Exercicis de reflexió amb Smalltalk

1. Fes un programa (per ser executat al workspace) que, donat un símbol que representarà el selector d'un mètode, generi un browser amb tots els mètodes que l'utilitzen dins el seu codi font.

```
| simbol |
simbol := #callcc:.
Object withAllSubclasses do: [ :objecte |
    objecte methodDictionary do: [ :mètode |
        (mètode sendsSelector: simbol) ifTrue: [
            "Transcript show: objecte ; cr."
            "Transcript show: mètode selector ; cr ; cr."
            Nautilus openOnMethod: mètode.
        ]
    ]
]
```

Aquest codi, a partir de `Object` mira en ell i totes les seves subclasses (`Object withAllSubclasses do: [ :objecte |`) tots els seus mètodes (`objecte methodDictionary do: [ :mètode |`), i en tots els mètodes, si aquest invoca al símbol (`(mètode sendsSelector: símbol) ifTrue: [`) l'obra al **Nautilus**.

Pots treure els comentaris dels transcript i comentar el Nautilus per veure l'objecte i el mètode que el crida.

Segurament hi ha alguna manera més eficient de fer-ho: El Jordi Fructos m'ha donat una solució més senzilla. Utilitza el mètode `CompiledMethod>>#sendSelector: aSelector`.

```
| simbol |
simbol := #moneda:.
Nautilus openInEnvironment: (RBBrowserEnvironment new
    forClasses: Object withAllSubclasses;
    selectMethods: [ :method | method sendsSelector: simbol])
)
```

2. Per quina raó s'avalua a true la seguent expressió?

```
Class class class class == Class class class class class class
```

`Class class class` és `Metaclass`. A `Metaclass`, al demanar la classe, ens donarà `Metaclass class`, i al demanar la classe a `Metaclass class` ens dona `Metaclass`. Aqui ja entrem en bucle, així que si demanes `class` de dos en dos sempre donarà el mateix.

3. Els mètodes `#new` i `#new:` són mètodes d'instància de la classe Behavior, tot I que és habitual que se'ls redefineixi en altres classes. Malgrat són mètodes d'instància de Behavior, usualment es redefineixen com a mètodes de classe. Per exemple, utilitzem `Array new: 5` per crear instàncies de la classe Array, i el missatge `#new:` l'estem enviant a la classe Array. Les redefinicions, doncs, s'acostumen a fer en el Class side. Això aparentment viola la "regla" que diu que en l'herència els mètodes d'instància s'hereten en l'instance side i els mètodes de classe s'hereten en el Class side. Expliqueu per quina raó no hi ha res d'incorrecte en el fet de redefinir `#new` i `#new:` en el Class side.

Doncs perque el class side es la metaclasse, i totes les metaclasses hereten de la classe Class, aquesta hereta de ClassDescription i aquesta de Behavior. Aixi que totes les metaclasses acaben heretan de Behavior.

4. Expliqueu com un bloc pot cridar-se a ell mateix (aconseguint així blocs sense nom, anònims, però recursius). No podeu suposar que el bloc ha estat assignat a una variable. Feu-ho servir per fer un bloc que calculi el factorial.

```
| fatorial |
fatorial := [ :x |
    | codi |
    codi := thisContext closure.
    (x = 1)
        ifTrue: [ 1]
        ifFalse: [ x * (codi value: x - 1) ].
    ].
Transcript show: (fatorial value: 5) asString ; cr.
```

Dins del bloc podem accedir a thisContext, que es el MethodContext del bloc. I allà accedir al mateix bloc (`codi := thisContext closure`). Després executar el mateix codi. La variable factorial no s'utilitza dins del bloc.

5. Escriviu un mètode `find: aString` que, enviat a una classe, retorni una col · lecció de selectors tal que els mètodes corresponents contenen aString dins del seu codi font.

A la classe `Object`:

```
find: aString
    ^(self methods)
        select: [ :m | (m sourceCode findString: aString) ~= 0 ];
        inject: Set new into: [ :set :m | set add: m selector. set ].
```

6. Escriviu un fragment de codi (per ser executat en el *workspace*) que ens digui quants mètodes tenen la String '*this*' en el seu codi font.

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

7. A classe vam veure un mètode anomenat `#haltIf: aSymbol` que permetia fer un halt només quan a la cadena de crides hi havia un mètode amb `aSymbol` com a nom. Feu ara un mètode `#haltIfTrue: aBlock` on aBlock és un bloc sense paràmetres. Aleshores, `self haltIfTrue: [...]` s'aturarà només si el resultat d'avaluar `[...]` és true. A quina classe ha d'anar aquest mètode?

A la classe `Object`:

```
haltIf: aBlock
    aBlock value ifTrue: [ Halt halt ]
```

8.

a) Què fa aquest codi?

```
#aa bindTo: 1 in: [
    Transcript show: #aa binding asString; cr.
    #aa bindTo: 2 in: [
```

```
        Transcript show: #aa binding asString; cr. ].
    Transcript show: #aa binding asString , ' again'; cr. ].
```

Dona un valor diferent al simbol `#aa` depenent del bloc en el que esta. En el primer bloc, és a dir, en la linia 2 i 5, `#aa` val 1. Quan entra al segón bloc `#aa` val 2.

b) I aquest codi?

```
#aa bindTo: '1' in: [
    #bb bindTo: 'a' in: [
        Transcript show: #aa binding , '-' , #bb binding; cr.
        #aa bindTo: '2' in: [
            #bb bindTo: 'b' in: [
                Transcript show: #aa binding , '-' , #bb binding; cr. ].
            Transcript show: #aa binding , '-' , #bb binding; cr. ].
        #bb bindTo: 'b' in: [
            Transcript show: #aa binding , '-' , #bb binding; cr ]]]
```

Va cambiant els valor de `#aa` i `#bb` segons el bloc. De fora a dins: `#aa` = '1', `#bb` = 'a', `#aa` = '2', `#bb` = 'b' i després `#bb` = 'b'.