Handy Handlers #4: IfX

"If I had eight hours to chop down a tree,
I would spend 6 hours sharpening the axe."
- Benjamin Franklin

In previous weeks we covered handlers that reduce a dozen or more lines to a single one-line call. This week we'll make one that's just one line itself. What's the point of using one line to call one line? Simple convenience.

How many times have you typed this:

```
if it is empty then exit repeat
```

This is often the first thing you'll write following any of the ask or answer commands, as those commands return their values in the local variable "it", which is empty if the user select's the dialog's "Cancel" button.

Sure, it won't kill you to type that line a few hundred times in your code. But your time is valuable, and if you code professionally it's your client's money you're spending. Programming always involves unknowns, since the only program worth writing is the one that hasn't already been done. Save your budget for when you need it, and give yourself a little edge here and there where you can.

To trim a little time from your development, this week's handler is one of the simplest in my library:

```
on IfX pIt
  if pIt is empty or pIt is "Cancel" then exit to top
end IfX
```

The handler name is just a simple abbreviation, and short enough that calling it reduces our typing appreciably:

```
answer "Save changes?" with "Cancel" or "Save"
IfX it

answer file "Select an image:"
IfX it
```

We can use it with custom modal dialogs as well if we set the value of the dialogData global variable to the short name of the button used to close the dialog:

```
modal "MyDialog"
IfX the dialogData
```

Such small conveniences can add up over a 10,000-line project. And reducing typing means reducing typos, so the benefits of using even simple one-liners as easy-to-type handlers add up when used often.

While we're here, note that I've been naming all of our custom handlers with the first letter capitalized, while all built-in commands and functions start with a lower-case letter. This helps to distinguish between built-in language features and your custom handlers. With a language offering as many commands, messages, and functions as Transcript, you can't be sure that the next person maintaining your code will have memorized all of them.

Nearly every project will have more than one set of eyes responsible for maintaining it. Even if it's just a personal project on which you're the only programmer, if you set it aside for several months and then come back to the code, you'll be seeing it through different eyes, no longer in the same head space you were in when you first wrote it. How many times have you looked at something you wrote last year and asked yourself, "What was I thinking?"

Good naming conventions can make your code easier to read. And perhaps "read" isn't the best word, since code is far more frequently skimmed than read. Anything that helps make special words stand out visually from the rest of the code will help you skim to what you're looking for faster.

In addition to visual distinction, naming conventions can provide a second benefit by describing the origin of a variable. For example, it's common practice in many programming languages to add a lower-case "g" before the name of a global variable, e.g., "gMyPrefsFilePath".

The practice of using naming conventions to carry information about a variable's type was popularized by Microsoft's Charles Simonyi, and in honor of his nation of origin it's referred to as "Hungarian Notation". True Hungarian notation as practiced by disciplined C++ programmers can be complex, noting not only origin but data type as well. But since high-level languages like Transcript do not require you to specify data types for variables, you can use a simplified form I call "Hugarian-lite".

Here's a table of notation commonly used among programmers working in Transcript and a few other high-level languages:

Char	Meaning	Example
g	Global variable	gMyGlobal
t	Local ("temporary") variable	tMyVar
s	Script-local var	sMyVar
p	Parameter	pMyParam
k	Constant	kMyNumber
u	User-defined (or custom) properties	uMyProp

This table is borrowed from the Script Language Guide.

You may be wondering why "u" us preferred to denote custom properties over "c", which might seem a logial choice. Indeed it is, and the clue can be discovered here on this page from the MetaCard site where the Revolution engine was born:

The MetaTalk language will also be extended to provide a more full featured object-oriented programming environment, which will allow development of larger-scale applications with MetaCard.

Anticipating object-oriented extensions to Transcript somewhere down the road, the use of "c" in Hungarian-lite is reserved for one day being able to denote object classes, as it is commonly used in other programming languages.

A we've been discovering with handler design, planning for future enhancement is useful when adopting naming conventions as well.