Chapter 1

Managing External Resources

Managing external resources (such as fonts, files, sockets...) can be tedious. Pharo comes up with an external resource manager called NBExternalResourceManager developed by Igor Stasenko. This chapter is based on gorgeous class comments and help. We are thankful for them. So let us explain it a bit.

1.1 External Resource Manager

An external Resource Manager is responsible for managing the finalization of external resources.

When object, registered as external resource is garbage collected, the resource manager is telling the object's class to finalize its associated data (by passing an object, received from resourceData message sent to an object at registration time).

The External Resource Manager automatically keeps track for session change (image save/boot), and ignores finalization of resources of old sessions (since they are not longer valid, and cannot be freed since session changed).

Like that, a user of NBExternalResourceManager doesn't need to implement a session checking logic, and need only to:

1. a) register object as external resource:

[[[NBExternalResourceManager addResource: anObject.]]]

1. b) an object should understand the resourceData message, which is remembered at registration point (it can be any external resource like, id, handle or memory pointer).

Then, when object is garbage collected, its class will receive the message finalizeResourceData: to finalize the resource data passed as argument. The passed data is exactly same as previously returned by resourceData method.

1.2 Example

Imagine that you want to represent an external resource by keeping its handle.

```
Object subclass: #MyExternalObject
   instanceVariableNames: 'handle'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'XYZ'
```

To let your object(s) to be managed by external resource manager, you need to register it. Usually you do it after successfully claiming an external resource:

```
MyExternalObject>>initialize

   handle := self createNewExternalResource.  "claim resource"
   self assert: self handleIsValid. "etc..."

   "Now, register receiver as external resource"
   NBExternalResourceManager addResource: self

"Another form of use is:
   NBExternalResourceManager addResource: self data: handle.
"
```

If you used addResource: method for registration, you should provide an implementation of resourceData method:

```
MyExternalObject>>resourceData
   ^ handle "since we need only handle to identify external resource"
```

Now, for properly finalizing the external resource we should implement:

```
MyExternalObject class>> finalizeResourceData: aHandle
   ^ self destroyHandle: aHandle. "do whatever is needed to destroy the handle"
```

Note that in finalizeResourceData: you cannot access any other properties of your instance, since it is already garbage collected. You also don't need to do a session checking, since it is done automatically by resource manager.

1.3 Sessions and external resource management

When we are manipulating external resources, it is important to clean them. Now it is usefull to know whether a resource has been allocated during another restart of the image or not (Imagine that your system restarted and your cleaning process is not yet invoked). To help you in this task, NativeBoost offers you the notion of session. By session, here, we mean the way to identify uniquely a period of activity corresponding to an image start and end running by VM. Each time image starts from disk , it means that we start a new session.

You can use [NativeBoost>>uniqueSessionObject] for checking if session is changed.

This is useful when you are holding a handle(or pointer) to external resource and need to manage it. Since user may decide to save image at any moment, the objects holding a handles to external resources will be persisted in image as well, but then when image restarted, those handles are no longer valid and need special treatment (otherwise crash, boom, bang...)

You can, of course, use the well known approach by defining your own startUp method, to cleanup invalid handles, but then you also need to register your class for startup as well as make an educated guess about order in which it should perform a startup relative to other services in your image etc etc. Putting extra startup procedure also means slowing down an image bootingß time (because you most probably will use allInstances message), which sometimes is not desirable.

By using NativeBoost>>uniqueSessionObject we can avoid some of those hurdles (we can't avoid all), and instead implement a lazy (re)initialization scheme, or just do a validity check before attempting to access a possibly invalid external resource handle.

1.4 Lazy initialization strategy

Here is a possible way to handle external resource. For every entity (or group of entities), which works with external resources, keep an associated session object.

For example, let's suppose that we have a class representing an external library window and holding a handle of it:

```
Object subclass: #MyWindow
   instanceVariableNames: "handle"
   category: "MyPackage"
```

The approach is simple: every time we're going to access a resource handle (like passing a handle to external function), we should check that we're in same session, and therefore handle is still valid. If session is changed we can either reinitialize our external resource, or just signal an error (instead of crashing the VM).

So, first, let's add an instance variable, named session to the class, which will be associated with handle:

```
Object subclass: #MyWindow
   instanceVariableNames: "handle session"
   category: "MyPackage"
```

then , we can implement a simple method to check if we're in same session and therefore handle is still can be used:

```
MyWindow>>initializeWithHandle: aHandle
   session := NativeBoost uniqueSessionObject.
   handle := aHandle.
   ... do whatever you need ...
```

```
MyWindow>>checkSession
   session == NativeBoost uniqueSessionObject
      ifFalse: [ self initializeForNewSessionOrSignalError ].
```

Now in all places where we're going to use the handle, we put a simple self checkSession , like following:

```
MyWindow>>setTitle: aString
   self checkSession; privateSetTitle: aString
```

(here, the privateSetTitle: is a method which performs the FFI callout).

1.5 Finalization

We can, of course, use approach for identifying session(s) together with finalization scheme (to prevent resource leaks when object representing an external resource is garbage collected)

```
MyWindow>>initialize
   super initialize.
   session := NativeBoost uniqueSessionObject.
   WeakRegistry default add: self.
```

```
MyWindow>>finalize
   session == NativeBoost uniqueSessionObject
      ifTrue: [
         ... destroy the handle .. ]
      ifFalse: [ ... we don"t care.. session is different anyways ... ]
```

This is useful in situations, where we want to automatically free the external resources held by our application, when they are no longer in use (and therefore garbage collected). But since our object in question may survive an image snapshot, and be GC'ed in different session, we should actually check if we're in same session and only then tell external library to deallocate associated external resource, or just do nothing, since usually any external resources which were available in previous session(s) is void anyways.

1.6 Summary

We hope that you enjoyed this little perl of Pharo programming. Note that NBExternalResourceManager may be renamed and included in Pharo since it is a really usefull class.