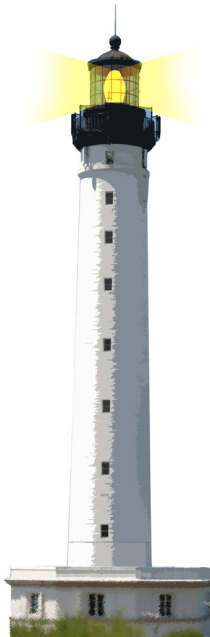
About Global Variables

S. Ducasse



Outline

- Singleton/Global is not nice because **globally** shared
- Difficult to test
- Singleton may take different forms
- Study some cases
- Think modular
- Messages need different instances to dispatch

Autopsy of an Error

```
MyApp >> menu
...
    icon: (Smalltalk icons iconNamed: #window)
...
```

Case 1: Global Variable

- One global variable
- What if as an application I want to extend, slightly change icons for my application only
- What if I want to be able to have two icon sets and the same time to compare

Case 2: A Disguised Global Variable

Since in Pharo we can extend core libraries we could think this is any better.

```
MyApp >> menu
...
   icon: #window asIcon
...
```

```
Symbol >> asIcon
   ^ Smalltalk icons iconNamed: self
```

Case 2: A Disguised Global Variable

```
MyApp >> menu
...
   icon: #window asIcon
...
```

- Does not duplicate Smalltalk icons iconNamed:
- This is already something!
- But still a global

Case 2: A Disguised Global Variable

- One global variable but disguished: only one place to edit but still fundamentally one giant global
- There is only one icon table
- MyApp cannot extend or slightly change icons for my application only!
- I cannot simply have two icon sets at the same time to compare them

A much better approach

```
MyApp >> menu
...
   icon: (self iconNamed: #window)
...
```

```
MyAppSuperclass >> iconNamed: aSymbol
  ...
  look for my icons (and may be delegate to an icon manager instance)
  potentially do a
  super iconNamed: #window
  ...
```

Why is this better?

- Modular
- **Each** receiver may do something **different**
- Each user may be **configured differently**
- Still we can share the common behavior

Case 3: asClass

Accessing programmatically a class is usually done as:

```
Smalltalk globals at: #Point
```

People wanted a shorter version

```
#Point asClass
```

```
Symbol >> asClass
  ^ Smalltalk globals at: self
```

- Shorter for scripting
- But there is a difference!
- A huge one...

Case 3: asClass Analysis

- Another global entry point
- What if we want to remotely access a class in another system
- We can only have one namespace
- We cannot inject a special namespace for test for example
- No way to dispatch to a different object

Case 3: Possible solution

Delegate to the class to get its environment

```
self class environment at: #Point
```

This supports different environments

Case 4: Smalltalk tools - The ugly

```
browseMethodFull
  "Create and schedule a full Browser and then select the current class and message."

  self currentClassOrMetaClass ifNotNil: [
    Smalltalk tools browser
      openOnClass: self currentClassOrMetaClass
      selector: self currentMessageName ]
```

Case 4: Smalltalk tools Analysis

```
browseMethodFull
  "Create and schedule a full Browser and then select the current class and message."

  self currentClassOrMetaClass ifNotNil: [
    Smalltalk tools browser
      openOnClass: self currentClassOrMetaClass
      selector: self currentMessageName ]
```

- One global entry point
- Everybody refers to this single point!
- Yes this is called monolithic thinking
- Only one toolset possible at the same time (could be ok).

Case 4: Smalltalk tools Possible Solution

- Each object that should refer to tools should do it via a parameter / instance variable and messages
- Avoid direct reference to a global singleton

```
Browser >> initialize
  toolEnvironment := ToolEnvironment new
```

```
Browser >> openDebugger
  self toolEnvironment debugger
```

Points to consider

- With a global, when it changes, all its users are updated
- How to manage the fact that a tool may change?
- Browsers may register to ToolEnvironment to be notified and update its instance

Conclusion

- Avoid Singleton as a global
- Think modular
- Give a chance to objects to specialize messages

A course by

S. Ducasse, L. Fabresse, G. Polito, and Pablo Tesone