Final CAP

Curs 2022-23 (13/1/2023) Duració: 3 hores

1.- (1 punt) Defineix *reificació*, *introspecció* i *intercessió*. Dóna exemples d'introspecció i intercessió en Smalltalk.

Solució:

Les definicions són a les transparències.
- Introspecció: Poder *veure* la pila d'execució amb **`thisContext`**.
- Intercessió: Poder *canviar* la pila d'execució, gràcies a **`thisContext`**.

2.- (1 punt) El caràcter **^** serveix per indicar el retorn d'un objecte des d'un mètode. Expliqueu què passa quan es fa servir dins un bloc. Serveix per retornar un valor del bloc? Si no és així, quin és l'efecte de **^** dins un bloc?

Solució:

El caràcter **^** serveix, efectivament, per retornar un objecte des d'un mètode. Si es fa servir dins un bloc, *no es retorna del bloc*, sinò que el que indiquem és un retorn *des del mètode on es va crear el bloc*. El valor retornat per un bloc és el valor de la darrera expressió avaluada dins del bloc.

3.- (1 punt) Hi ha cap diferència entre aquestes dues expressions:

a/ `Continuation callcc: [ :k | k value: 1@3 ]`

b/ `Continuation callcc: [ :k | 1@3 ]`

Explica quines són les diferències, si n'hi ha. En altre cas, explica per quina raó són equivalents.

Solució:

Externament no hi ha cap diferència. Sempre que s'envia el missatge `#callcc:` s'avalua el bloc passat com a paràmetre. En el cas **b** es retorna `1@3` i s'ignora la continuació, en el cas **a** s'invoca la continuació i se li passa com a argument `1@3`. En aquest cas el que passa és que la continuació és *precisament* la continuació del moment en que s'ha enviat el missatge `#callcc:`, per tant l'objecte `1@3` es retorna al mateix lloc que en el cas **b**.

Internament sí que hi ha diferència, ja que en el cas **b** no es fa res amb la pila d'execució, però en el cas **a** la pila és sustituida per ella mateixa en el moment d'invocar la continuació, la qual cosa representa una feina addicional.

4.- (2 punts) Fes petits bocinets de Smalltalk (per ser executats al *Playground*) per respondre les següents qüestions:

- Suposant que la variable `metode` conté un selector de mètode, obriu un *browser* amb tots els mètodes que facin servir aquest selector en el seu codi.
- Quants mètodes tenen la *string* `this` en el seu codi font?
- Quantes classes implementen el mètode `#new`?
- Hi ha mètodes per trobar objectes que contemplen la possibilitat que allò que es busca no es

trobi. Aquests mètodes acostumen a portar una *keyword* `ifAbsent` en el seu selector. Obriu un *browser* amb aquells mètodes que portin la *string* `ifAbsent` en el seu selector.

Solució:

```
a) SystemNavigation default browseAllSendersOf: metode.

b) (SystemNavigation default allClasses)
     inject: 0
     into: [ :coll :c |
                 coll + (c methodDict select: [ :v |
                        (v sourceCode findString: 'this') ~= 0 ] ) size ]

c) (SystemNavigation default allImplementorsOf: #new) size

d) SystemNavigation default browseMethodsWhoseNamesContain: 'ifAbsent'
```

5.- (2.5 punts) Aquest fragment de codi, executat al *playground*:

```
((Continuation callcc: [ :k | k ]) value: [ :x | 'aplico id' traceCr. x ]) value: 'HEY!'
```

retorna la *string* `'HEY!'` i escriu *dues* vegades `'aplico id'` al *Transcript*. Podeu comprovar-ho vosaltres mateixos. Expliqueu detalladament què passa en l'execució d'aquest codi.

Solució:

Comencem executant l'enviament de missatge `Continuation callcc: [ :k | k ]`, que, com ja sabem, retorna la continuació `k`, és a dir, el fragment de codi que espera un valor en el moment de l'enviament de missatge `#callcc:`:

```
k ≡ (□ value: [ :x | 'aplico id' traceCr. x ]) value: 'HEY!'
```

on el "*forat*" és on la continuació espera un valor, el valor que se li passarà com a argument quan s'invoqui. Aquesta continuació tot seguit s'invoca, *ignorant tot allò que queda per fer*:

```
k value: [ :x | 'aplico id' traceCr. x ]
```

Penseu-ho bé, això és correcte. Per quina raó ha desaparegut el `value: 'HEY!'` del codi original? Per quina raó no tenim:

```
(k value: [ :x | 'aplico id' traceCr. x ]) value: 'HEY!'
```

Senzillament perquè forma part *del que queda per fer* i que la invocació de la continuació eliminarà. El que no s'elimina és l'argument que se li passa a la continuació en invocar-la. Així doncs:

```
k value: [ :x | 'aplico id' traceCr. x ]  ≡

(□ value: [ :x | 'aplico id' traceCr. x ]) value: 'HEY!') value:
                                                    [ :x | 'aplico id' traceCr. x ]
```

El que queda finalment és:

```
([ :x | 'aplico id' traceCr. x ] value: [ :x | 'aplico id' traceCr. x ]) value: 'HEY!'  ≡
```

```
(s'avalua el bloc [ :x | 'aplico id' traceCr. x ] amb ell mateix com a argument, escriu
'aplico id' al Transcript i retorna el que se li ha passat com a argument)

≡  [ :x | 'aplico id' traceCr. x ] value: 'HEY!'   ≡

(s'avalua el bloc [ :x | 'aplico id' traceCr. x ] amb la string 'HEY!' com a argument,
escriu, un altre cop, 'aplico id' al Transcript, i retorna 'HEY!', que és el que se li ha
passat com a argument)

≡ 'HEY!'
```

6.- (2.5 punts) Un dels exemples de reflexió que vam veure a classe va ser com afegir mètodes en temps d'execució. Ho vam il·lustrar mitjançant l'exemple de la classe `DynamicAccessors`, on es creava el *getter* d'una variable d'instància en fer-lo servir. Modifica ara el mètode `#doesNotUnderstand:` de la classe `DynamicAccessors`, i els mètodes `testAccessors` i `tearDown` de la classe `DynamicAccessorsTest`, per afegir el tractament dels *setters* per a les variables d'instància. És a dir, en l'exemple teniu la creació del *getter* quan es fa servir i no hi és, feu ara que el mètode `#doesNotUnderstand:` afegeixi el *getter* i el *setter* quan es faci servir qualsevol dels dos per a variables d'instància que no els tenen definits. A més, us demano que modifiqueu els tests per comprovar que la modificació funciona bé.

Solució:

La classe `DynamicAccessors` era, recordem:

```
Object subclass: #DynamicAccessors
	instanceVariableNames: 'x y'
	classVariableNames: ''
	package: 'Reflection-Complete'
```

El codi original de l'exemple és:

```
doesNotUnderstand: aMessage
	| messageName |

	messageName := aMessage selector asString.

	(self class instVarNames includes: messageName)
		ifTrue: [self class compile: messageName , String cr , ' ^ ' , messageName.
			^ aMessage sendTo: self].

	super doesNotUnderstand: aMessage

testAccessors
	self deny: (DynamicAccessors methodDict keys includes: #x).
	self assert: DynamicAccessors new x = nil.
	self assert: (DynamicAccessors methodDict keys includes: #x).
```

on també hi ha:

```
tearDown
	DynamicAccessors removeSelector: #x.
```

Aleshores, les úniques modificacions que cal fer en el mètode `#doesNotUnderstand:` són (indicades en vermell):

```
doesNotUnderstand: aMessage
	| s messageName |
```

```
	"Aquí senzillament treiem els ':' del final del selector si és un setter"
	s := aMessage selector asString.
	messageName := (s last = $:) ifTrue:  [ s allButLast ]
	                             ifFalse: [ s ].

	(self class instVarNames includes: messageName)
		ifTrue: [self class compile: messageName , String cr , ' ^ ' , messageName.
			"Aquí afegim el setter"
			self class compile: (messageName , ': anObject') ,
			                    String cr ,  '    ' , messageName , ' := anObject'.
			^ aMessage sendTo: self].

	super doesNotUnderstand: aMessage
```

Per comprovar que funciona bé podem modificar els tests de moltes maneres. Una possible solució seria:

```
testAccessors
	| t |
	self deny: (DynamicAccessors methodDict keys includes: #x).
	self deny: (DynamicAccessors methodDict keys includes: #x:).
	self deny: (DynamicAccessors methodDict keys includes: #y).
	self deny: (DynamicAccessors methodDict keys includes: #y:).
	t := DynamicAccessors new.
	t x: 2.
	self assert: t x = 2.
	self assert: t y = nil.
	t y: -2.
	self assert: t y = -2.
	self assert: (DynamicAccessors methodDict keys includes: #x).
	self assert: (DynamicAccessors methodDict keys includes: #x:).
	self assert: (DynamicAccessors methodDict keys includes: #y).
	self assert: (DynamicAccessors methodDict keys includes: #y:).
```

on:

```
tearDown
	DynamicAccessors removeSelector: #x.
	DynamicAccessors removeSelector: #x:.
	DynamicAccessors removeSelector: #y.
	DynamicAccessors removeSelector: #y:.
```