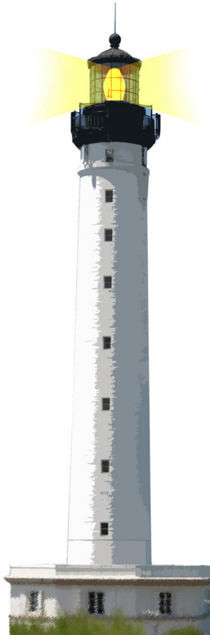
Advanced Object-Oriented Design

SharedPools

Static sharing across hierarchies

S. Ducasse



http://www.pharo.org

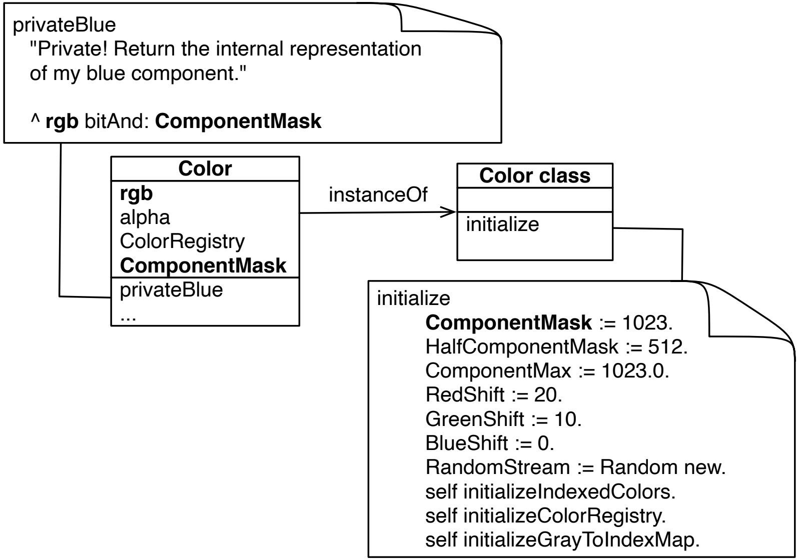
Goal

- Using shared variables, we can share values over multiple subclasses within the **same** hierarchy.
- How can we share objects across **different** hierarchies?

Remember: Sharing within a hierarchy

- A shared variable can be accessed from the instance and class side of a class
- But also from its subclasses
- Usually a shared variable is initialized from the class side.

Remember ComponentMask



```
privateBlue
   "Private! Return the internal representation
   of my blue component."

   ^ rgb bitAnd: ComponentMask
```

```
initialize
        ComponentMask := 1023.
        HalfComponentMask := 512.
        ComponentMax := 1023.0.
        RedShift := 20.
        GreenShift := 10.
        BlueShift := 0.
        RandomStream := Random new.
        self initializeIndexedColors.
        self initializeColorRegistry.
        self initializeGrayToIndexMap.
```

Need for sharing across different hierarchies

- Sometimes we need to share values (generally constants) over **multiple** hierarchies:
- For example LF, CR, ... in String and Text, =Days
- We don't want to repeat the shared variables and their initialization.

SharedPools to the rescue

A SharedPool is a kind of group of shared variables:

- It contains the definition
- the initialization of shared variables

Users (classes) just have to declare that they use a shared pool to get access to the values.

A SharedPool definition

```
SharedPool << #ChronologyConstants
  slots: {};
  sharedVariables: { #NanosInSecond . #MonthNames . #SecondsInHour .
      #SecondsInDay . #DayNames . #DaysInMonth . #HoursInDay . #NanosInMillisecond
      . #SecondsInMinute . #SqueakEpoch . #MinutesInHour . #MicrosecondsInDay };
  tag: 'Chronology';
  package: 'Kernel'
```

A SharedPool initialization

```
ChronologyConstants class >> initialize
  "ChronologyConstants initialize"
  SqueakEpoch := 2415386.      "Julian day number of 1 Jan 1901"
  SecondsInDay := 86400.
  MicrosecondsInDay := SecondsInDay * 1e6.
  SecondsInHour := 3600.
  SecondsInMinute := 60.
  MinutesInHour := 60.
  HoursInDay := 24.
  NanosInSecond := 10 raisedTo: 9.
  NanosInMillisecond := 10 raisedTo: 6.
  DayNames := #(Sunday Monday Tuesday Wednesday Thursday Friday Saturday).
  MonthNames := #(January February March April May June July
      August September October November December).
  DaysInMonth := #(31 28 31 30 31 30 31 31 30 31 30 31).
```

Shared pools are initialized at class load time.

SharedPool users

```
Magnitude << #DateAndTime
  slots: { #seconds . #offset . #julianDayNumber . #nanos };
  sharedVariables: { #ClockProvider . #LocalTimeZoneCache };
  sharedPools: { ChronologyConstants };
  package: 'Kernel'
```

DateAndTime

- defines some shared variables
- uses the shared pool ChronologyConstants

SharedPool’s sharedVariable access

- A shared variable defined in a shared pools is accessed as if it would be defined in the class itself.

```
DateAndTime >> secondsSinceMidnightLocalTime
  ^ self localSeconds \\ SecondsInDay
```

```
Duration class >> days: aNumber

  ^ self seconds: aNumber * SecondsInDay nanoSeconds: 0
```

SecondsInDay is just accessed directly both from the instance or class side.

SharedPool users (2)

```
Timespan << #Week
  slots: {};
  sharedVariables: { #StartDay };
  sharedPools: { ChronologyConstants };
  package: 'Kernel−Chronology−Extras'
```

```
Week class >> indexOfDay: aSymbol

  ^ DayNames indexOf: aSymbol
```

Mixing shared variables and sharedPools

There is no problem mixing shared variables and shared pools.

```
Timespan << #Week
  sharedVariables: { #StartDay };
  sharedPools: { ChronologyConstants };
  package: 'Kernel–Chronology–Extras'
```

```
Week class >> startDay
  ^ StartDay ifNil: [ StartDay := DayNames first ]
```

Warning! Only for constants

- Only store non constant objects in shared pools
- Else you are creating global variables and you are breaking testability in isolation

Conclusion

Shared pools are

- Handy to share constants over multiple classes
- Handy to manage constants for bindings to C-libraries
- Only use them to share constants

A course by

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