Advanced RbScript

by Thomas Tempelmann

Advanced RbScript

Topics handled in this session

- First part
  - Pausing a script that is waiting for an event using a semaphore
  - Killing a thread that is suspended by a semaphore
  - Catching exceptions in scripts
- Second part
  - Exchanging object between the main program and its scripts

Pausing a script

- Situation:
  - A script has to wait for input that it can not get immediately
- Examples:
  - An event driven script that keeps running in a loop and waits for being told what to do
  - A script may invoke an operation through a Context provided method that shall only return when an Event handler gets called (asynchronous network operations)

Pausing a script

- When the script makes a call to a *Context*-provided method, that method cannot return immediately because it must wait for the event to happen first.
- To achieve this, a *Semaphore* is used. It allows us to stop suspend execution of the method and instead have other program code run. Once the event we are waiting for occurs, we can release the semaphore to let the method continue

Pausing a script

- In order to use a semaphore, we have to run the script in a separate process (*Thread*).

RbScript example

```
dim res as String

do
  res = Input ("gimme a cookie")
  if res = "cookie" then
    print "  yummy!"
  else
    print "  bah!"
  end
loop until res = "stop"
```

Code to invoke the script

```
// prepare the semaphore
ScriptLock = new Semaphore
ScriptLock.Signal

// run the script
RbScript1.Run ()
```

Script’s input processing

```
// this gets called when the script calls the Input method:

Function RbScript1.Input (prompt As String) As String
  ScriptLock.Signal
  return InputForScript
End Function

// this we call when we have input for the script:

Sub ResumeScriptWithInput (inputMsg as String)
  InputForScript = inputMsg
  ScriptLock.Release
End Sub
```

Killing a suspended thread

- Situation:
  - The script's thread is still running (waiting) when its environment (window or application) is closing. This may lead to unexpected effects including raised exceptions.
  - The Kill method of the Thread class does not work if the thread is locked by a semaphore, as it is in our case.

Killing a suspended thread

- The solution is to unlock the semaphore and have the freed code raise a ThreadEndException from inside the thread.
- The exception will have the effect that the script's loop is exited and eventually will end the script and return from the RbScript.Run() call in the thread. This in turn ends the thread properly.

Catching exceptions in scripts

- Situation:
  - When an exception occurs in a script that is not getting caught, the documentation suggests that the script's *RuntimeError* event gets called with information about the exception, similar to the App's *UnhandledException* event, then the script ends and returns normally from the RbScript.Run() call.
  - However, this is not happening.

Catching exceptions in scripts

- Situation (cont.):
  - Instead, what happens is this:
    - The RuntimeError event is *not* called
    - The Run() method does not return normally but instead an exception is propagated.
    - In some cases, the application may then crash.
  - This is a confirmed bug and present in RB 5.5.5 as well as in 2006r1

Catching exceptions in scripts

- Solution (work-around):
  - The script has to have a catch-all wrapper around its main code.
  - Problem is that it must not use a variable to learn the type of the exception - that would cause another crash. This means that we can only learn *that* an exception has occurred, but not which type it is, nor get a stack trace.
  - We use the Context to set a flag for reporting this incident to the Run() caller.

REALWorld

End of Part 1

- The project *EventDrivenScript.rbp* contains a sample application using the aforementioned techniques.

Passing objects to the script

- Goals:
  - We want to be able to let a script use objects in the same way we use them in our main program.
  - In particular, the script shall have access to some of the same data we access in our main program.

Example of use in a script

```
dim dir, f as FolderItem
dim idx as Integer

dir = DesktopFolder ()

for idx = 1 to dir.Count ()
  f = dir.TrueItem (idx)
  Print f.Name
next
```

Passing objects to the script

- Key steps to the solution:
  - We automatically add a set of classes to the user script that provide the classes and functions.
  - Those added classes communicate with the main program through specially provided *Context* methods that the common part of the script does not officially know about.

Passing objects to the script

- Key steps to the solution (cont.):
  - The added classes in the script create *proxy* instances of the main app’s objects.
  - To connect the app’s objects to their script counterparts, *Hash codes* are used...
  - These hash codes are Integer values that are unique to every existing object.

Passing objects to the script

- Key steps to the solution (cont.):
  - When the script wants to create a proxy instance of a main app’s object, it calls a context method, which finds the object and returns the object’s hash value to the caller in the script.
  - Later, if the script wants to access a property or member of such a proxy class, another context method is called along with the hash value for identification of the main app’s obect.

Passing objects to the script

- Key steps to the solution (cont.):
  - A Dictionary is used on the main app side to remember the objects the script proxy has requested.
  - The proxy class implements the *Destructor* method from which it tells the main app (though another Context method) to release the object again, i.e. remove it from the Dictionary.

Passing objects to the script

- The project *ScriptClassProxies.rbp* contains a sample application using this technique.

My thanks go to Joe Ranieri (aka. *SirG3*) for introducing me to this elegant way of using RbScript with complex data structures.

Demonstration

Q & A