What's In a Name? ObjectPal Naming Conventions

By Vince Kellen
3/13/1994
vince@kellen.net
http://www.kellen.net/vk

Having a memorable name has helped many personalities in their careers. Actors, actresses, writers and other artists have given themselves names that roll off of our tongues with ease. After all, having a highly visible name means money.

Well, in the tangled world of programming, assigning things memorable names has similar benefits (except, perhaps, the money). This fact is so obvious and has been beaten over our heads by countless programming gurus over the past two or three decades. *Name your variables something meaningful! Don't use one-letter names for variables! Use descriptive nouns and verbs!* These and other admonitions haunt us.

In fact, developers in many different environments establish rigorous naming conventions which helps ensure that they will name variables and other programming objects something meaningful. Hungarian notation is one such naming convention, widely used in the C world.

In Hungarian notation, you are supposed to add meaningful prefixes to the variable name. These prefixes include a few letters to denote the variables base type and a few more letters to denote how the base type is used. For example, one could refer to a global string variable used to hold a window title as gstrWinTitles. The first letter in the variable name, g, denotes a global variable. The next three letters, str, denote its type - string. You then tack on the prefix, *gstr*, (pronounce g-string? OK- bad joke) to the variable's name WinTitles.

Pros and Cons

The advantages of a naming convention are clear. It provides programmers with some consistent ground rules as to how variables should be named. As long as a programmer knows the rules, they can understand the variable's use and meaning easier. Naming conventions often include some description about the variable's type, which means programmers won't have to read a variable declaration to determine the variable's type.

The disadvantages are just a bit murky. When a variable name includes its type, and the type needs to change, the variable name needs to change *everywhere it is referenced.* Variable meaning becomes linked to its internal representation, which can be a problem. In the hands of lazy programmers, variable names can also look a bit like city names in Wales. Guess which one of the following is a variable, Hungarian notation style, and which ones are cities in Wales: tywyn, tydfil, gachwin? (Send your correct answer to PI and receive a free PI baseball cap!)

Nonetheless, people are fond of naming conventions for variables and other objects. As long as the programmer has access to a good search and replace utility to change a variable's name if its type changes, then a naming convention makes good sense.

In order to get a handle on how naming conventions can actually help us, I'd like to propose the following: that naming conventions have more to do with the way our eyes process visual information than the way our minds process logical, abstract information. With a naming convention, we can glance at a variable or object and more quickly discern salient features about that object. In much the same way we can recognize someone's face quicker in broad daylight where we have more visual cues than at night where there are less clear visual cues. Programmers

are continually scanning code, with their eyes, having to quickly distinguish variable names, run time library methods, custom procedures, field names and object names. How quickly a programmer can discern these components determines his or her productivity.

The four following precepts might help us in understanding the usefulness of naming conventions:

1. *Our ability to process visual information is much better than our ability to process abstract, mathematical and to a lesser extent linguistic information.*

2. *Objects which contain less visual and hence logical or abstract information are easier to understand.*

3. *Objects which make immediately visible important details are easier to understand.*

4. *The more order and structure inherent in an object's visual appearance, the easier it is to understand. The more random or chaotic the object's visual appearance, the more difficult it is to understand.*

In fact, if we were to take visual programming to the extreme, a programmer would never write code using logical symbols. Instead, the programmer would manipulate physical, visual objects so that the job of quickly discerning important items is performed by our eyes and our visual cortex, which is quite good at quick information processing. If you have watched the new television series, called *Tek Wars*, you might have noticed that the computer hacker of this future world cracks into secure computer systems not by writing code, but by manipulating a three dimensional virtual reality space using legs, arms and body. Kind of like calf roping.

At Kallista, we have been bandying about various ways of enhancing an object's comprehensibility by adding to their names important codes. What we have settled on is by no means perfect, and by no means is it the only proper naming convention. Since a naming convention serves to make programs more readable by *your organization's programmers*, what works best might be very well different than what is espoused here.

We have decided on the following rules for naming conventions:

1. Don't deviate too far from any naming conventions which Borland has used.
2. Take advantage of the way the eye processes information.
3. Try to compress the virtual space code occupies, where ever possible.

Let's explore these rules in more detail.

Borland Conventions

Borland has adopted various conventions for naming things in ObjectPal. Variables begin with a lower case letters and the first letter of subsequent words are placed in upper case. From now on, this will be referred to as intercaps. An example of intercaps is the following variable declaration:

```
var
    dateOfBirth Date
endvar
```

Constants begin with a capital letter and intercaps from then on for each word in the constant name. From now on, this will be referred to as proper case. An example of this is the following constant declaration:

```
const
```

```
    FileClear = 1
endConst
```

Run-time library method and procedure names are written using intercaps. UIObject noise names use proper case. Field object names take on the same case as the corresponding field names. Table frame (TF) objects and multi-record objects (MRO) names are the name of the attached table and always appear as all upper case, by default. Our naming convention tries to accommodate these and any other default Borland naming conventions.

How does the eye process information?

Because Hungarian notation adds information to the front of a variable, one has to wade through the prefixes before getting to the name of the variable. This seems a bit backward. Since programmers are in the habit of reading from left to right, and since an object's name usually has more value than it's type, it would be wiser to place the type portion of the object name at the end of the variable name, not in front of it. We feel that this is important, because in typical database programs, the object's name, which indicates its use in the application, is more important than its type. In addition, if one where to produce a sorted list of object names, would a variable report sorted by type be more important than one sorted by name? Most likely not.

The variable's name has a better chance of communicating what role the variable (or object) plays in a particular algorithm than its type. For this reason, we feel that the type portion of the naming convention should appear as a suffix to the object's name, not a prefix.

Besides, prefixes in language serve to place two word parts with separate meanings next to each other in order to create a new meaning. The word disinterested is comprised of a prefix dis- and a word interested. The prefix negates the root word. Assigning prefixes the same role in a programming naming convention under the assumption that these prefixes can be as easily understood as those in written language is wrong. Prefixes in language most frequently appear in front of verbs. Variable names are nouns. And prefixes in programming naming conventions help us distinguish between different types of nouns. This information can be moved to a suffix position and improve variable or object recognition and recall.

Compressing Cognitive Code Space

Code takes up abstract space. The more code an application has, the harder it is to find things in the code, the harder it is to navigate through the code and the harder it is to comprehend. If the environment supports tools which help find things and help navigate, then those tools compress the cognitive code space. The application "appears" smaller because we can move around in it quicker.

In ObjectPAL, variables declared in a Var window are not immediately accessible, whereas variables declared after a method declaration are. Hence it is useful to include type information in a variable's name. If the Var window is not immediately accessible, then having the type information as part of the variables names lets us rope those calves faster.

In ObjectPAL, since a great many variables are declared is some far-off Var window, using a naming convention helps shrink the cognitive code space. In addition, if we can designate how the variable is being used -- is it a global variable or a regional variable? -- then we can shrink the cognitive code space even further.

In addition, ObjectPAL has more than one thousand run-time library methods and procedures. It is impossible to remember which methods and procedures are custom and which are not. For this reason, a naming convention can label custom methods and procedures with some identifier. Some examples include:

cmAddProjects()
addProjectsCm()

In the first case, the prefix cm is added and in the second, it is added as a suffix. Adding such a designation to custom methods and procedures serves two purposes. First, it clearly marks custom methods and procedures. Second, it ensures that future versions of Paradox won't have method or procedure names which conflict with your own method or procedure names.

Hungarian notation allows for prefixes to denote global use of a variable. In ObjectPal, a variable has two levels of visibility. If the variable is declared either above or below the method or procedure declaration, then the variable is only visible to the method declared. If the variable is declared in a var window of a given UIObject, then the variable is visible to that UIObject and any other objects below it in the object tree.

Frequently, however, programmers declare variables in a form's Var window. This means that all variables declared in this manner are visible to all objects on the form. Hence they are often called global variables. Variables declared at some intermediate object in the containership hierarchy are often called regional variables. These variables are visible to a portion (a subtree) of the object tree.

In addition, variables can be declared in a library, and the variable's value can be set and retrieved by specialized library methods. These variables can then be shared across forms. These variables, although global to the library, are in practice global to the application.

We then have three types of 'global' variables:

1. variables global to all forms, declared in a library or a form
2. variables global to a form
3. variables global to a portion of the object tree, 'regional' variables

How should these be denoted? One could tack on another suffix to a variable's name, such as *gf* which stands for global at the form level:

```
; form:var() built-in
var
    beginningCustNumberSIgf  SmallInt
endvar
```

Or the use of the variable can be denoted by adding a prefix:

```
; form:var() built-in
var
    gfBeginningCustNumberSI  SmallInt
endvar
```

But since a prefix can diminish the ability of programmers to read left from right and determine a variable's name, perhaps the suffix designation might be better. In any case, the following variable designations might work:

gl	Global in a library - global to all forms
gf	Global to the form
gr	Global at some level other than the form

Other Bits of Trivia

There are other things a naming convention ought to cover. For example, custom methods and procedures can return a value. Perhaps the method or procedure name should be appended with the return type as in:

```
cmGetHighestLI()
cmDisplayErrorX()
```

The first custom method returns a long integer (LI). The second custom method returns nothing, which is designated by the X suffix. Or, if you want, perhaps you can leave off the X to designate that a custom method returns nothing.

Another place where a naming convention can help is to clarify the contents of a dynarray. Since a dynarray takes a string as an index to an array element, that string can be written exactly like a variable name. The following example may help illustrate the point.

```
var
    securityDY DynArray[] AnyType
endvar
securityDY["nameS"] = "Vince Kellen"
securtyDY["levelSI"] = 3
securityDY["activeL"] = True
```

In this example, the dynarray, security, is designated by the DY suffix. Each index in the dynarray is written as if it were a variable name, with the type denoted by the suffix in the dynarray index. S means string; SI means small integer and L means logical .

And lastly, a naming convention can be used to restrict how fields in tables should be named. Why would one want to restrict how field names in tables are represented? Because if the application ever needs to be moved to a client-server environment, careful use of a naming convention can ensure than none of the field names ever need to be renamed because some server does not allow certain characters.

If this is something desirable to you, you might want to consider leaving out any spaces in field names in tables, any special characters, such as $ or % or the period character. In addition, you might want to consider using underscores to separate words in field names rather than case. Some back-end servers might represent the field in all upper case, which can sometimes make the field name look confusing, as in the field name BillLabel, which when presented in upper case looks like BILLLABEL. The three L's all in a row will most likely lead to a spelling problem.

A Proposed Naming Convention

Now that the issues have been explored, let's take a stab at summarizing a proposed naming convention. First, the rules for tacking on type suffixes to variables. Each variable type will be represented with a suffix at the end, denoting the variable's type. Table 1 shows each variable's suffix and an example.

Table 1. Variable suffixes.

AnyType	A	tokenA
Application	APP	selectedAPP
Array	AY	securityAY
Binary	BIN	selectedFileBIN
Currency	$	totalPay$
Database	DB	selectedDB
Date	D	startD

DateTime	DT	stopDT
DDE	DDE	qproDDE
DynArray	DY	securityDY
FileSystem	FS	currentDirFS
Form	F	customerF
Graphic	G	empPictureG
Library	LIB	appLIB
Logical	L	continueL
LongInt	LI	totalCountLI
Memo	MM	commentsMM
Menu	M	dataM
Number	N	totalWeightN
OLE	OLE	wpOLE
Point	PT	currentLocationPT
PopUpMenu	P	editP
Query	Q	adhocQ
Record	REC	userREC
Report	R	ytdSummaryR
Session	SS	auditSS
SmallInt	SI	counterSI
SQL	SQL	openOrdersSQL
String	S	lastNameS
Table	TB	selectedTB
TableView	TV	selectedTV
TCursor	TC	customerTC
TextStream	TS	commentFileTS
Time	T	endT
UIObject	UI	selectedUI
User-defined type	UDT	QuoteHeaderUDT

Absent from this list are all the eventInfo variable types:

- ActionEvent
- ErrorEvent
- Event
- KeyEvent
- MenuEvent
- MouseEvent
- MoveEvent
- StatusEvent
- TimerEvent
- ValueEvent

These variables are always declared in the built-in method's header (with the name eventInfo) and very rarely does the programmer have to declare a variable for these object types. For this reason, they have been removed from the list. However, if you want to declare these variables outside the built-in methods we propose the following suffixes:

ActionEvent	AE
ErrorEvent	EE
Event	E
KeyEvent	KE
MenuEvent	ME
MouseEvent	MSE
MoveEvent	MVE

StatusEvent	SE
TimerEvent	TE
ValueEvent	VE

Next, the following rules for naming variables, constants, types and other objects are proposed:

1. Variables will be represented in intercaps.
```
ageOfRetirementSI
```

2. Constants will be represented in proper case.
```
AgeOfRetirementSI
```

3. Object types will be in proper case.
```
var
    counterS SmallInt
endvar
```

4. Custom types will be represented in proper case.
```
QuoteHeaderUDF
```

5. Fields in tables will be represented in proper case with underscores between words.
```
Age_Of_Retirement
```

6. UIObject names will be in proper case, except for TF and MRO objects, which will be in upper case.
```
CustomerNumber ;(field object)
CUSTOMER ;(MRO)
```

7. Form, report, library, script, table, aliases and other file name string literals will be in all lower case.
```
custF.open(":custdata:customer")
```

8. RTL methods and procedures will be represented in intercaps.
```
isFirstTime()
```

9. Indexes will be represented in proper case with an NDX suffix. If the index contains multiple fields, each field should be represented in the index name, if practical.
```
CustomerDateNDX
```

10. Referential integrity rules will have an RI suffix.
```
AccountCodeRI
```

11. Global variables will be represented with the following suffixes: gl, gf, gr where gl is global in a library, gf is global at the form level and gr is global at some level lower than the form.
```
lastOrderPrintedSgl
isFormApprovedLgf
currentListSgr
```

12. Don't include form, report, script and library extensions in quoted strings. When an application has is delivered and form, report, library and scripts extensions change from .?SL to .?DL, none of the code will need to be changed. Also, don't include table extensions (.DB or .DBF) unless you absolutely need to. It's not wise to have two tables with the same eight character name but a different table type, as in "customer.dbf" and "customer.db."

13. Custom methods and procedures include a cm suffix after it's return type. If the custom method or procedure has no return type, the suffix X will be used.

```
createNewProjectLcm()
createNewProjectXcm()
```

14. Dynarray indexes will follow the same naming conventions as variables.
```
houseDY["ageSI"] = 28
houseDY["marketValue$"] = 120500
```

15. If you plan on using *shadow event* custom methods in a library, then give the custom method the same name as the built-in method, except add the cm suffix. Shadow events are custom methods that are placed in a library and are typically called from the form level built-in methods.
```
menuActionLcm()
```

A Starting Point

Until we can move to an entirely visual and kinesthetic programming environment (much like the cowboy hackers in *Tek Wars*), naming conventions can help us in the here and now. They can help give us meaningful visual cues which lets us distinguish important features in our code.

The naming conventions described here are intended as a starting point. As with most things in programming, standards are often local. Almost every shop tailors standards for their environment. Naming conventions can help, not hinder, your development efforts. Use them wisely. And remember, punch them doggies.