12. Exemplary Solutions: Virtual Machines and Repetition

Exercise 12.1:

```
markAndSweepGC
   self mark.
   self sweep.
   self removeMarks.

mark
   SystemNavigation default allObjectsDo: [:obj | self markObject: obj].

markObject: anObject
   anObject mark.
   1 to: (self numberOfFieldsOf: anObject) do: [:index |
      self markObject: (self fetchPointer: indes ofObject at: anObject) ]

sweep
   SystemNavigation default allObjectsDo: [:obj |
      obj isMarked ifFalse: [self releaseObject: obj ] ]

removeMarks
   SystemNavigation default allObjectsDo: [:obj | obj unmark ].
```

Exercise 12.2:

```
Object subclass: #LoadAverageSampler
   instanceVariableNames: 'loads index process'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'LoadAvg'

LoadAverageSampler >> open
   self isStarted ifFalse: [ self start ].
   (RectangleMorph new)
      layoutPolicy: TableLayout new;
      layoutInset: 10;
      listDirection: #topToBottom;
      hResizing: #shrinkWrap;
      vResizing: #shrinkWrap;
      addMorphBack: ((UpdatingStringMorph on: [ self printString ]
                                          selector: #value)
         stepTime: 1000;
         minimumWidth: 170;
         growable: true);
```

```
      color: Color white;
      borderColor: Color black;
      openInWindowLabeled: 'Load Avg'.

LoadAverageSampler >> start
   self stop.
   loads := IntegerArray new: self defaultNumberOfMeasurements.
   index := 0.
   process := [ self sampleBlock repeat ] forkAt: self defaultPriority.

LoadAverageSampler >> stop
   process isNil ifFalse: [ process terminate ]

LoadAverageSampler >> loadAverageForMinutes: aNumber
   | size measurements |
   aNumber > self defaultTotalSamplingMinutes
      ifTrue: [ self error: 'Out of bounds' ].

   size := aNumber * 60 / self defaultSamplingPeriod.
   measurements := self lastLoads: size.
   ^ measurements average roundTo: 0.01

LoadAverageSampler >> printString
   ^ String streamContents: [ :stream |
      #(1 5 15)
         do: [ :each |
               stream nextPutAll: ((self loadAverageForMinutes: each)
                  printShowingDecimalPlaces: 2) ]
         separatedBy: [ stream nextPutAll: ', ' ] ]

LoadAverageSampler >> isStarted
   ^ process notNil

LoadAverageSampler >> lastLoads: aNumber
   ^ (index - aNumber to: index) collect: [ :each |
      loads atWrap: each ]

LoadAverageSampler >> recordLoad: aNumber
   loads atWrap: (index := index + 1) put: aNumber

LoadAverageSampler >> sampleBlock
   ^ [
      (Delay forSeconds: 5) wait.
      self recordLoad: Processor currentLoad ]
```

```
LoadAverageSampler >> defaultNumberOfMeasurements
   "We take 1 measurement every defaultSamplingPeriod seconds for
    defaultTotalSamplingMinutes"
   ^ self defaultTotalSamplingMinutes * 60 / self defaultSamplingPeriod

LoadAverageSampler >> defaultPriority
   ^ Processor userInterruptPriority

LoadAverageSampler >> defaultSamplingPeriod
   ^ 5

LoadAverageSampler >> defaultTotalSamplingMinutes
   ^ 15
```

Exercise 12.3:

1. A class instance variable is like an instance variable, but on the class side. Thus with a class instance variable, we model the private state of this class. Instances of this class do not see this private state. As for instance variable, a class instance variable is also shared among sub(meta)classes. A class variable, however, is shared among all instances of a class and subclasses and can be read in instance- and class-side methods of all these classes.

2. `self` is dynamically bound to the receiver of a message send, `super` is statically bound to the method in which it is written, independently from the receiver. Thus with `super` you always directly point to the superclass of the class in which you use `super` while with `self` you point to the current receiver object of a message send, which might be an instance of sub-classes of the class in which you write `self`.

3. `thisContext` provides a reification to the stack, that is, the execution flow up to the current method in which `thisContext` is evaluated. Concretely `thisContext` points to the current stack frame from which we can go back to previous stack frames using #sender.

4. A metaclass is the class of a class. Each class has exactly one associated metaclass created automatically and transparently by the system. The so-called class-side method are actually just methods of this anonymous metaclass. As each class is an object, a metaclass is an object too. Thus Metaclass inherits from Object.

5. A good example for double dispatch can be found when considering Integer and Float:

```
Integer >> + aNumber
    ^ aNumber sumFromInteger: self

Float >> + aNumber
    ^ aNumber sumFromFloat: self

Integer >> sumFromInteger: anInteger
    <primitive: 40>
```

```
Float >> sumFromInteger: anInteger
    ^ anInteger asFloat + self

Integer >> sumFromFloat: aFloat
    ^ aFloat + self asFloat

Float >> sumFromFloat: aFloat
    <primitive: 41>
```

6. A Feature Envy is a code smell, for instance a method or parts of a method which accesses too much information from one or several external objects. Thus this methods is tightly coupled to other external objects, which make the method hard to understand, maintain and evolve. This undesirable situation can be tackled by moving either functionality from this method to the other objects or from the other objects directly to this method, if appropriate.

Exercise 12.4:

What are the results of the following expressions?

```
1. 'Hello'
2. 42
3. 6
```

Exercise 12.5:

1.

```
#(3.4 5)
```

is also answered by

```
#(1 2 3.4 5) reject: [:each | each <= 3]
```

2.

```
detect: aBlock ifNone: exceptionBlock
   "Evaluate aBlock with each of the receiver's elements as the argument.
   Answer the first element for which aBlock evaluates to true. If none
   evaluate to true, then evaluate the argument, exceptionBlock."

   self do: [:each | (aBlock value: each) ifTrue: [^ each] ].
   ^ exceptionBlock value.
```

3.

```
#(3 5 7 9)
```

Exercise 12.6:

1. We simply implement two methods `isBehavior`, one in `Object` and one in `Behavior`:

```
Object >> isBehavior
    ^ false

Behavior >> isBehavior
    ^ true
```

2. `#new` is implemented in `Behavior, Behavior >> new`

3.

- `Integer class class => Metaclass`
- `Metaclass superclass => ClassDescription`
- `Class isKindOf:  Object => true`

Exercise 12.7:

Answer the following questions:

1. Press on '++': The counter increases by one, that is, '1' is shown.
2. Back button: '0' is shown again.
3. Press on '++' again: '2' is shown.
4. Fix for for back button: Implement a `#states` methods to answer an array with `self` as its sole value.

Exercise 12.8: Bytecode

```
tt: aPoint
    ^ aPoint extent: 4@3.
```