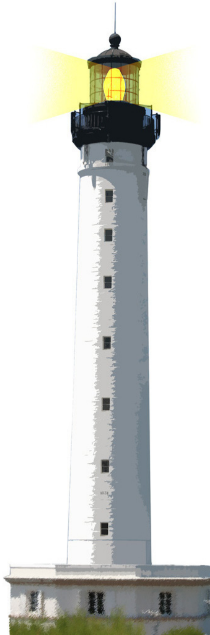
Global To Parameter

Basic but important

S. Ducasse

http://www.pharo.org



Goal

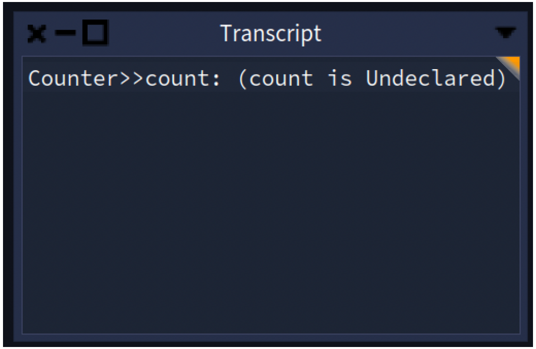
- Globals are not a fatality
- Some can be turned as computation parameters (such as instance variables)
- There are pros and cons

Roadmap

- Example: Analysis of Transcript usage
- Cure
- Analysis
- Other analysis
- Related to the plague of Singleton Design Pattern

The case: Transcript

Remember: Transcript is a global variable pointing to a log stream instance



```
Counter>>count: (count is Undeclared)
```

Handy in development

```
myMethod
  Transcript show: 'foo' ; cr.
  self doSomething.
```

The core of the problem

When not in development

```
MicAbstractBlock >> iterate
  ...
  Transcript
    nextPutAll: 'Start ';
    nextPutAll: step asString;
    cr.
  ...
  Transcript
    nextPutAll: 'Stop ';
    nextPutAll: step asString;
    cr.
```

- What if I would like to have a specific log for Microdown?
- What if we want to test that such logs are correct?

Analysis

Some facts:

- You may not want the extra dependencies (such as Transcript) in your code
- With Transcript, **your** log can be mixed with **other** logs
- You do not want to **dirty** build log without a bit of control

Far worse and more important:

- You cannot reliably write tests to be sure that the log is correctly happening

The solution: Use locality and encapsulation

- Think about object self-containment
- What is the object encapsulate a log stream
- Add an instance variable to hold a stream

```
MicAbstractBlock >>initialize
  super initialize.
  logStream := WriteStream on: (String new: 1000)
```

- Use and write to THAT stream

```
MicAbstractBlock >>closeMe
  logStream << 'Closing ' << self class name; cr
```

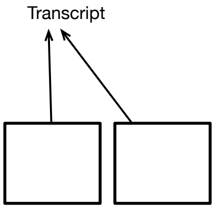
Get the butter and the money

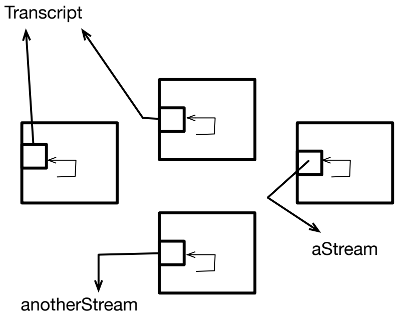
- Make sure that you can plug another stream as a logstream

```
MicAbstractBlock >> logStream: aStream
  logStream := aStream
```

- Now you can pass a Transcript and get the same as before but better.
- Bonus: You can write **tests in isolation**

Comparing





Do you see the pattern?

```
RubScrollTextMorph >> defaultScrollTarget
  | textArea |
  textArea := self textAreaClass new.
  textArea backgroundColor: Color lightGray veryMuchLighter.
  ^ textArea
```

- Why `Color lightGray veryMuchLighter` is hardcoded?

A solution

- Make it configurable!

```
RubScrollTextMorph >> defaultScrollTarget
  | textArea |
  textArea := self textAreaClass new.
  textArea backgroundColor: defaultBackgroundColor.
  ^ textArea
```

```
RubScrollTextMorph >> initialize
  defaultBackgroundColor := Color lightGray veryMuchLighter
```

Supporting personalisation

```
RubScrollTextMorph >> setBackgroundColor: aColor
  defaultBackgroundColor := aColor
```

Now each instance can have its specific value!

Instance variables

Instance variables are parameter of your computation

About Globals

Pros:

- You do not have to add an instance variable to your domain
- You do not have to initialize such global on your specific case

Cons:

- You have **only one** (e.g., if an entity belongs to one global model, you cannot have two entites living in different models)
- Testing requires care and is sometimes **not possible** or cumbersome because of **side effects**
- You cannot **initialize**, **specialize** the global for your context (there is only one)

About parametrization

Sometimes you cannot add an instance variable to your objects

- Too many of them
- Fixed size inherited from old design
- About space consumption, check Sharing Lectures and Flyweigth

At least factor the global usage to ease future changes

In general: Avoid Globals

- Avoid Singleton
- Avoid Globals
- They make your code less modular, less **testable**
- Check Lectures on Singleton and Disguised Singleton

A course by

S. Ducasse, G. Polito, and Pablo Tesone