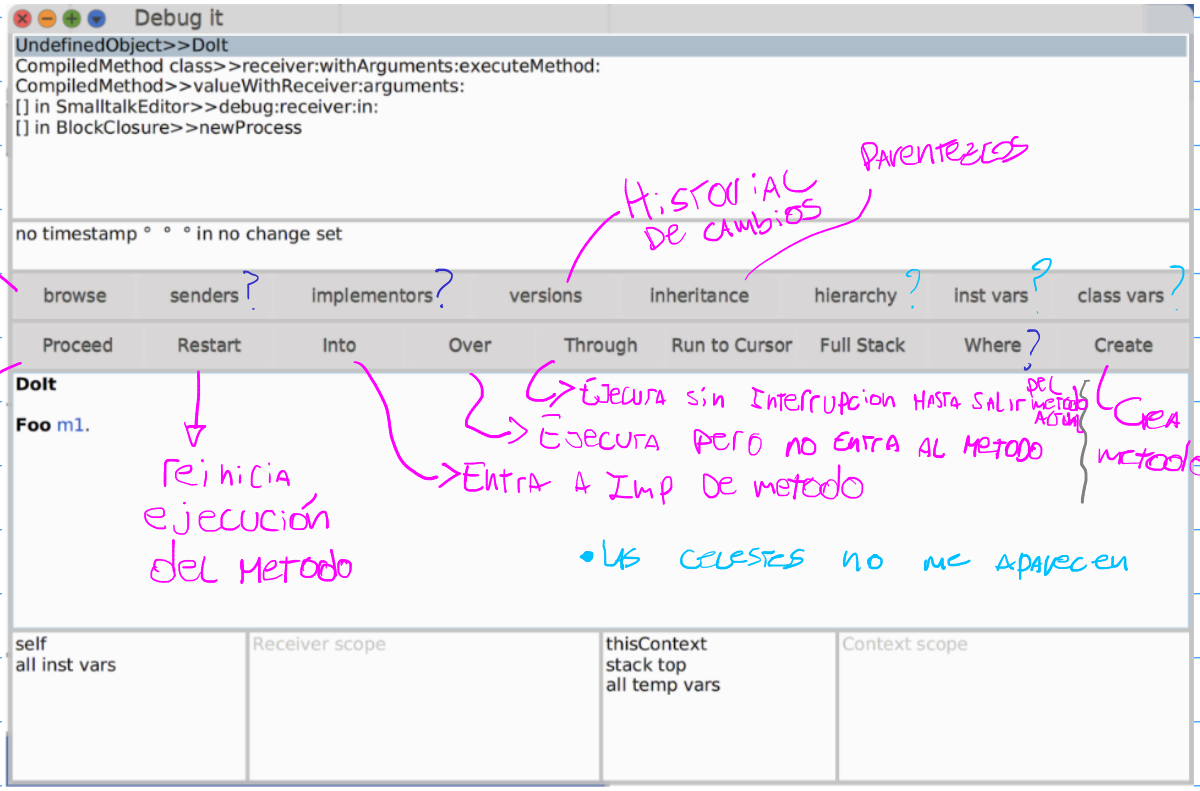
0[1]. Debugger

0.1 Identificar y nombrar las diferentes partes y secciones del debugger:



1. Colecciones

1.1 Acerca de algunas colecciones muy utilizadas

a. Array (fixed length collection)

x := Array with: 5 with: 4 with: 3 with: 2.

Sintaxis reducida para crear arrays:

x := #(5 4 3 2).

Para resolver en el Workspace:

- Crear un array usando alguna de las sintaxis anteriores.
- Cambiar el elemento en la primera posición con el valor 42.
- ¿Qué pasa si queremos agregar un elemento en la posición 5?

x := #(1 2 3)

x at: 2 put: 7

Error

Array(Object)>>errorSubscriptBounds:

"Create an error notification that an improper integer was used as an index."

b. Ordered Collections → Listas :)

x := OrderedCollection with: 4 with: 3 with: 2 with: 1.

Resolver en el Workspace:

- Agregar elemento 42.
- Agregar elemento 2.
- ¿Cuántos elementos tiene la colección?
- ¿Cuántas veces aparece el 2?

x add: 42

x add: 2

x size. 6

2 veces

Consultar que sería "; & yourself"

d. Dictionary

x := Dictionary new.
x add: #a->4; add: #b->3; add: #c->1; add: #d->2; yourself.

Esto lo hice sin pasar x, pero los métodos son ⊜

Resolver en el Workspace:

- Agregar la *key* #e con el *value* 42.
- ¿Cuántos elementos tiene la colección?
- Listar las keys.
- Listar los values
- Obtener el value del key #a.
- Obtener el value del key #z (en caso de no encontrarlo retornar 24)

```
a Dictionary(#a->42 ) .
x size. 1 .
x keys. #(#a) .
x values #(42) .
x at: 'a'. 42 .
x at: 'z' ifAbsent: 24. 24 .
```

1.2 Conversión de colecciones

e. Convertir el Array del punto a en una OrderedCollection y en un Set.
f. Convertir el Set del punto c en Array
g. ¿Qué retorna convertir el Dictionary en Array?

```
|x xList xSet|
x:= #(1,2,3,4).
xList:= x asOrderedCollection .
xSet:= x asSet.
```

```
|x xList xSet y|
x:= #(1,2,3,4).
xList:= x asOrderedCollection .
xSet:= x asSet.
y := x asArray. #(1 #, 2 #, 3 #, 4) .
```

```
|x y|
x:= Dictionary new add: #a->1; add:#b->2; add: #c ->3.
y := x asArray.|
```

DA ERROR

1.5 Usar las opciones. *Do it, print It, Inspect It , Explore It, Debug It*. Comprobar que es lo que hace cada una.

Do It → ejecuta
Inspect It → Permite Inspeccionar cada objeto
Print It → Imprime RESULTADO
Explore It → Permite Inspeccionar el RESULTADO
Debug It → Abre debugger

1.6 Cambiar los elementos de la colección elements para comprobar que las colaboraciones funcionan

→ #(1 2 3 4 6

→ #(1 3) y FUNCA

1.7 Enumerar los problemas que tiene ese algoritmo según lo visto en la carrera.

```
| elements index odds | O(1)
elements := #(1 2 3 4 6). O(n)
odds := OrderedCollection new. O(1)
index := 1. O(1)
[index <= elements size] whileTrue: [ O(n)
        ((elements at: index) odd) ifTrue: [odds add:(elements at: index)]} O(1)
        index := index + 1.
    ].
^odds
```

Complejidad: O(n) Si uso Add de List es O(1) y At de Arr es O(1)

Problemas: Consultar ¨

1.8 Convertir el script de 1.1 sin usar *#whileTrue*, utilizando el mensaje *#do*:, ¿qué ventaja tiene la nueva versión?

```
| elements odds |
elements := #(1 2 3 4 6).
odds := OrderedCollection new.
elements do:[:elem | (elem odd) ifTrue: [odds add: elem]].
^odds
```

Mas declarativo y conciso

1.9 Volver a convertir el algoritmo sin cambiar su comportamiento pero usando el mensaje *#select:* en lugar de #do ¿qué ventaja tiene la nueva versión?

#select: filtra elems de una colección. Equiv al filter en otros leng.

```
miColeccion select: [:elemento | <condición booleana>]
```

```
| elements odds |
elements := #(1 2 3 4 6).
odds := elements select:[:num | num even].
^odds
```

#(1 3)

→ Devuelve el Tipo sobre el que select fue llamado

odds ACA es ARRAY

1.10 Crear una secuencia similar a la de 1.1 pero que obtenga el doble de cada elemento de la colección. Por ejemplo elements = #(1 2 5) debería retornar #(2 4 10)

```
| elements index act|
elements := #(1 2 3 4 6).
index := 1.
[index <= elements size] whileTrue:[
        act := (elements at: index)*2.
        elements at: index put: act.
        index := index +1.
    ].
^elements .
```

1.11 Reescribir el algoritmo utilizando *while* y luego utilizando *do* ¿Donde se acumulan los resultados?

```
| elements index |
elements := #(1 2 3 4 6).
index := 1.
elements do: [:elem | elements at: index put: elem * 2. index := 1 + index].
^elements
```

1.12 Encontrar luego un mensaje mejor en colecciones y dejar el algoritmo más compacto. ¿Qué retorna el nuevo mensaje?

```
| elements |
elements := #(1 2 3 4 6).
elements :=elements collect: [:elem | elem*2].
^elements  #(2 4 6 8 12) .
```

1.13 Crear una nueva secuencia de colaboraciones para encontrar el primer número par, utilizando otro mensaje de colecciones. Como siempre primero con while: luego con do: y luego con un mensaje específico. Ejemplo: dado *#(1 2 5 6 9)* debería retornar 2

```
| index elements res |
elements := #(1 1 3 4 6).
index := 1.
res := nil.
[index <= elements size] whileTrue: [
        (res = nil and: [(elements at: index) even]) ifTrue: [ res := index].
        index:= index + 1.
    ].
^res
```

```
| elements res |
elements := #(1 1 3 4 6).
res := nil.
elements do: [:elem |(res = nil and: elem even) ifTrue: [res := elem].].
^res
```

Falta con msg especifico. Cual seria?

1.14 Utilizar la secuencia de colaboraciones con una colección sin pares. Por ejemplo *#(1 5 9)*. ¿Qué ocurre?

↳ Según mi implementación devuelve nil

Corrección!!!

1.15 Modificar la secuencia para generar un error en caso de no contener pares utilizando *self error: 'No hay pares'*. Evaluarlo en una colección con pares (retorna el primero) y sin pares (se genera un error con el mensaje específico)

```
| elements res |
elements := #(1 1 3 3 3).
res := nil.
elements do: [:elem |(res = nil and: [elem even]) ifTrue: [res := elem]].
(res = nil)
    ifTrue: [Error signal: 'No se encontro numero par']
    ifFalse: [^res].
```

Pedir corrección!!!

1.16 Sumar los números de una colección utilizando primero *while*, luego *do* y luego un mensaje de sumar colecciones. Hay un mensaje específico para la suma y otro para acumular elementos llamado inject:into: Solucionarlo utilizando ambos.

Con while y do es trivial por acumulador

Con sum

```
| elements res |
elements := #(1 2 3).
res := nil.
res := elements sum.
^res. 6 .
```

```
| elements res |
elements := #(1 2 3).
res := 0.
res := elements inject: 0 into: [:acu :elem | elem + acu].
^res.
```

1.17 ¿Cuántos colaboradores recibe *inject:into:* ? Pruebe debuggearlo con el menú o poniendo *self halt.* antes de las colaboraciones (esto detendrá la ejecución y abrirá el debugger)

Serían estos los COLAB?

```
inject: thisValue into: binaryBlock
    "Accumulate a running value associated with evaluating the argument,
    binaryBlock, with the current value and the receiver as block arguments.
    The initial value is the value of the argument, thisValue.
        For instance, to sum a collection, use:
            collection inject: 0 into: [:subTotal :next | subTotal + next]."

    | nextValue |
    nextValue := thisValue.
    self do: [:each | nextValue := binaryBlock value: nextValue value: each].
    ^nextValue
```

1.18 Crear una nueva secuencia para extraer únicamente las vocales en el orden que aparecen en un *string*.

```
| str res vocales |
str := 'hola como estas!'.
vocales := 'aeiou'.
res := str select: [:char | (vocales includes: char)].
^res. 'oaooea' .
```

1.19 ¿Qué observa con respecto a los *strings* y otras colecciones?

↳ Aparentan las colecciones manejar metodos en común
Esto tiene sentido
Aunque se implementan distinto adentro aun así comparten interfaz! (conclusion mia, puede estar mal)

1.20 ¿Conocía estos mensajes de colecciones de materias anteriores? ¿Cómo se llamaban?

do es como un FOR EACH
select como un Filter

2. Bloques (Closures)

a. ¿Cuál es la definición de Blocks que se encuentra en el libro *Smalltalk-80 The Language and its Implementation*.?
b. ¿Qué valor retorna un Block cuando se evalúa (con `value`)?

a) Claramente buscado en CHAT GPT

Definición de un Block en el Blue Book

Un **Block** en Smalltalk es una **expresión anónima y ejecutable**, que se define entre `[` y `]` y que puede tomar parámetros y devolver un resultado. En términos más formales:

> ***"A block is a sequence of expressions enclosed in brackets (`[ ]`). A block may be thought of as an anonymous function or a lambda expression. It may take arguments, execute a sequence of expressions, and return a value."***

b) Retorna el valor de su ultima expresión

c. Evaluar en el Workspace lo siguiente:

```
| x |
x := [ y := 1. z := 2. ].
x value.
```

da 2 pq

i. ¿Qué sucede si queremos acceder a una variable definida en el bloque desde fuera del bloque?

```
| x |
x := [ y := 1. z := 2. ].
x value.
y.
```

¿Qué sucede al acceder a una variable definida fuera del bloque desde dentro del bloque?

```
| x y |
x := [ y := 1. z := 2. ].
x value.
y.
```

i) No aparenta dar problema en ninguno de los dos casos

ii. Dé un ejemplo de un bloque con dos parámetros y su evaluación.

```
|x y|
x:= [:v1 :v2 | y:=v1. z:=v2.].
x value: 1 value: 3.
y + z. 4 .
```

3. Símbolos

a. ¿Cuál es la definición de Symbol que se encuentra en el libro *Smalltalk-80 The Language and its Implementation*.?

ay

Un **símbolo** en Smalltalk es un identificador único, inmutable y globalmente compartido. Formalmente, el libro define un símbolo de la siguiente manera:

> ***"A symbol is a unique, immutable string that is used as an identifier. Unlike strings, which may have multiple identical copies in memory, symbols are stored only once, making comparisons and lookups efficient."***

b. Evalúe en el Workspace:

```
| x y |
x := #pepe.
y := #pepe.
x = y.
```

ma True

Me imagino que a dif de hacer 'pepe', aca estamos ocupando solo un lugar en memoria y x e y referencial al mismo espacio/valor

c. ¿Cuál es el resultado de concatenar símbolos?

#Hello , #World, #!

```
#Hi , #Hello, #!. 'HiHello!' .
```

ma Parece ser que devuelve una string?

4. Medidas

4.1 Sobre la importancia de las medidas en nuestra profesión y sobre las responsabilidades de los desarrolladores de software en la industria:

- http://www-users.math.umn.edu/~arnold/disasters/ariane.html
- https://motherboard.vice.com/en_us/article/qkvzb5/the-time-nasa-lost-a-mars-orbiter-because-of-a-metric-system-mixup
- https://en.wikipedia.org/wiki/Gimli_Glider

Revienta un cohete
y casi choca un avión
por mala conversión

Tranqui 

4.2 Evalúe estas colaboraciones

```
10 * peso + 10 * dollar
```

→ MessageNotUnderstood
→ Multiple compound measure Err.

¿Qué resultado esperaba? ¿Cuál Obtuvo?

Lo mismo

4.3 Evalúe estas colaboraciones anotando previamente que resultado cree va a obtener

```
10 * peso +( 10 * dollar)
```

? En peso? → `10 * dollars+10 * pesos .` ...

```
10 * peso +( 10 * dollar) - (2 * dollar)
```

→ `10 * pesos+8 * dollars .` Parece que opera lo que puede

```
10 * peso +( 10 * dollar) - (2 * dollar) - (8 * dollar)
```

→ `10 * pesos .`

4.4 ¿Qué es peso? inspecciónelo:

peso inspect

4.5 ¿qué es 10 * peso? evalúe:

```
(10 * peso) amount
```
`10 .`

```
(10 * peso) unit
```
`peso .`

4.6 Y ¿qué son los números en este contexto?¿Qué unidad llevan?

1 amount → 1 Los numeros son valor numerico y ya.
1 unit → nil No tienen unidad

4.7 ¿Cuánto es (10 * peso) + 1 y 1 + (10 * peso) ? → `1+(10 * peso). 10 * pesos+1 .`

`(10 * peso) + 1. 10 * pesos+1 .`

(10 * peso) * 5 50 pesos
(10 * peso) * 5 50 pesos
(10 * peso) * (5 * peso) 50 pesos*pesos

4.8 Cree el peso nuevamente:

peso := BaseUnit nameForOne: 'peso' nameForMany: 'pesos' sign: $$

¿Qué representa $$? (Evaluar la expresión ayuda a entenderlo)

```
peso := BaseUnit nameForOne: 'peso' nameForMany: 'pesos' sign: $$.
100 * peso. 100 * pesos .
```

CONSULTAR!!! No llego a entender como devuelve con sign y cual es su fin

```
|metro centimetro pulgada diezMetros sesentaPulgadas|
metro := BaseUnit nameForOne: 'metro' nameForMany: 'metros'.
centimetro := BaseUnit nameForOne: 'centimetro' nameForMany: 'centimetros'.
pulgada := BaseUnit nameForOne: 'pulgada' nameForMany: 'pulgadas'.
diezMetros := 10 * metro.
sesentaPulgadas := 60 * pulgada.
diezMetros + sesentaPulgadas. 60 * pulgadas+10 * metros .
```

No recuerdo de cual las equiv ¿como las hacia?

Consultar!!!