EYRIE SOLUTIONS™

Advanced Windows Integration with Eagle, Garuda, & Harpy

Joe Mistachkin @ Tcl 2016

https://eyrie.solutions/

Overview

EYRIE SOLUTIONS™

What is Eagle?

- Eagle (Extensible Adaptable Generalized Logic Engine) is an implementation of the Tcl scripting language for the Common Language Runtime (CLR).
- It is written completely in C#. Superficially, it is similar to Jacl; however, it was written from scratch based on the design and implementation of Tcl 8.4.
- It provides most of the functionality of the Tcl 8.4 interpreter while borrowing selected features from both Tcl 8.5 and 8.6.
- There are some extra features that are not present in native Tcl, mostly for dealing with Windows and the .NET Framework.

EYRIE SOLUTIONS™

What can Eagle do for me?

- Help you to seamlessly integrate with applications, libraries, and system components on Windows.

- Help you to securely deploy applications and packages written in Eagle or native Tcl to your users.

Integration

Like Tcl before it, Eagle enables integration.

- COM components via `[object]` command
- .NET Framework via `[object]` command
- Databases via `[sql]` command
- Native Tcl/Tk via `[tcl]` command
- Native DLLs via `[library]` command
- Web via `[uri]` command
- Other protocols via `[socket]` command
- Command line tools via `[exec]` command

EYRIE SOLUTIONS™

Case Study: Win32

EYRIE SOLUTIONS™

How can I prevent the native console window from closing?

- Using the Win32 API via Eagle.
- With the **`[library]`** command, you can access native APIs, including those provided by the underlying operating system.
- The necessary code can be found in the script file:
  - `examples\Win32\ex1.eagle`

EYRIE SOLUTIONS™

Case Study: COM

Can I use a COM class?

- If you have the Primary Interop Assembly (e.g. for Microsoft Office), you can **`[object load -import]`** it and use COM “early-bound” (see https://urn.to/r/pia).

- Otherwise, you’ll be using COM “late-bound”; example code can be found in the “object-15.10” test for the Eagle core library.

Case Study: calc.exe

EYRIE SOLUTIONS™

How can I automate GUI applications?

- Per a post to the “tcl-core” mailing list, somebody wanted to automate the venerable “calc.exe” applet that ships with Windows 7.
- They referred to an existing, non-working, example written in PowerShell.
- The working example, written in Eagle, can be seen in the script file:
  - `examples\calc.exe\ex2.eagle`

EYRIE SOLUTIONS™

Why does this matter?

- Subjectively, the Eagle script seems easier to understand and it is less verbose.
- The user wanted to use native Tcl.
  - This can be accomplished by using Garuda to evaluate the Eagle script.
- Yes, it is a “toy” example; however, it shows that Windows applications supporting advanced accessibility can be automated using native Tcl.

Non-Trivial Accessibility Example

- Assistive Context-Aware Toolkit (ACAT) is an open source platform developed at Intel Labs to enable people with motor neuron diseases and other disabilities to have full access to the capabilities and applications of their computers through very constrained interfaces suitable for their condition (see https://urn.to/r/acat).

Non-Trivial Accessibility Example (continued)

- Since ACAT is written entirely in C#, it is easily usable from Eagle.
- Using something like ACAT from native Tcl (i.e. without Eagle) would not be possible.

Case Study: Excel

EYRIE SOLUTIONS™

How can I automate Excel?

- Really, this is just an example of using COM early-bound, via a Primary Interop Assembly.
- It applies to any part of Microsoft Office, not just Excel.
- Example code can be found in the “excel-2.1” test for the Eagle core library.

Case Study: Mathematica

How can I automate Mathematica?

- Wolfram Research provides a managed assembly called “NETLink”.
- As with any other managed assembly, it is easy to access using Eagle.
- Example code can be seen in the script file:
  - `examples\Mathematica\ex3.eagle`

Case Study: WinForms

How can I create a GUI application?

- The simplest way to create a GUI application when running on the .NET Framework is via Windows Forms (WinForms).
- Example code can be found in the “winForms-8.1” test for the Eagle core library.

Case Study: Tcl/Tk

How can I use native Tcl/Tk?

- Eagle is a stand-alone implementation of Tcl.
  - Why would you want to access native Tcl from it?
  - Why not?
- The **`[tcl]`** command provides the ability to dynamically load, use, and unload any supported native Tcl library.
- Example code can be seen in the script file:
  - `examples\Tcl\ex4.eagle`

EYRIE SOLUTIONS™

Case Study: SQLite

EYRIE SOLUTIONS™

How can I use SQLite?

- When using the .NET Framework, database access is typically accomplished via ADO.NET.
  - PostgreSQL, MySQL, Oracle, SQL Server, and SQLite are all supported, via their associated ADO.NET providers.
- Example code can be seen in the script file:
  - `examples\SQLite\ex5.eagle`

EYRIE SOLUTIONS™

Case Study: Fossil

EYRIE SOLUTIONS™

How can I automate command line applications?

- The **`[exec]`** and **`[kill]`** commands provide the ability to manage external processes.
- The Unix-centric redirection syntax from native Tcl is not supported, nor are the command pipelines; however, many options have been carefully added in order to facilitate important integration scenarios.
- Example code can be seen in the script file:
  - `examples\Fossil\ex6.eagle`

EYRIE SOLUTIONS™

Case Study: #record

EYRIE SOLUTIONS™

How can I save interactive commands?

- The REPL exposes an interactive input callback that receives all interactive input and may allow it verbatim, alter it, or cancel it.
- This feature can be used to save all interactive commands to a file (e.g. upon exiting the shell, etc).
- Example code can be seen in the script file:
  - `examples\#record\ex7.eagle`

EYRIE SOLUTIONS™

A bit about interactive commands…

- All interactive commands start with a “#” character (try “#help” to see them).
- They may be built-in or “extensions”.
- Interactive extension commands may override built-in interactive commands.
- An extra “#” character may be used to force an built-in interactive command to be used.
- An extra two “#” characters may be used to bypass the interactive input callback.

EYRIE SOLUTIONS™

Case Study: #check

How can I check for updates?

- There is “#check” built-in interactive command that checks the latest release version against the one currently running.
- It is designed with security in mind.
- All communication is done using SSL.
- All downloaded binaries are signed and hashed, with multiple algorithms.
- If an update is needed, the dedicated updater tool “Hippogriff” is launched.

EYRIE SOLUTIONS™

Case Study: Remote Debugger

EYRIE SOLUTIONS™

How can I debug a script running somewhere else?

- Using the `[socket]` command and the ScriptThread class, it is relatively easy to inspect an Eagle interpreter in a different process or on a different machine.
- The necessary code can be found in the directory:
  - `examples\RemoteDebugger`

Other Integration

But wait, there's more!

- MSBuild
  - There are custom "build tasks" that expose Eagle to projects that use MSBuild.
- PowerShell
  - There are custom "cmdlets" that expose Eagle to PowerShell scripts.
- WiX
  - There is a custom "extension" that exposes Eagle to installer projects that use WiX.

EYRIE SOLUTIONS™

Ok, but I use native Tcl.

- Using native Tcl is fine, of course; however, that does not mean you have to use it *exclusively*.

- Native Tcl is cross-platform; however, when your primary (or only) platform is Windows, taking full advantage of its features may be advantageous.

EYRIE SOLUTIONS™

Plugins

EYRIE SOLUTIONS™

What is an Eagle plugin?

- A managed (CLR) assembly (binary) containing executable code that defines one or more classes implementing the IPlugin interface and providing a suitable constructor.
- Typically, a plugin will add one or more commands to the interpreter; however, this is not required.

EYRIE SOLUTIONS™

What is Harpy?

- Harpy is an Eagle plugin.
- Originally, it provided public key infrastructure-based software license enforcement.
- Enhanced to provide signed script verification and evaluation.
- Supports management of license certificates, script certificates, and trusted key rings.

What is Garuda?

- Garuda (a.k.a. the “Eagle Package for Tcl”) is a stubs-enabled package for Tcl that provides any application using Tcl full access to components written for the Common Language Runtime (CLR).

EYRIE SOLUTIONS™

Native Tcl?

What about native Tcl scripts?

- Originally, Eagle was designed to be a standalone library that simply provided a purely managed (via C#) implementation of Tcl, without using P/Invoke or "unsafe" code.
- Eventually, Eagle was extended to provide optional compile-time features that were allowed to use P/Invoke.
- This permitted the creation of the native Tcl integration subsystem (e.g. the TclWrapper class, etc).
- Much later, the Garuda native package for Tcl was created. It leveraged the existing native Tcl integration subsystem of Eagle, **after** bootstrapping the CLR in the native Tcl process, to enable seamless integration of native Tcl with Eagle.

Using Eagle from native Tcl?

- Yes, using Garuda, it is possible to make full use of Eagle, including its plugins, from native Tcl.
- By default, an `[eagle]` command gets added to the native Tcl interpreter, which simply calls into the Eagle `[eval]` command.
- The integration between native Tcl and Eagle can be customized; however, the defaults are good enough for most uses.

EYRIE SOLUTIONS™

I still don't understand.

- To summarize:
  - Using Eagle and Garuda together allows any supported native Tcl (i.e. version 8.4 or higher, including TclKits and KitDlls) to have complete access to all functionality provided by the CLR, the .NET Framework, and anything built on top of them.

Garuda

Installing Garuda (and Eagle)

- The “easy” way is to use the Teacup tool included with ActiveTcl; however, the Garuda binary in the ActiveState package repository has not been updated in quite some time, e.g.:

```
CD /D "C:\path\to\ActiveTcl\bin"
teacup install Garuda
```

- The “hard” way involves downloading the appropriate Garuda and Eagle binary packages and extracting them using zip into a directory that will be added to the auto-path for the desired installation of Tcl, e.g.:
  - https://urn.to/r/eagle_pkg
  - https://urn.to/r/garuda_pkg

Garuda Example #1

```
#
# NOTE: This is the only required command.
#
package require Garuda;     # a few seconds…
#
# NOTE: The following commands are optional…
#
garuda dumpstate;           # observe…
eagle parray tcl_platform;  # observe…
garuda shutdown;            # cleanup…
```

What just happened?

- The Garuda native Tcl package was loaded into the interpreter.
- Per its default behavior, this process involved loading the CLR into the process, starting it, loading the Eagle managed assembly into the default application domain (more on this later), and calling into Eagle to setup the bidirectional bridge between native Tcl and Eagle.
- All of the above happened in response to the **`[package require Garuda]`** command; all the other example commands simply demonstrate using the package.

EYRIE SOLUTIONS™

That sounds complicated.

- Internally, it is a bit complicated; however, the only script-visible “side-effects” that really matter are the two new commands that were added to the native Tcl interpreter, e.g.:
  - **`[garuda]`**
    - This command is used to startup and shutdown the CLR as well as introspect various state information associated with the Garuda package (more on this later).
  - **`[eagle]`**
    - This command is used to evaluate an Eagle script in the Eagle interpreter associated with the current native Tcl interpreter (more on this later).

Garuda and “NativePackage”

- The “NativePackage” class in the Eagle core library implements the non-native (i.e. managed) entry points used by Garuda.
- Garuda supports connecting to Eagle via the default AppDomain.
- It supports “safe” native Tcl interpreters.
- It supports Eagle interpreter isolation.

Eagle Enterprise Edition

EYRIE SOLUTIONS™

Commercial Licensing

- The Harpy and Badge plugins are both commercial products.
- The files on the provided USB thumb drive are licensed for your private use only; they are not for redistribution.
- A commercial license may be obtained, at a specially discounted “conference tutorial” rate, directly from me…
- Also see: https://urn.to/r/eee_license

Harpy & Badge

EYRIE SOLUTIONS™

Installing Harpy

- Copy the Harpy distribution files to a (new) subdirectory "**`lib\Harpy1.0`**" within the Eagle installation directory (i.e. the directory that contains the "**`bin`**" and "**`lib`**" subdirectories).
- Set the environment variable "**`Master_Certificate`**" to the fully qualified path to the license certificate file, typically via the Control Panel applet.

EYRIE SOLUTIONS™

Installing Badge

- Copy the Badge distribution files to a (new) subdirectory “**`lib\Badge1.0`**” within the Eagle installation directory (i.e. the directory that contains the “**`bin`**” and “**`lib`**” subdirectories).

They are installed, now what?

- Having the Harpy (and Badge) plugins installed allows you to load them via "**`[package require]`**".
- Typically, there are at least three phases when the signed script evaluation feature is going to be used:
  - Loading the Harpy plugin.
  - Configuring the Harpy plugin for signed-only script evaluation.
  - Loading the Badge plugin.
  - The Badge plugin provides the script certificates for all core script library and test files.

Loading Harpy & Badge

```
# STEP 1: Load the Harpy plugin.
package require Security.Core
# STEP 2: Enable Harpy policies.
security true
# STEP 3: Load trusted keyring(s).
keyring bootstrap
# STEP 4: Load the Badge plugin.
package require Security.Certificates
```

Harpy Demo

That was quite complex.

- It’s a bit complex because the Harpy and Badge plugins are modular and designed to support multiple scenarios.
- However, most of the time, the command **`[source enableSecurity]`** should be used instead (via the “**`-security`**” command line option).
- There is a corresponding **`[source disableSecurity]`** command as well.

They are loaded, now what?

- As long as the signed-only script policy is enabled, all attempts to use `[source]` will result in Harpy verifying the script certificate associated with the target script file.
- If the script file being evaluated is not local (i.e. `[source]` was used on a remote URI), Harpy will attempt to download the script certificate and then verify it.

How does this apply to native Tcl?

- Since the native Tcl **`[source]`** command is not handled by Harpy, how can it be used to secure native Tcl scripts?

Harpy from native Tcl…

- How do we take advantage of the underlying Harpy signed-only policy functionality when evaluating a native Tcl script?

```
[interp readorgetscriptfile]
```

Other alternatives…

- Of course, you could replace the Tcl **`[source]`** command with something that takes advantage of Harpy.

- However, that is far more intrusive than simply using the **`[interp readorgetscriptfile]`** Eagle sub-command followed by the native Tcl **`[eval]`** command.

Does it work on Mono?

Yes.

Quiz #1

How dangerous is the following command?

```
source http://example.com/file.tcl
```

Why?

What does it do in native Tcl?

What does it do in Eagle?

Quiz #2

- Can we make it safer?
- How?

Wrapped Script Demo

EYRIE SOLUTIONS™

So, what did we just see? (#1)

- Load an Authenticode signed native Tcl library using default search semantics:

```
tcl load -findflags \
    +TrustedOnly -loadflags \
    +SetDllDirectory
```

So, what did we just see? (#2)

- Setting up the native Tcl interpreter with variables, a procedure, etc:

```
tcl eval [tcl master] {
  set argv {}
  # ... etc ...
}
```

So, what did we just see? (#3)

- Load the Harpy plugin:

```
package require Security.Core
```

So, what did we just see? (#4)

- Enable the Harpy signed-only policy:

```
security true
```

So, what did we just see? (#5)

- Load the trusted key rings:

```
keyring bootstrap
```

So, what did we just see? (#6)

- Read the source code for Tk Tetris:

```
set scriptFile \
   [file join $path tetris.tcl]

set script \
   [interp readorgetscriptfile \
       -- "" $scriptFile]
```

So, what did we just see? (#7)

- Copy the Tk Tetris source code into the native Tcl interpreter:

```
tcl set [tcl master] script \
   $script
```

So, what did we just see? (#8)

- Have native Tcl service events...

```
tcl eval [tcl master] {
    eval $script
    after 0 list
    vwait forever
    unset -nocomplain forever
}
```

So, what did we just see? (#9)

- Unload native Tcl (optional):

```
tcl unload
```

EYRIE SOLUTIONS™

So, what did we just see? (summary)

- The important steps were from #3 to #6, simplified and shown here:

```
package require Security.Core
security true
keyring bootstrap
interp readorgetscriptfile \
   -- "" tetris.tcl
```

Ok, but how does that improve security?

- For fun, we'll run the demo again, but this time we'll slightly alter the "tetris.tcl" file first.

Threats

1. Web server is (or becomes) compromised.

2. Man-in-the-middle of HTTP response.

3. Malicious script.

Defences

- The client does not really care about the web server, per se; it only cares about the script(s) that it downloads. Therefore, it can use the Harpy signed-only script policy to defend against this threat.
- We can (and should) use HTTPS. This is not a Harpy requirement, it's just a good security practice.
- We can use a "safe" interpreter. Depending on who signed the script and how "trusted" they are, this may be overkill.

What have we learned?

- When using the Harpy signed-only policy, any script that is unsigned or has been altered in any way since being signed will cause the Eagle script engine to reject it.

Security & Stability

Denial of Service (DoS)

- Consuming CPU cycles.
  - e.g. `while 1 {}`
- Causing a hard stack overflow.
  - e.g. `proc r {} {r}; r`
- Causing a hard out-of-memory error.
  - e.g. `set x 1; while 1 {set x $x$x}`
- Corrupting the interpreter state.
- Crashing the process or operating system.

Information Disclosure

- Detailed version information.
  - For the operating system.
  - For the Eagle core library.
  - Can be used to target vulnerabilities.
- Operating system and environment information.
- User information.
  - Any information accessible via the currently logged in account.

Elevation of Privilege (EoP)

- Escaping the “safe” interpreter.
- Escaping the AppDomain.
- Escaping the process and/or session.
- Escaping the machine.

Generalizations for Security

- You cannot have security without stability.
- You can only ever be as secure as the underlying platform.
  - Think "full stack" here, including the hardware, operating system, and runtime / virtual machine.
- You are only as secure as your least secure component or layer.
- You are rarely as secure as you think you are.
- If you have not tested your security, you are not secure.

EYRIE SOLUTIONS™

Surface Area

- What is the surface area of the system?
  - Can it be reduced and still retain all the necessary functionality?
- How are users expected to access it?
  - e.g. Thick client/server, web site, etc.
- Can users access it any other way?
  - e.g. Talk directly to the server (i.e. bypass client), connect to local database, etc.

Microsoft.NET versus Mono

- Any version of the Microsoft.NET implementation of the CLR running on Windows is more stable and secure than any version of Mono running on any operating system.

- Recent versions of Mono (e.g. 4.x) are getting better, partially due to including more code from Microsoft verbatim; however, they still have a long way to go.

What security does Eagle provide?

- Core script engine.
  - Supports on-demand script cancellation and timeouts.
  - Handles soft stack overflow errors gracefully, avoiding hard stack overflow errors.
  - Handles out-of-memory gracefully, mostly thanks to the CLR itself.
  - Prevents unhandled exceptions from being thrown by a script being evaluated.
- **`interp create`**
  - Use the "**`-safe`**" option to limit surface area.
  - Use the "**`-isolated`**" option to create an entirely new AppDomain (more on this later).
- **`load`**
  - Capable of loading each plugin into a new AppDomain.
  - Capable of verifying Authenticode and strong name signatures.

EYRIE SOLUTIONS™

What security does Harpy provide?

- Signed-only script policies.
  - Prevents any unsigned (or untrusted) scripts from being evaluated using `[source]` and its associated script commands and/or managed API methods.
- License verification.
  - Prevents any protected plugin from being loaded unless an appropriate license can be located and verified.

Compatibility

EYRIE SOLUTIONS™

Compatibility with Tcl

- More-or-less 100% compatible with Tcl 8.x, where “x” is currently 4. Missing **`[binary]`**, **`[scan]`**, and **`[trace]`**.
- Yes, it runs on Mono.
- Yes, it has namespaces (**no creative reading or writing**).
- Has TIPs #127, #178, #182, #194, #207, #241, #269, #285, #405, #426, #429, and #440.
- If all else fails, you can still use native Tcl from Eagle, completely seamlessly (e.g. Tk with WinForms or WPF, etc).

EYRIE SOLUTIONS™

Advanced Eagle

EYRIE SOLUTIONS™

Common Options & Idioms

- Typed options (bool, int, wide, enum, type, type list, etc).
- Flags values (enum with Flags attribute).
- Opaque object handles and reference counting.
- Using **`[object create]`**, **`[object invoke]`**, and **`[object dispose]`**.
- Options used for CLR / .NET Framework integration).

Typed Options

- Must have a value, e.g.: **`-name value`**
- Values must conform to the type.
- Boolean must be 0, 1, “false”, “true”, et al.
- Integers may be base 2, 8, 10, or 16.
- Enumerated values use the name of the value, e.g. “Red” for “ConsoleColor.Red”.
- Type values must resolve to the name of a loaded type (e.g. “Int32”, “String”, etc).

Flags Values

- Enumerated types decorated with the FlagsAttribute are treated specially.
- Like normal enumerated values, either the name (or an integer) may be used to specify a single value, e.g. “Space” or 0x2 for CharacterType.
- Unlike normal enumerated values, the value may be a list of values and each one may be prefixed by an “operator”.

Flags Operators

- Operate on the old flags and the new flags as their operands.
- The “+” (add) operator adds flags to the old flags.
- The “-” (remove) operator removes flags from the old flags.
- The “=“ (set) operator discards all flags in favor of the ones that follow.

Flags Operators (continued)

- The “&” (keep) operator retains only those flags that match the mask that follows.
- The “:” (set-then-add) operator initially discards all flags in favor of ones that follow and then switches to “+” (add) mode.

Loading Assemblies

- Managed assemblies may be loaded using the **`[object load]`** sub-command. If the reside in the Global Assembly Cache (GAC) or along the probing path for the application domain, no other options are necessary.
- To load an assembly from a specific location, the **`-loadtype File`** option must be used.

Type Names

- All type names must be fully qualified unless they have been imported, implicitly or explicitly, using the **`[object import]`** sub-command.
- By default, the following namespaces are implicitly imported into every interpreter:
  - **`System`**
  - **`Eagle._Components.Public`**
  - **`Eagle._Containers.Public`**

Opaque Object Handles

- Returned by various commands to refer to a specific CLR object.
- May be stored in any variable and will be automatically reference counted. When the reference count is zero, the object will be disposed.
- The default reference counting and disposal behavior may be overridden (which is sometimes necessary).

Opaque Object Handles (continued)

- Introspection via `[info objects]` and various `[object]` sub-commands.
- Reference counted via an interpreter-wide variable trace.
- When created with the `-alias` option, a command will be added as well, to permit invocation using the syntax:
  - `$handle MethodName arg1 … argN`

Opaque Object Handles (continued)

- The **`null`** opaque object handle is special.
- It always represents a null value; thus, it may only be used where the value of a reference type is expected.
- It cannot be modified or removed.
- It is always present, even in “safe” interpreters.
- There is a **`null`** global variable that points to it; therefore, it can be passed by value or by reference.

`[object create]`

- Used to create a CLR object of a specific type by calling one of its constructors.
- Returns an opaque object handle with a reference count of zero.
- Any parameters to the constructor should be passed immediately after the CLR type name.
- Example: `[object create Int32]`

`[object create]` (continued)

- Normally, the return value should be captured into a variable for subsequent use.

- The **`-alias`** option should be specified if the script intends to call one or more members on the object.

`[object invoke]`

- Used, directly and indirectly, to call a member on a CLR object or type (e.g. for static members).
- Returns a value based on the target member return type.
- Supported return types will be automatically converted to a string.
- To force object creation (i.e. instead of a string), use the **`-create`** option, with the **`-alias`** option if the script intends to call one or more members on the object.

EYRIE SOLUTIONS™

`[object invoke]` (continued)

- Yes, you can invoke that member.
- The general guidelines are:
  - If it does not work or does not do what you want, you probably need more flags.
  - If the types do not seem to match up correctly, you may need to use the **`-parametertypes`** option, which accepts a list of CLR type names corresponding to the ones for the desired method overload.

`[object invoke]` (continued)

- Supports fields, properties and methods.
- Supports all CLR types, including generics.
- Supports Nullable value types.
- Supports arrays, multi-dimensional, nested, etc.
- Supports ByRef for all of the above.
- Supports delegates (i.e. managed function pointers) with arbitrary method signatures.

EYRIE SOLUTIONS™

`[object invoke]` (continued)

- Unfortunately (?), there are a huge number of options available.
- Most options are needed rarely.
- Key options include:
  - The **`-flags`** option (with **`+NonPublic`**).
  - The **`-objectflags`** option (with **`+NoDispose`**).
  - The **`-marshalflags`** option (with **`+DynamicCallback`**).

Private Members?

- Yes, you can invoke any private member.
- In general, this should be considered as a “last resort”.

Eagle Core Library?

- Yes, you can invoke any member of any type in the Eagle core library (i.e. “Eagle.dll”).
- In fact, the test suite does this quite often in order to verify correct operation of various subsystems.
- Invoking private members in the Eagle core library is ***not guaranteed*** to work between versions as members may be added or removed and/or the method signatures may change.

Nested Objects & Members

- When using `[object invoke]`, the `object` and `member` arguments may use period delimited names.

- For the `object` argument:
  - The first part may be an opaque object handle or a type name.
  - Subsequent parts must be member names that do not require any parameters.

Nested Objects & Members (continued)

- For the **`member`** argument:
  - All parts, except the last one, must be member names that do not require any parameters.

Method Overload Resolution

- First, all methods for the target type are filtered by the selected binding flags.
- Next, methods are filtered by name.
- Next, methods are filtered by expected parameter counts (minimum / maximum versus arguments).

Method Overload Resolution (continued)

- Next, methods are filtered by the specified parameter types, if any.
- Next, methods are filtered based on the provided arguments.
  - Each argument is checked, in order, to see if it can be converted to the expected type for the method overload being checked.

Method Overload Resolution (continued)

- If the argument cannot be converted, the method overload is disqualified and errors may be logged for later use.

- If the argument can be converted, the new (typed) value is stored for later use.

- When attempting to convert an argument, the currently registered **`IBinder`** for the interpreter is consulted.

Method Overload Resolution (continued)

– Deeper down, the provided binder (i.e. **`ScriptBinder`**) consults the registered “change-type” callbacks, which may be customized.

– All primitive types provided by the CLR have built-in change-type callbacks registered with the built-in script binder.

– These can be overridden or removed if necessary.

Method Overload Resolution (continued)

- If the number of filtered method overloads so far exceeds the value for the **`-limit`** option, if specified, the filtering stops.
  - By default, there is no limit on the number of method overloads.

Method Overload Resolution (continued)

- Next (optional), the list of filtered method overloads is reordered based on the specified criteria.
  - To use this feature, specify the –**`marshalflags +ReorderMatches`** option.
  - Depending on the desired ordering, the –**`reorderflags`** option may also be necessary.

Method Overload Resolution (continued)

- Next, from the filtered list of method overloads, the final method overload is selected.
  - By default, the first one is selected.
  - This can be overridden using the **`-index`** option.

Method Overload Resolution (continued)

- If there is any ambiguity based on the specified arguments, using the –**`parametertypes`** option is recommended.
  - This will generally force the method overload with exactly the same parameter types to be selected.

ByRef Parameter Handling

- Before calling the target method, all output parameters must be resolved.

- Each argument to **`[object invoke]`**, et al, that corresponds to a ByRef parameter must be the name of an existing variable in the current call frame.

ByRef Parameter Handling (continued)

- The variables referenced must contain values that are compatible with the formal types associated with the method overload parameters.
- For arrays, the variable must be an array containing elements compatible with the formal type.
- Typically, this is accomplished using opaque object handles; however, that is not only way.

ByRef Parameter Handling (continued)

- For output parameters of reference types, the following pattern is somewhat common:

```
set varName null
object invoke $o Method varName
```

ByRef Parameter Handling (continued)

- Upon returning from calling the target method, any output (i.e. “by-reference”) parameters are extracted and set into the variable names specified in the original **`[object invoke]`** call.
  - Arrays are supported and the corresponding variable name must be undefined or a Tcl array of the necessary rank.

EYRIE SOLUTIONS™

ByRef Parameter Handling (continued)

- For each output parameter, the value will be handled just as though it were a return value.

- That means the information in the "Return Value Handling" section will apply to these values as well.

Return Value Handling

- The return value for the method overload is converted to a string.
  - Depending on the return type and value, this may involve creating a new opaque object handle.
  - By default, all primitive types that can be converted to a string **without losing any information** will be converted to a string.

Return Value Handling (continued)

  - When checking if a return value can be converted to a string, the registered script binder (and indirectly its registered “to-string” callbacks) are consulted.

- The default handling for all non-primitive types, including value types, is to create a new opaque object handle.

Introspection

- The `[object assemblies]` and `[object members]` sub-commands are quite useful.

- The `[object assemblies]` sub-command returns a list of managed assemblies that have been loaded into the current application domain, optionally filtered by a pattern.

Introspection (continued)

- The **`[object members]`** sub-command returns a list of methods, properties, and fields for an opaque object handle or type name, optionally filtered by a pattern.

- There are many options for **`[object members]`** that can make filtering and result handling easier.

Introspection (continued)

- Useful options for `[object members]` include:

```
    -matchnameonly
    -nameonly
    -mode
    -pattern
    -signatures
```

Iteration

- The **`[object foreach]`** sub-command can iterate over any object that implements the IEnumerable interface, including managed arrays, collections, etc.

Iteration (continued)

```
object import System.Reflection

set assms [object invoke \
  -create Assembly.GetEntryAssembly \
  GetTypes]

object foreach -alias t $assms {
  puts stdout [$t FullName]
}
```

EYRIE SOLUTIONS™

Questions & Answers

EYRIE SOLUTIONS™

Contact Information

- Eyrie Solutions
  sales@eyrie.solutions
- The Eagle Project
  https://eagle.to/
- Me (Joe Mistachkin)
  joe@mistachkin.com