versions.

- Get Form Object's Characteristics (i.e. from Objects (Forms) Theme)

**OBJ\_SET\_OBJ\_CHARACTERISTICS** ( Object Name ; Object ) → set Object characteristics from object's attributes

**choicesLongint:** these are constants defined using 4D Pop Constants Editor and contained in the isWidgets resources folder.

_OB_All	
_OB_RGB	.foreground, .background, .altBackground
_OB_Action	.action
_OB_Coordinates	.left; .right; .top; .bottom; .height; .width
_OB_Font	.fontFamily; .fontSize; .fontStyle
_OB_Format	.displayFormat
_OB_Filter	.entryFilter
_OB_StyleSheet	.styleSheet
_OB_Spellcheck	.spellcheck
_OB_BorderStyle	.borderStyle
_OB_ContextMenu	.contextMenu
_OB_CornerRadius	.borderRadius
_OB_DragDrop	.dragging; .automaticDrag; .dropping; .automaticDrop
_OB_Enabled	.enabled

_OB_Enterable	.enterable
_OB_Events	.events
_OB_FocusRectInv	.focusRectInvisible
_OB_HelpTip	.helpTip
_OB_Alignment	.horizAlign; .vertAlign
_OB_ListNames	.choiceList; .excludedList; .requiredList
_OB_ListRefs	.choiceListRef; .excludedListRef; .requiredListRef
_OB_MaxMin	.max; .min
_OB_MultiLine	.multilineLine
_OB_Placeholder	.placeholder
_OB_ResizeOpts	.resizeHoriz; .resizeVert
_OB_Shortcut	.shortcut; .modifiers
_OB_StyledText	.styledText

Complimentary method is:

**OBJ\_GetObjCharacteristics** ( Object Name ; {\$choice} ) → infoObject

# 4D CLASSIC Commands → Object-Versions.

**OBJ\_GetWindowInfo** ({winRef}) → windowInfoObject { .top .left .bottom .right .title .kind .process }

**OBJ\_SetWindowInfo** ( windowInfoObject ) → based on \$1.winRef, sets { .top .left .bottom .right .title }

**FORM\_Get** ( requestConstant; { \$formName ; {\$pageNo} } ) → Variant — This is use to get information on an ACTIVE FORM

Examples:

**FORM\_Get** ( \_FO\_EntryOrder; {pageNo} ) → [ ObjectNames ]

**FORM\_Get** ( \_FO\_Resizing ) → { .horiz: { .resize; .min; .max } ; .vert: { .resize; .min; .max } }

**FORM\_Get** ( \_FO\_Objects; {PageNo} ) → [ .name; .ptr; .pageNo ]

...And on a 4D Form:

**FORM\_Get** ( \_FO\_Properties ) → { .name; .width; .height; .numPages; .fixedWidth; .fixedHeight; .title }

**FORM\_Get** ( \_FO\_ScreenShot; \$formName; {\$pageNo} ) → Pict

# BROWSERS AND EDITORS

These was implemented in v17 and hence does not use Classes :(

This leverages the use of subForms. I never used subForms in 'Classic 4D' because of the issue of project variables. But with the **Form.** object, subForms became extremely convenient and powerful in my UI.

**MODULARITY.** Tried to make it as plug-and-play as I could.

**STANDARD.** The Browsers and Editors 'know' their job; special 'instructions' can be provided as needed, though.

**SIMPLIFICATION.** In 'Classic 4D', I tended to deploy Browsers and Editors in cooperative processes because of the complication of having to use process variables, lots of arrays, and so forth. They would 'communicate' when they 'handed off' control, providing current selection, current record, window size and position, and so on.

# BROWSERS AND EDITORS



Combined onto single 4D Form



Matching Browsers and Editors share same **Entity Selection** and, therefore, operate on the same entities

**DURING THE FORM's 'ON LOAD' handler**, the Browsers, Editors, UI Widgets and so on are configured.

- The configuration commands build the **Form**. objects that are used to operate **Browsers** and **Editors**.

EXAMPLE SETUP: edit\_Contacts ...

**BROWSER CONFIGURATION** is performed in the **On Load** of its container object.

**LB\_LOAD** ( ... ) is provided the default columns (for initial configuration) and other specs to automatically operate.

**BROWSERS** are always one of two subforms containing a listBox:

**LB**: 'main' listBox

**LB\_LIST**: 'related records' listBox

## CONFIGURATION

- stored in a [UIObjects] Entity of UIClass **Browser**
- GUI-configured by Designer (me)
- can be 'defaulted', but that is more of a starting point
- to configure a **Browser**, I start the form and configure using GUI.

**EDITORS** can be

- 1) a special page on the 4D Form (i.e. page 1)
- 2) a subform that can appear on any page (including same page as its browser)

An **EDITOR** can 'control' any number of Browsers (**LB\_LIST**), which also can have their own associated **EDITORS**. These Browsers are used for related entity selections (ex. 'invoices' of a 'customer')

- An **EDITOR** can also control UI - 'widgets' such as comboBoxes & Date Entry / Picker.
  - These widgets edit a single [Entity]Attribute or [Entity]Attribute.attr
  - The **EDITOR** automatically refreshes these UI widgets when loading an entity for editing.

**ALL EDITOR CONFIGURATIONS** are performed in the **On Load** of the **FORM**.

**ED\_CONFIGURE** ( ... ) is provided details as to which **Browser** 'owns' the editor, valid buttons, browsers it controls (for related records), GUI objects it manages, etc.

# BROWSERS

## BROWSER FEATURES : GUI-BASED CONFIGURATION

Configuration is done using **LB\_Decorator** as a subform



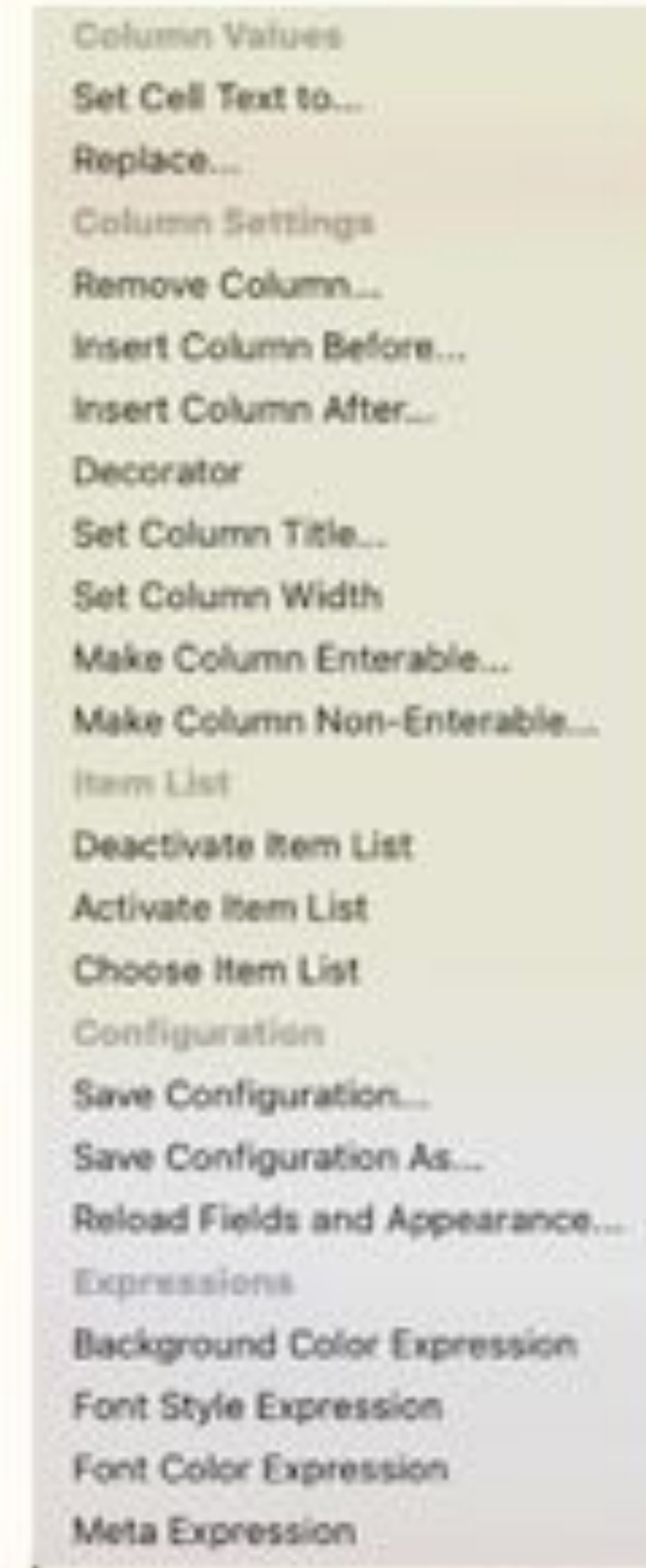
The configuration is stored in a [UIObjects] 'Browser' - class record  
configuration changes are **SAVED** by choosing 'Save Configuration...'

\* Demo how to change configuration in a LB and a LB\_LIST

- \* With edit\_Companies, demo turning on a choice list (City) (which will require 'Make Column Enterable' as well)
- Demo adding a column, removing a column, setting the column title.
- Demo **Enterable** on the booleans

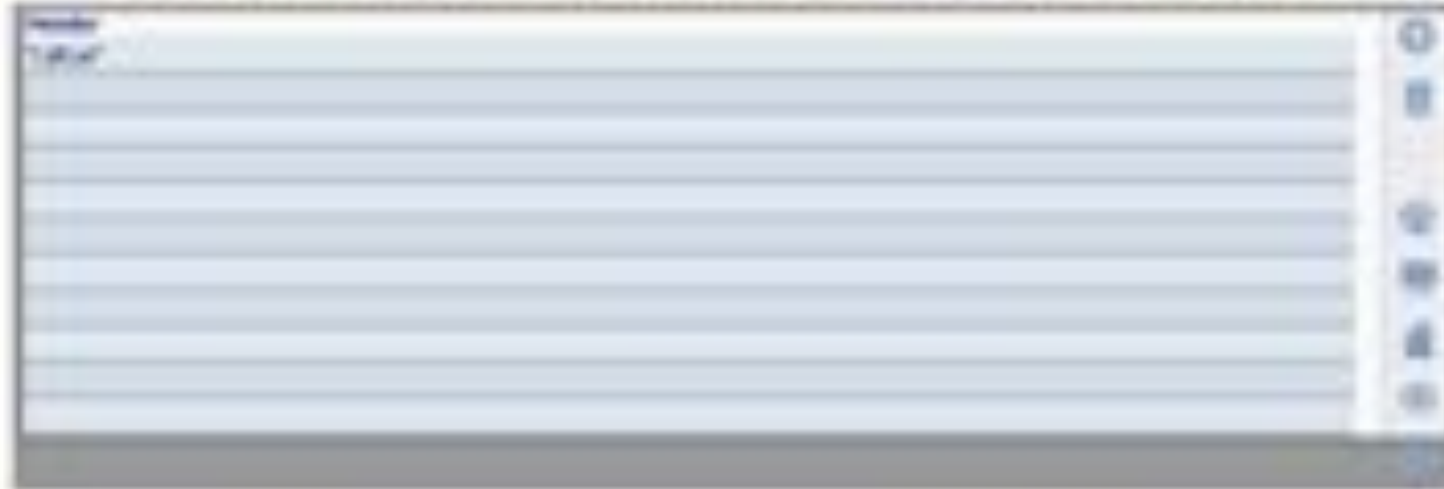
\* **Extra goodie:** The Client 'remembers' the position of main windows and uses that new position next time it is opened. This is done using a nice little mechanism I built into **ds.DataStore**

Certain characteristics are more simply defined using the popup menu options (which only 'appear' for the Designer (me) )



# BROWSERS

LB\_LIST (intended for related records), and 'owned' by an EDITOR



Standard button panel, saved in the UIObjects entity for the browser  
Configured using a ca.ButtonGroup object according to the specs

## EVENT MESSAGE CONSTRUCTION

A single event is sent: `_msgPanelEvent`.

But in its `Form.msg` it provides additional details for the container object

**BUTTONS** (not all shown; but you see the messages are very simple):

NEW RECORD: { .event: `_msgGoto` ; .recNo: `_msgNewRecNo` }

DELETE RECORD: { .event: `_msgDelete` } — Note: The Container Object already 'knows' which ones because they are `.es_Selected`

PRINT: { .event: `_msgPrint` ; .specs { details } }

DISPLAY: { .event: `_msgDisplay` }

PDF: { .event: `_msgPDF` }

## SAMPLE BUTTON OBJECT METHOD:

`Self->:=0` // reset appearance of this picture button.

`Form.msg:=New object("event": _msgPDF)`

`CALL SUBFORM CONTAINER(_msgPanelEvent)`

**LB\_LIST** can handle many matters directly.

It is also configured using GUI, like **LBs**

The **LB\_LIST**, because it *always* functions as a subform, uses events to tell its Container Object of anything that needs the attention of the Container Object.

## EVENT MESSAGE CONSTRUCTION

### EVENTS GENERATED BY INTERACTION WITH THE LISTBOX

GOTO RECORD: { .event: `_msgGoto` ; .recNo: Integer (index) }

SINGLE CLICK: { .event: `_msgClicked` ; .recNo; .rowNo; .colNo }

SELECTION CHANGE: { .event: `_msgSelect` ; .rowNo; .colNo }

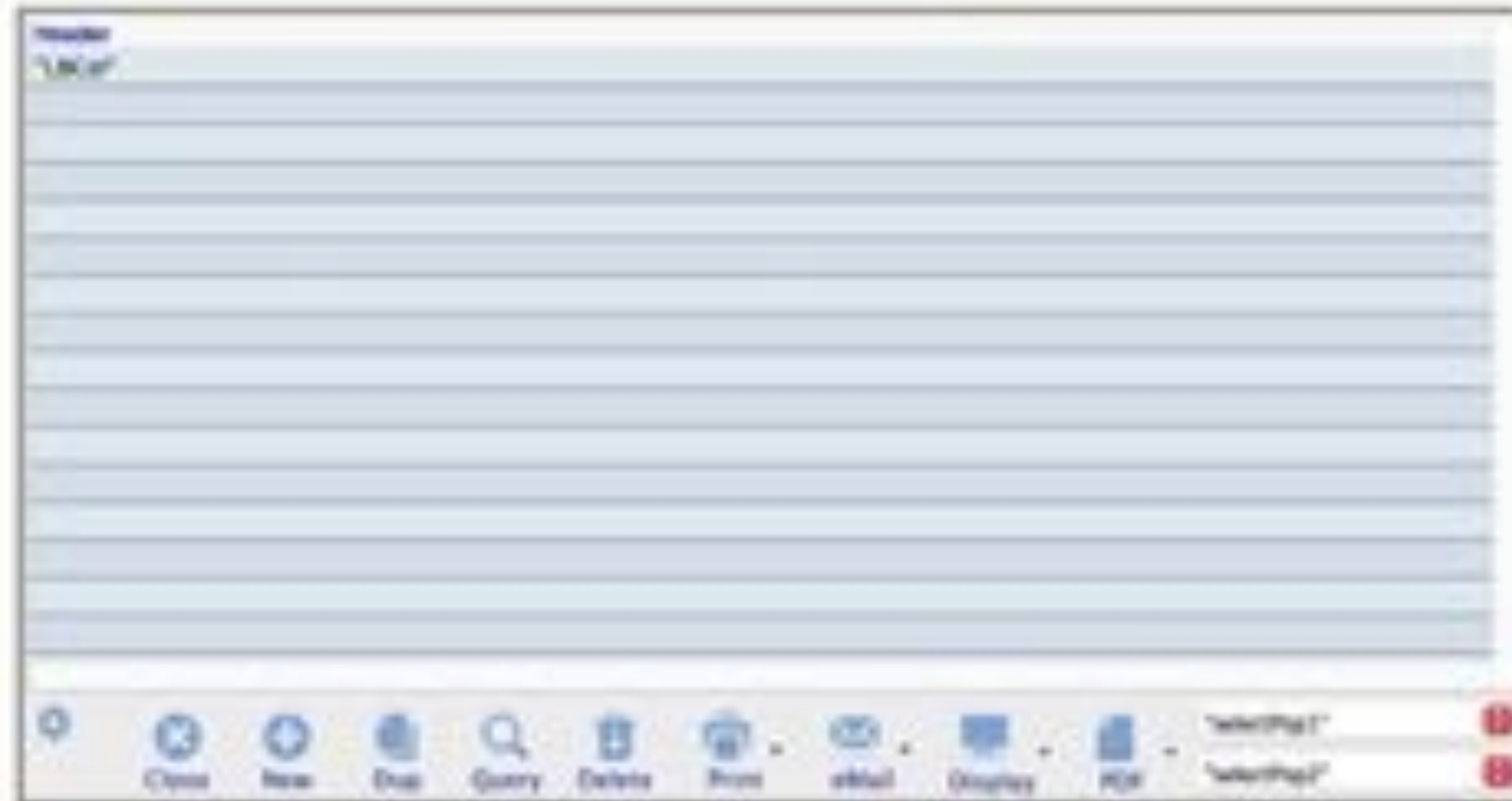
... etc.

**LB\_LIST** can automatically handle things like:

- column resizing, drag / drop
- configuration via 'Decorator'
- activating the corresponding 'editor'
- ... etc.

# BROWSERS

**LB** is intended to be used as a 'master Browser'



Because **LB** functions as a **SUBFORM**:

it also messages its Container Object so that it can handle things that **LB** does not handle itself. Same mechanism as **LB\_LIST**:

single event sent: `_msgPanelEvent`. Extra details returned in `Form.msg`

**Close**: closes window, evoking 'window position saving'. No record will be being edited, so simply needs to close.

**New**: { .event: `_msgGoto` ; .recNo: `_msgNewRecNo` }

**Duplicate**: { .event: `_msgDuplicate` } (container knows what's selected)

.... similar for others.

Like **LB\_LIST**:

Standard button panel, saved in the **UIObjects** entity for the browser

Configured using a **cs.ButtonGroup** object according to the specs

It also has a pair of 'selection' popups that get their configuration from the 'contextual menu' we define in **UI Designer**.

The actual selection of records is done using a custom-routine we supply.

- The 'gear' icon only displays when **Designer** is logged in

**WINDOW POSITION SAVING:**

Because **LB** is used in 'main' windows, I placed the code in it to save the window position upon closing:

: (Form event code=On.Close.Box) // this will only be triggered when this is a stand-alone browser. It will not be called when it is a subform

**DO\_CLOSE\_WINDOW**(Frontmost window; Bool(Form.savePos); String(Form.infoKey)) // this will update menuBar singleton & WindowManager singleton and SAVE THE POSITION of the form for this workStation

**CANCEL**

# EDITORS

Returning to EDITORS ...

Sadly, EDITORS are not yet implemented as a 4D Class. They were designed in v17, so they are principally implemented using Project Methods

- ED\_CONFIGURE
- ED\_CONFIGURE\_NAVIGATOR
- ED\_Do
- ED\_DO\_FORM
- ED\_DO\_THIS
- ED\_EMAIL\_FORM
- ED\_Load
- ED\_PANEL\_HANDLER
- ED\_PDF\_FORM
- ED\_PRINT\_FORM
- ED\_RELEASE
- ED\_Save
- ED\_SUBFORM\_DO
- ED\_UPDATE\_NAVIGATOR

**Editors** know most of how to handle the LOAD & SAVE of the entities they operate on. They also know how to 'Do' most of what needs to be done.

**Editors** know 'who to consult' for what they don't know, such as LOAD & SAVE, as they can have formulas:  
.onLoad( ); .onSave( ).

Maybe show the LB and EDITOR objects as they exist in a currently-opened window using the debugger.

**SPECIAL NOTE...** I had implemented a BROWSER -> EDITOR configuration in 'classic 4D', but it involved all sorts of arrays, using pointers, process variables, 'ARRAY FARMS', and because of all of this, I had to implement a matching BROWSER and EDITOR in SEPARATE PROCESSES. They had to exchange current selection / current record, window size & position, and while it worked, it was very involved.

**NOW ...** because of the use of Collections and Objects (a Collection of objects is really superior to a group of arrays) it is much simpler and more powerful. I could *finally* implement these structures in subforms! And include them in the same forms.

EXAMPLE: **edit\_Jobs** form

# EDITORS

## TOP-LEVEL EDITOR CONSTRUCTION

- show it using `edit_Contacts`
- 'Navigator' panel
- 'Button' panel

NAVIGATOR (`panel_Navigator`)

### OBJECT CONTAINER SCRIPT:

```
ED_PANEL_HANDLER("ED_Main")
```

```
// handle the panel! We just need  
to provide the Editor Name it  
works for
```

### 'FORM METHOD':

```
ED_CONFIGURE_NAVIGATOR //  
configure our navigator by positioning  
its form objects. It will only truly  
happen once.
```

perhaps show how  
simple the scripts are

## BUTTON PANEL



- buttons are configured (i.e. shown / hidden) with a single command in configuration
- Button Panel messages to its CONTAINER (which is the Editor)

'FORM METHOD' of the `Panel_Editor`:

```
PanelEditorFormMethod // do the panel editor (unified for Panel_Editor,  
Panel_MiniEditor & Panel_MiniEditorVert
```

### OBJECT CONTAINER SCRIPT:

```
ED_PANEL_HANDLER("ED_Main") // handle the panel! We just need to provide  
the Editor Name it works for
```

# LIVE DEMO OF BROWSERS — EDITORS

## edit\_Jobs

- 1) Show 'live' form momentarily
- 2) Show **edit\_Jobs** form

Intention: To enable the user to perform everything they need to do within the same form.

Show the subforms off-screen, and explain what I do when the form loads (i.e. position these correctly)

DEMO in **edit\_Jobs**: double-clicking a permit, having the permit editor come up in its place. when closed, back the permit LB\_LIST.

Show 'Customer' popup thing.

point out *all* of the listboxes are subforms (LB\_LIST). The buttons available are determined by settings stored in the **Browser**.

Perhaps show the 'map' zone opening a document, etc.  
Show the 'permit' document.

# NEW: DATE ENTRY WIDGETS

## 4D has some Date Widgets



However, these are not Object-compliant; they require variables as the datasource.

To me, this renders them useless to my projects

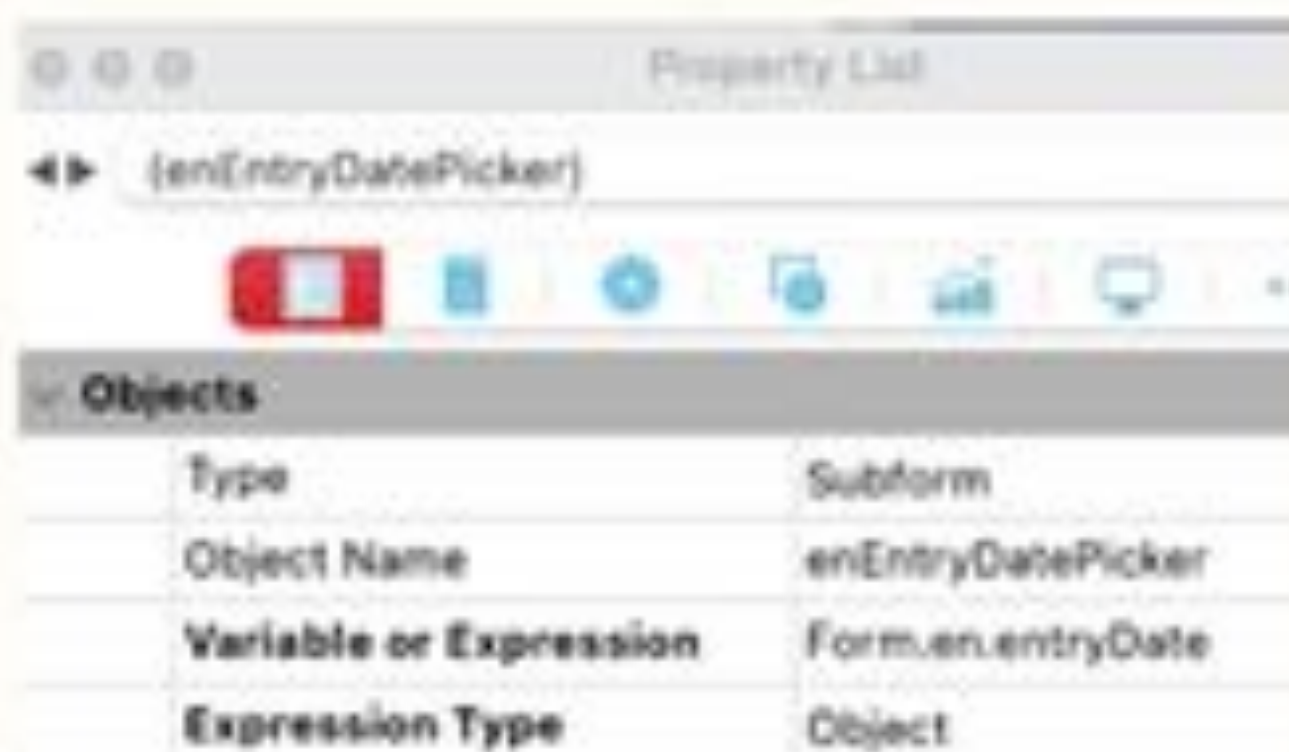
What is perhaps the single greatest obstacle for them to be made object-compliant?

The **dataSource**!

Because the datasource cannot be an object or an object attribute (i.e. **Form.xxx**)

Why? "Because the datasource would become the **Form**. object in the widget (Which is a subform), and you cannot access the context of the subform itself via pointers."

This does not work for 4D's date widgets:

The screenshot shows the 4D IDE interface. At the top, there's a 'Property List' window showing the selected object as '(enEntryDatePicker)'. Below it, there's a toolbar with various icons. At the bottom, there's an 'Objects' panel with a table showing the properties of the selected object.

Type	Subform
Object Name	enEntryDatePicker
Variable or Expression	Form.en.entryDate
Expression Type	Object

## DATE WIDGETS — for ENTRY and PICKER

### DESIGN SPECIFICATIONS I WANTED:

- The date widget should operate *directly on the attribute*; it could be something like **Form.en.entryDate** or even **Form.en.details.entryDate** where entryDate is an attribute of the object field (like: [Table]details.entryDate)
- The date widget should avoid the use of even a single process variable

The problem is that, because the widget is a **subForm**, whatever **Object** is used as its **dataSource** becomes its **Form**. object.

If the **dataSource** is from an **ENTITY**, then you cannot 'store' datums you need to operate the widget using **Form**.

### SO HOW CAN IT BE DONE?

There needed to be another way to store the 'datums' for operating the widget.

*But where????*

In my own 'date widget' development, I could get it to work swell using the **Form**. object, but that would absolutely not work if it were given a [Table]Date or [Table]ObjectField.date as a **dataSource**.

*Was I able to do it?*

(quick demo)

*What was the trick?*

# NEW: DATE ENTRY WIDGETS

## DATE WIDGETS — for ENTRY and PICKER

HOW DID I HAVE 'DATUMS' STORED IN THE WIDGET WITHOUT USING PROCESS VARIABLES OR THE FORM OBJECT (which was off-limits)?

I made another place to store them, right in the widget's form itself! How?

(Show IS\_DateEntry and explain; show Form Method as example)

Values for operating this widget are stored in the OBJECT variable of 'formData'. It is used much like the form object but without affecting the Container's form object in the parent form.

formData

Objects	
Type	Input
Object Name	formData
Variable or Expression	
Expression Type	Object

```
Sobj_FormData:=OBJECT Get pointer(Object_named: "formData")-> // form data object
$ptr_BoundVar:=OBJECT Get pointer(Object_subform_container)
```

note: Later, this could be done with a new command: **OBJECT Get value("formData")**

Objects	
Type	Input
Object Name	theMonth
Variable or Expression	
Expression Type	Numeric

I also made use of unnamed Variables; these can only be manipulated using pointers (and, eventually, **OBJECT GET VALUE()** , **OBJECT SET VALUE()**)

I also needed to provide the dataSource differently than what would have been 'optimal': like this:

Objects	
Type	Subform
Object Name	enCallDateInitial
Variable or Expression	Form.LB.Browser.en_edit
Expression Type	Object

and then specify the attribute that contains the actual date in the SCRIPT of the widget (i.e. CONTAINER OBJECT):

```
: (Form event code=On Load)
  IS_DATEENTRY_INIT("CallDate")
```

the effect of this is that the widget operates on the value:

```
Form.LB.Browser.en_edit.CallDate
which is a date field.
```

# NEW: DATE ENTRY WIDGETS

## DATE WIDGETS — CONFIGURATION

As you can see in the form `Date_Object_Widget_Test`, the configuration is very simple:

- 1) Determine the date attribute you want the widget to manage;
- 2) Specify the path leading to the attribute (but not including the attribute) in the **Widget DataSource**

EXAMPLE: The top-left example in the Demo

Type	Subform
Object Name	formDate
Variable or Expression	Form.myEntity
Expression Type	Object

Then specify the name of the attribute in the **Widget Script**.

```
// script of the included entryDate CONTAINER
Case of
: (Form event code=On_Load)
  IS_DATEENTRY_INIT("formDate"; "entryDate") // operate
  on .formDate; NEXT OBJECT name is entryDate

: (Form event code=_msgUpdate)
  BEEP // just to audibly confirm the receipt of this message

End case
```

It needs to be done this way because there is no way to access the attribute inside the widget otherwise.

But **IS\_DATEENTRY\_INIT**( AttributeName; nextEntryObject ) has the added benefit that you can specify the name of the object you want focus to go to when the user exits the widget.

## DATE WIDGETS — REFRESHING APPEARANCE WHEN DATASOURCE CHANGES

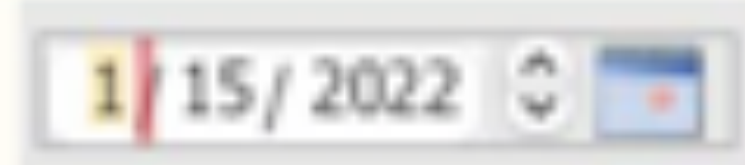
Because the Date Widget has no way of 'knowing' when an entity changes (for the purposes of refreshing its appearance), if you change its datasource (i.e. go to another entity) you must 'nudge' the widget to refresh its display.

The method **IS\_DATEENTRY\_REFRESH** must be executed in the context of the Widget, like this:

```
var $dateWidget : Text // name of the Date Widget
For each ($dateWidget; $col_DateWidgets)
  EXECUTE METHOD IN SUBFORM($dateWidget;
  "IS_DATEENTRY_REFRESH") // configure for each attribute
End for each
```

# NEW: DATE ENTRY WIDGETS

## DATE ENTRY WIDGET USER GUIDE



Month / Day / Year

### TOTALLY KEYBOARDIST-COMPLIANT

- use [Tab] and [Shift][Tab] to move between m/d/y. Can also use Left & right arrows
- **↑** — increase one. If used in the 'day' place, will observe month boundaries
- **↓** — decrease one.
- **T** — enter today's date
- **C** — open the Calendar picker (same as pressing the little calendar button)
- **[CR]** — exit the widget (focus to next object)

## DATE PICKER WIDGET USER GUIDE

January 2022						
Su	Mo	Tu	We	Th	Fr	Sa
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

### TOTALLY KEYBOARDIST-COMPLIANT

- **←** one day into future
  - **→** one day into past
  - **↑** previous week (goes into prev month)
  - **↓** next week (goes into next month)
  - **⌘→** next month
  - **⌘←** previous month
  - **⌘↓** next year
  - **⌘↑** previous year
- (IF INVOKED FROM **DateEntry** Widget)
- **[CR]** pick the date and exit (exits **DateEntry** and goes to next object)
- (IF "Stand-alone")
- **[CR]** picks the date (like a CLICK) and sends a `_msgUpdate` to the container, and updates the `dataSource` date
  - **[TAB]** treated like **[CR]** above

# Coding Suggestions

## ANALOGY: CEO OF COMPANY

Analogy: View yourself as CEO and your app as the COMPANY YOU RUN.

As CEO of our App, we:

- define organizational structure (data structure; code structure)
- Hire Workers — both ‘permanent’ and ‘temps’
- Appoint managers at different levels:
  - DataClasses (which may have several sub-divisions if you use polymorphism like I demonstrated with [UIClass] & [UIObjects]
    - EntitySelections (they are lower-level managers, under DataClasses, over Entities)
  - Custom Classes
    - ex: UI, Scheduling, etc.
- Hire Workers:
  - Entities

When work is assigned out by the CEO or by Managers, they should allocate it to the correct ‘worker’  
‘Workers’ should gain ‘knowledge’ and ‘skills’ and know their function well.

— DESIGN YOUR APP TO ALLOW THE ‘STAFF’ TO BE FULLY KNOWLEDGEABLE, TRAINED, AND COOPERATIVE

# Coding Suggestions

## EXAMPLE: 4D'S STANDARD DATA - RELATED CLASSES

Sometimes we think of a program as operating upon data, and so we may be treating our DataClasses, EntitySelections, and Entities *only* like data.

That is like a company wasting good 'employees'; they underperform and do not contribute to the success of the company. And a lot more work is required to achieve a result.

Here are some examples:

You have scheduling functions in a DataClass. Should you write *project methods* to manage an Entity's schedule, or get the *Entity* to? If a group of entities need to perform a function, could the corresponding *DataClass* or *EntitySelection* coordinate it?

Object-oriented programming is based on appropriately 'training' the objects with *knowledge and skills* to do a defined job.

**PROJECT METHODS** are like standard skill-sets or tools that any of the 'workers' can use to help them do their job.

# Coding Suggestions

## WHEN DECIDING WHERE TO PLACE YOUR CODE:

Decide: which ‘workers’ should be responsible, and most capable?

How can the ‘workers’ better-interact? Do they need some tools or knowledge to do this new task?

## EXAMPLES IN MY CODE:

I have `cs.Unit` and `cs.Worker` that have schedules.

Rather than writing Project Methods to figure out those schedules, I get the `ds.Unit` and `cs.Worker` manage their own schedules.

If their schedule for a certain period is needed, the **ENTITY** is ‘asked’ for it.

For example, the `cs.UnitEntity` can be asked:

FOR EXAMPLE: A `cs.UnitEntity` may be asked:

```
// Unit Entity
Class extends Entity

▼ local Function getState() : Text // returns the current 'state' of the record. This is used in Form: Job_EditCustomer to provide feedback on its i
  ▼ Case of
    ▼ : (This.isNew()) // if it is a new record...
      $0:="NEW"
    ▼ : (This.touched()) // changed?
      $0:="%"
    ▼ Else
      $0:="✓"
    End case

▼ local Function initConfig($seed : Object) : cs.UnitEntity // set up the initial state of a new record
  UUID_Get[This] // this will ASSIGN a new UUID
  This.isActive:=True // new ones are always assumed to be ACTIVE
  OBJ_SetAttrVals[This; $seed] // if $seed is null, then nothing will change
  $0:=This // return so it can be chained if desired

▼ local Function editValidToSave : Boolean // TRUE if the [Unit] record is valid to save
  $0:=(This.Unit#_Blank) & isAUUID[This.Type] // ensure we have the basic unit information

▶ local Function getUnitStatus($startDate : Date; $endDate : Date) : Text // check and see if the unit is scheduled for CVIP or OH. OR if it is curr

  /* ----- [ODUnit] SCHEDULING CONFLICT SUPPORT FUNCTIONS ----- */
▶ local Function getODUnitsInDateRange($startDate : Date; $endDate : Date; $ignoreJobUUID : Text) : cs.ODUnitSelection
  ▼ /* getUnitSchedJobConflicts($startDate : Date; $endDate : Date; $ignoreJobUUID : Text; $theseDatesOnly : Collection) : Text
    Called by ODUnitEntity.getScheduleStatus[ ], this checks for conflicts in the unit's deployment for the period.
    $ignoreJobUUID is the UUID that the unit is scheduled to do, and hence we can subtract that from the 'conflict'
    */
▶ local Function getUnitSchedJobInfo($startDate : Date; $endDate : Date; $ignoreJobUUID : Text; $theseDatesOnly : Collection) : Collection

▶ local Function getUnitSchedJobConflicts($startDate : Date; $endDate : Date; $ignoreJobUUID : Text; $theseDatesOnly : Collection) : Object /
```

# Standard Classes → Suggestions

Example from `cs.Unit` & `cs.UnitEntity`

Here are some standard functions I deploy in the standard 4D data-related classes:

`cs[DataClass]:`

```
local Function newEntity($seed : Object) : cs.UnitEntity // create an entity, seeding it with data from $seed
    $0:=This.new().initConfig($seed) // let the Entity configure anything it needs to for a new record
```

The `.initConfig( )` is an ENTITY Method that initializes the configuration of a new entity. It can be ‘seeded’ via an object. It sets default values, and initializes object fields as needed

`cs[Entity]:`

```
local Function initConfig($seed : Object) : cs.UnitEntity // set up the initial state of a new record
    UUID_Get(This) // this will ASSIGN a new UUID
    This.isActive:=True // new ones are always assumed to be ACTIVE
    OBJ_SetAttrVals(This; $seed) // if $seed is null, then nothing will change
    $0:=This // return so it can be chained if desired
```

# Standard Classes → Suggestions

Example from `cs.Unit` & `cs.UnitEntity`

TO AVOID SAVING 'EMPTY' OR USELESS ENTITIES:

(This is used to 'ignore' when a user clicks the 'close & save' button when nothing has been done (instead of the 'cancel' button))

`cs[DataClass]:`

**local Function `edIsValidToSave`**(`$en` : `cs.UnitEntity`) : **Object**

`// { .result = TRUE — it has all required information; FALSE = missing some; .errorMessage = the error message }`

`$0:=$en.edIsValidToSave()` `// get the Entity to tell us`

The `.edIsValidToSave( )` is an ENTITY Method that determines if the entity has sufficient data.

`cs[Entity]:`

**local Function `edIsValidToSave` : Boolean** `// TRUE if the [Unit] record is valid to save`

`$0:=((This.Unit#_Blank) & isAUUID(This.Type))` `// ensure we have the basic unit information`

# Standard Classes → Suggestions

Example from `cs.Unit` & `cs.UnitEntity`

TO ASSIST MY EDITORS:

`cs[DataClass]:`

**local Function edOnLoad** (`$editor: Text`) // do the 'load' in the EDITOR. Given Form.ED [ `$editor` ]. Operates in the context of the Form

**local Function edOnSave**(`$editor : Text`) : **Object** // do the 'save' in the EDITOR. Given Form.ED[ `$editor` ]. Operates in the context of the Form — Returns: Result of the save attempt

# Standard Classes → Suggestions

DON'T FORGET THE ENTITYSELECTION CLASSES!

EXAMPLE IN THE SAMPLE DATABASE AND COVERED EARLIER:

In EntitySelection class:

```
Function applyFormula($formula : Variant; $mode : Integer)  
    ds.applyFormula(This; $formula; $mode)
```

# NEW: 'FINDER' WIDGET

Search Contacts by Name or Company

Search Contacts by Name and Company

Search for

## Class-based Implementation


- Configured in Form Method's *On Load* handler
- variable number of columns, widths, number of rows
- 'live' feedback to Container Object each keystroke
  - however, most is managed in the widget.
- relevance levels

To use in a form:

- 1) set up the widget as a subform, with properties like:
- 2) the widget uses a listbox + opaque box to display matches. Copy / paste these from the **FinderTest** or **lookupCompany** forms.



- 3) configure the finder(s) programmatically in the Form method's *On Load* handler, like the sample forms.



Property List	
(FinderCompanyContact)	
Objects	
Type	Subform
Object Name	finderCompanyContact
Variable or Expression	Form.FC.finderCompanyContact
Expression Type	Object
CSS Class	
Subform	
Source	whenex
Detail Form	Finder (siftidgate)

## Standard Script for Container Object:

```
var $event : Object
$event:=Form.FC[OBJECT Get name].do(FORM
Event) // perform our functions!
Case of
: ($event.code=_msg_ECC_Select) // if we have a
selection made, we can process it.
GOTO OBJECT(";enContactFirst") // done

: ($event.code=_msg_ECC_Cancel) // told to cancel
GOTO OBJECT(";enContactFirst") // done

End case
```

# NEW: 'FINDER' WIDGET

## UNDER THE HOOD

In the Demo (and standard):

Form.FC — the *FinderGroup*

Form.FC.finderxxx — the *Finder* controlled by the group

Form.FC.finderxxx.columns[ ] — the *FinderColumns* belonging to a *Finder*.

The 'Finder' is a collaboration of 3 classes:

- cs.FinderColumn** — contains the configuration a column in the listbox { .colNo; .theFormula; .width; .dataType; .colName\* }  
consulted with constructing the listBox that shows matches \* (internal use)
- cs.Finder** — contains specifications; controls appearance of the Finder Widget
- cs.FinderGroup** — does the main work of operating the finders 'under' its control.

# NEW: 'FINDER' WIDGET

As you know, widgets are basically subforms, so managing them involves the **CONTAINER OBJECT** and the **WIDGET** / subform itself. The classes manage both the CONTAINER and the WIDGET code.

MESSAGES:

WIDGET-LEVEL MESSAGES:

- \_msg\_FCC\_Init — initialize appearance, settings
- \_msg\_FCC\_Edit
- \_msg\_FCC\_ResumeEdit — **resume** editing (after an up/down keystroke in the listBox, or a click in the listBox)

CONTAINER-LEVEL MESSAGES:

- \_msg\_FCC\_Select — User has selected a choice. Container should process the selection
- \_msg\_FCC\_Update — selected record was 'updated' and/or the selection was updated. This is sent to the Container to be process.
- \_msg\_FCC\_TextChange — **text** in the 'Finder' was changed. Sent to Container, but processed in cs.FinderGroup.do()
- \_msg\_FCC\_Previous — go to the previous record (listbox-caused). Sent to Container, but processed in cs.FinderGroup.do()
- \_msg\_FCC\_Next — go to the next record (listbox-caused). Sent to Container, but processed in cs.FinderGroup.do()
- \_msg\_OK
- \_msg\_Cancel