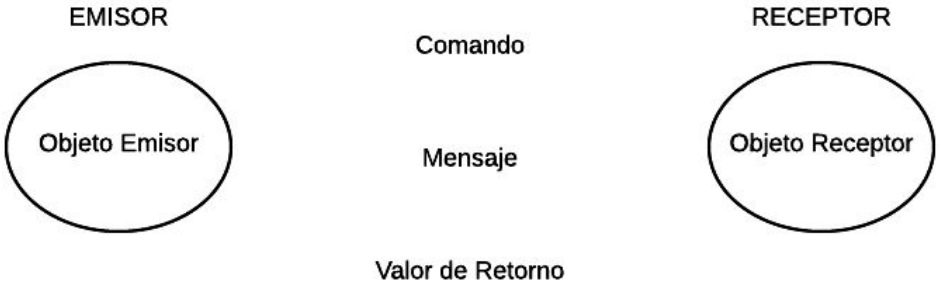
# ¿Qué es la Programación Orientada a Objetos?

La programación Orientada a Objetos (POO) es diferente de los lenguajes de programación procedurales (C. Cobol, Pascal, etc.). Todo en POO está conceptualizado como “objetos”. La POO, definida en su más puro sentido, es implementada por el envío de mensajes a los objetos. para entender este concepto, primero debemos conocer qué es un objeto.

# ¿Qué es un Objeto?

Un objeto puede ser considerado una “cosa” que puede realizar un conjunto de actividades. El conjunto de actividades que un objeto puede realizar define el comportamiento del objeto. por ejemplo, un objeto **`EstadoAlumno`** puede informarle su promedio, año de escuela, etc.; o puede agregar una lista de las materias cursadas. un objeto **`Alumno`** puede informarle su nombre o su dirección.

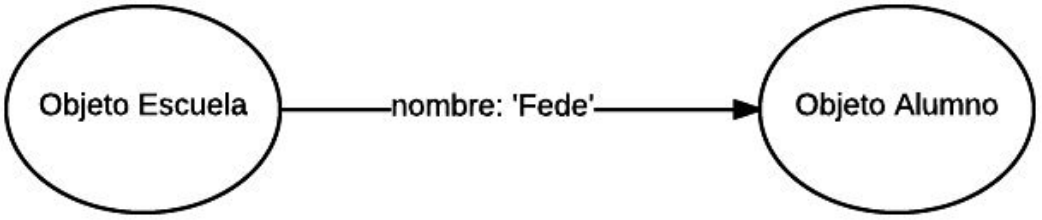
La interface del objeto consiste en un conjunto de comandos, cada comando desarrolla una acción específica. un objeto puede pedirle a otro objeto que realice una acción, enviándole un mensaje. El objeto que envía el mensaje es designado emisor, y el objeto que recibe el mensaje es designado receptor.




El control le es dado al objeto receptor hasta que complete la acción; luego el control regresa al objeto emisor.
Por ejemplo, un objeto **`Escuela`** le pide al objeto **`Alumno`** su nombre enviándole un mensaje preguntándole por su nombre. El objeto receptor **`Alumno`** devuelve su nombre al objeto emisor.

Un mensaje también puede contener información que los objetos emisores necesitan pasar al objeto receptor, a esta información se la llama argumento de un mensaje.un objeto receptor siempre devuelve un valor al objeto emisor. Este valor devuelto puede ser, o no, útil al objeto emisor.

Por ejemplo, el objeto **`Escuela`** ahora desea cambiar el nombre del alumno. Esto lo hace enviando al objeto **`Alumno`** un mensaje para cambiar su nombre a uno nuevo. El nuevo nombre es pasado como argumento en el mensaje. En este caso, al objeto **`Escuela`** no le interesa el valor que devuelva el mensaje.



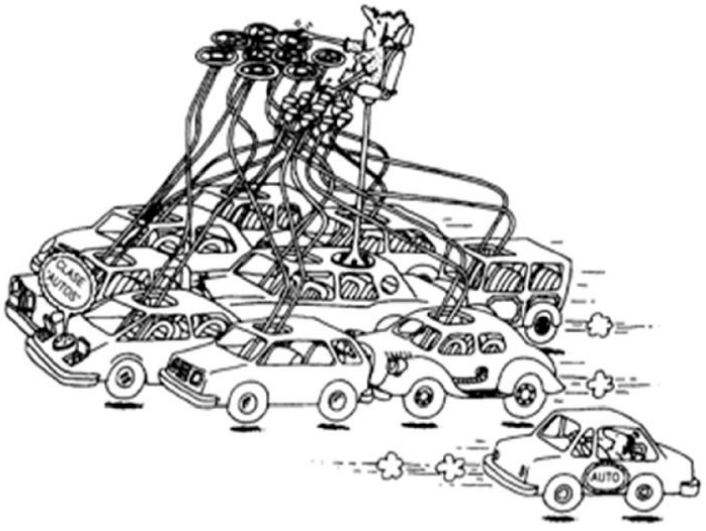

## Clases y Objetos

Los conceptos de clase y objeto están estrechamente relacionados; sin embargo, existen diferencias entre ambos: mientras que un objeto es una entidad concreta que existe en el espacio y en el tiempo, una clase representa sólo una abstracción, la "esencia" de un objeto.

Se puede definir una clase como un grupo, conjunto o tipo marcado por atributos comunes, una división,distinción o clasificación de grupos basada en la calidad, grado de competencia o condición. En el contexto del paradigma Orientado a Objetos se define una **clase** de la siguiente forma:

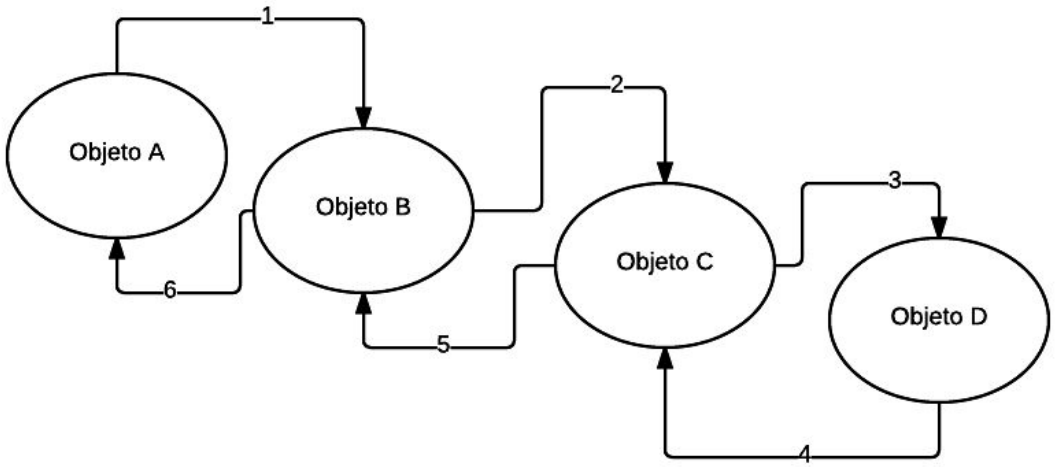
*"Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común"*

Un objeto es una instancia de una clase. Los objetos que comparten estructura y comportamiento similares pueden agruparse en una clase. En el paradigma orientado a objetos, los objetos pertenecen siempre a una clase, de la que toman su estructura y comportamiento. Por mucho tiempo se ha referenciado a las clases llamándoles objetos, no obstante, son conceptos diferentes.



## Operación secuencial

Es muy común que un mensaje provoque el envío de otros mensajes, ya sea el mismo u otros objetos, para completar su tarea. A esto se lo llama *operación secuencial*. El control no le será devuelto al objeto emisor original hasta que todos los otros mensajes hayan sido completados. Por ejemplo, el siguiente diagrama, el objeto a envía un mensaje al Objeto B. Para que el Objeto B procese este mensaje, envía otro mensaje al Objeto C. Así mismo el Objeto C envía otro mensaje al Objeto D. El objeto D devuelve el control al Objeto C, asimismo el Objeto C devuelve el control al Objeto B que, a su vez, lo devuelve al Objeto A. El control no es devuelto al objeto A hasta que todos los otros mensajes se han completado.



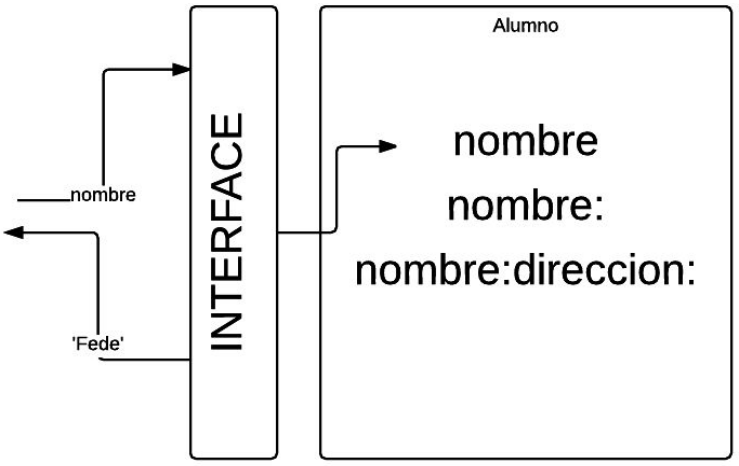

## Método

¿Cómo los objetos receptores interpretan los mensajes de los objetos emisores?¿Cómo se procesan los mensajes?

Cada mensaje posee un código asociado a él. Cuando un objeto recibe un mensaje, su código es ejecutado. en otras palabras, estos mensajes determina el comportamiento de un objeto y el código determina cómo, el objeto, lleva a cabo cada mensaje. El código que está asociado a cada mensaje se llama **método**. El nombre del mensaje; también suele llamarse el *nombre del método* debido a su asociación con el método.

Cuando un objeto recibe un mensaje, determina qué método es el solicitado y pasa el control al *método*. un objeto tiene tantos métodos como acciones se desea que tenga.

Observe el siguiente diagrama: `nombre`, `nombre:`, `dirección`, `nombre:direccion:` son nombre de métodos para el objeto **`Alumno`**. Cuando el objeto **`Alumno`** recibe el mensaje `nombre`, el mensaje `nombre` pasa el control al método `nombre` definido en **`Alumno`**.

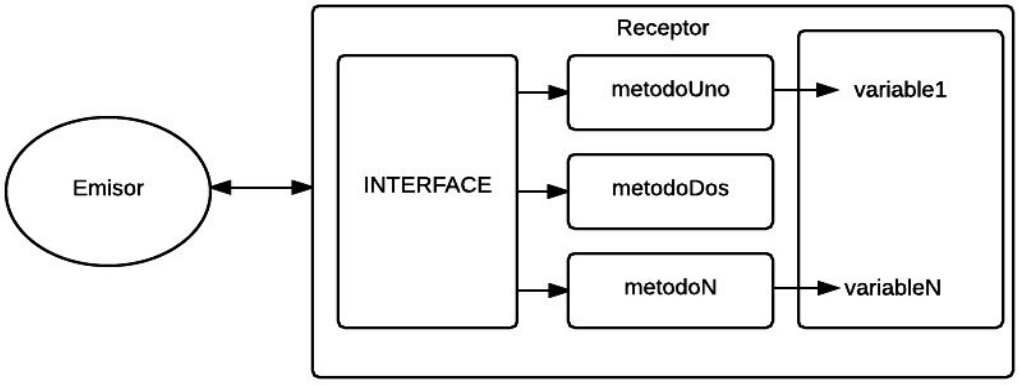



Los métodos que operan sobre objetos específicos son los ***métodos de instancias***, y los mensajes que invocan los métodos de instancia de llaman *mensajes de instancia*. Los métodos que operan sobre clases de objetos específicas son los ***métodos de clase.***
Los métodos son similares a las subrutinas, procedimientos o funciones que se encuentran en los lenguajes procedurales. Por ejemplo, un nombre de método equivale al nombre de una subrutina, y el código del método equivale al código de la subrutina. El envío de mensajes a objetos es similar al llamado de subrutinas

## Datos de un Objeto

Cada objeto necesita guardar la información sobre cómo realizar su comportamiento definido. algunos objetos también contienen variables que soportan su comportamiento. Estas variables son llamadas *variables de instancia* Los métodos de instancia de los objetos no pueden referirse a los datos de otros objetos. un objeto sólo puede acceder a los datos de otro objeto enviándole mensajes. A esto se lo llama **encapsulación** y asegura que el proceso para obtener los datos de un objeto sea seguro.

Observe el siguiente diagrama, las variables de instancia *variableUno* hasta *viarableX* sólo pueden ser accesadas por el emisor a través de los métodos de instancia *metodoUno* hasta *medotoN*. EL emisor no puede referirse a las variables, directamente, por sí mismo.



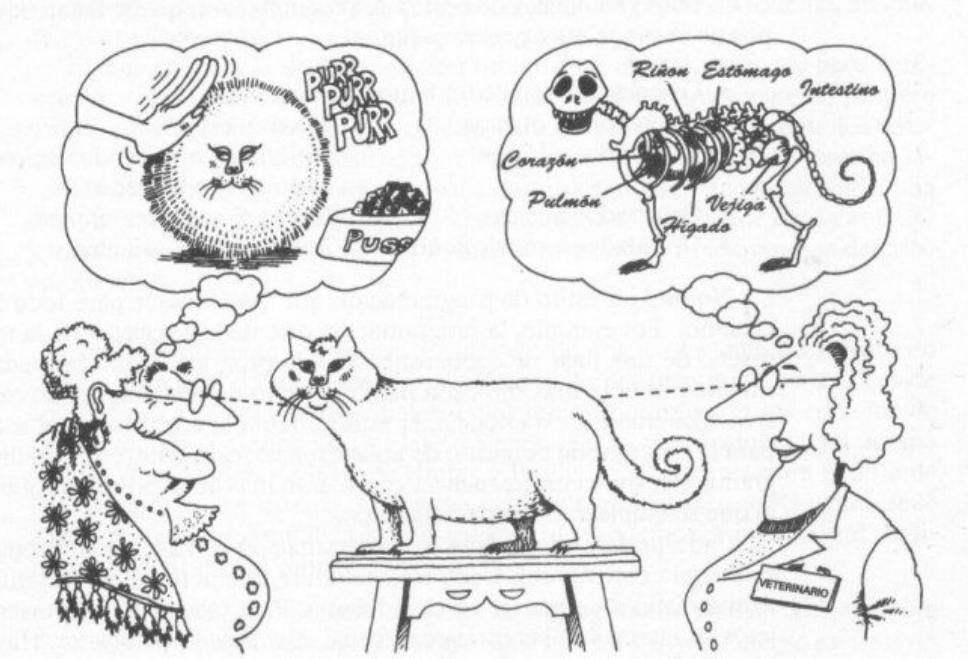

A diferencia de la programación procedural, donde las áreas de datos de uso común son utilizadas para compartir información, la programación orientada a objetos no aprueba el acceso directo a los datos de uso común (exceptuando el uso de las *variables globales*) por otros programas. Sólo el objeto que "posee" los datos puede cambiar su contenido. otros objetos pueden ver o cambiar estos datos enviando mensajes al "dueño".
Los nombres de variables de instancia pueden ser idénticos a los nombres de los métodos que están asociados a ellos. Por ejemplo, el objeto **`Alumno`** tiene los métodos *`nombre`*, *`dirección`* y *`carrera`* así como también las variables de instancia *`nombre`*, *`direccion`* y *`carrera`*. Smalltalk distingue entre los identificadores de variables y los identificadores de métodos a través de la posición que ocupan en la expresión.

# Propiedades específicas de los objetos

Un lenguaje de computación está orientado a objetos si soporta las cuatro propiedades específicas de los objetos, llamadas: **abstracción, polimorfismo, herencia y encapsulamiento**.

## Abstracción de datos

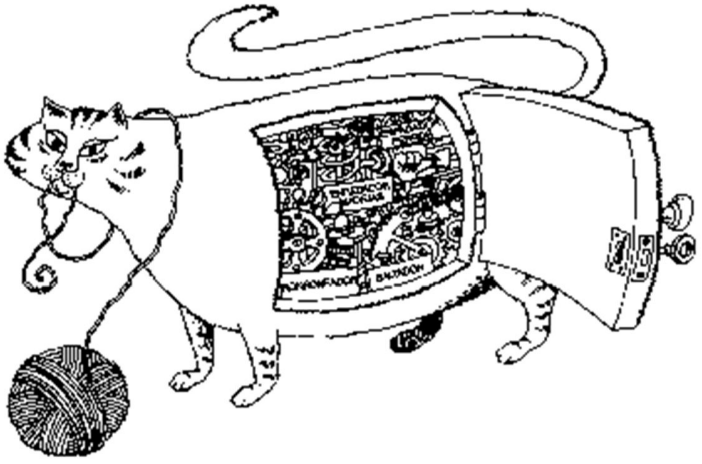
La abstracción de datos permite diferenciar entre el comportamiento de un objeto (la acción que es capaz de realizar) y cómo lleva a cabo este comportamiento. Esta abstracción de datos se implementa a través de una interface de objeto.
A través de esta interface un objeto emisor se comunica con otro objeto receptor, pero el objeto emisor desconoce la forma en que se lleva a cabo la acción solicitada (mensaje).
Siguiendo la imagen presentada, dependiendo del observador, nos va a permitir concentrar en los aspectos más relevantes para el estudio.




La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

## Encapsulación

En el POO, los objetos interactúan entre ellos a través de los mensajes. lo único que un objeto conoce de otro es su interface. los datos y la lógica de un objeto está oculta a los otros objetos. O sea, la interface, encapsula los datos y código de un objeto.



Esto permite al programador separar la implementación de un objeto, de su comportamiento. Esta separación crea una “caja negra” en donde el usuario está añejado de los cambios de la implementación. mientras la interface permanezca igual, cualquier cambio interno a la implementación es transparente al usuario. Por ejemplo, si el mensaje *`jugar`* es enviado al objeto **`Gato`**, al usuario no le importará cómo el programador

implementó el código que maneja este mensaje. El programador puede cambiar la implementación en cualquier momento, pero el mensaje nombre aún debería trabajar ya que la interface es la misma.

## Polimorfismo

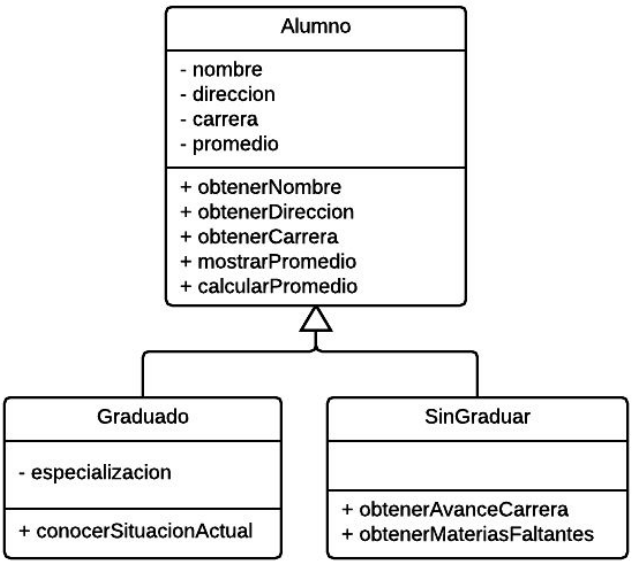
Otro beneficio que surge al separar la implementación del comportamiento es el polimorfismo. El polimorfismo permite a dos o más objetos responder al mismo mensaje. Un método llamado *nombre* también podría ser implementado en un objeto de la clase **Curso**. Aunque la implementación de este mensaje nombre devolvería el número de curso y su nombre, su protocolo es el mismo que el mensaje *nombre* del objeto **Alumno**. El polimorfismo permite a un objeto emisor comunicarse con diferentes objetos de una manera consistente sin tener que preocuparse sobre las diferentes implementaciones de un mensaje.

Una analogía del polimorfismo con la vida diaria es cómo los alumnos responden al timbre de la escuela. Todos los alumnos conocen el significado del timbre. Cuando el timbre (mensaje) suena, sin embargo, tiene su propio significado para cada alumno (Objeto). Algunos alumnos irán a casa, otros a biblioteca y otros irán a clases. cada alumno responde al timbre, pero su respuesta puede ser diferente.

El polimorfismo permite al objeto emisor comunicarse con los objetos receptores sin tener que comprender qué tipo de objeto es, siempre y cuando el objeto receptor soporte los mensajes.

## Herencia

Otro concepto importante de la POO es la herencia. La herencia permite a una clase tener el mismo comportamiento que otra y extender o limitar ese comportamiento para proveer una acción especial a una necesidad específica.




Suponga la siguiente aplicación. La clase **`Graduado`** y la clase **`SinGraduar`** tienen comportamiento similar, como por ejemplo, el manejo de nombre, direccion, carrera, promedio, etc. En vez de colocar este comportamiento en ambas clases, el comportamiento es colocado en una nueva clase llamada **`Alumno`**. Ambas clases **`Graduado`** y **`SinGraduar`** se vuelven subclases de clase **`Alumno`**, y ambas heredan el comportamiento de **`Alumno`**.

Ambas clases **`Graduado`** y **`SinGraduar`** pueden agregar comportamiento adicional que sea único a ellas. Por ejemplo, **`Graduado`** puede ser tanto para un doctorado o una carrera de postgrado. por otro lado, la clase **`SinGraduar`** puede conocer cómo un cierto alumno va en la carrera.

Las clases que heredan de otras clases se llaman subclases o clases hijas. La clase de la que una subclase heredada, es llamada superclase o clase padre. En el ejemplo, **`Alumno`** es una superclase de **`Graduado`** y **`SinGraduar`**. **`Graduado`** y **`SinGraduar`** son subclases de **`Alumno.`**

## Enviando Mensajes a Objetos

Como ha sido mencionado antes, todo el procesamiento de Smalltalk se realiza por medio del envío de mensajes a objetos. Un enfoque inicial de la resolución de problemas, en Smalltalk, es tratar de reutilizar los mensajes y objetos existentes. El programador de Smalltalk, trabaja para desarrollar una secuencia objeto-mensaje que proveerá la solución deseada.
Los objetos son instancias de una clase en particular. Los mensajes a los que un objeto puede responder están definidos en el protocolo de su clase. El cómo estos mensajes se

ejecutan o son implementados está definido en los métodos de clase. Los métodos dan los detalles de implementación para los mensajes, y representan el comportamiento de la clase.

## Dando nombre a Objetos y Mensajes

Por convención, todos los nombres de clases comienzan con una letra en mayúscula, como por ejemplo **`Alumno`**. Los nombres de Mensajes comienzan con una letra en minúscula y pueden tener cualquier combinación de letras y números sin espacios en blanco. Cuando el nombre de un mensaje contiene más de una palabra, las palabras extras comienzan con una letra en mayúscula. Un nombre válido de mensaje podría ser *`unMensaje`* o *`miDirección`*

Un mensaje como "dame tu nombre" podría ser *`dameTuNombre`*. SIn embargo se trata de acortar los nombres de mensajes, pues es muy difícil trabajar con nombres largos; por lo tanto el nombre de mensaje adecuado sería *`nombre`*.

Un mensaje que pasa información o parámetros, tal como "cambia el nombre a este nombre", se maneja en forma similar al anterior, con la excepción que el nombre del mensaje debe finalizar con dos puntos (:). Así que el nombre del mensaje se vuelve *`nombre:`*. Estos tipos de mensajes son llamados mensaje de palabra clave. Los mensajes que no pasan información son mensajes unarios.

Un mensaje puede contener múltiples argumentos. Debe haber una palabra clave por cada argumento en el mensaje. Por ejemplo, es posible indicar el nombre y la dirección de un alumno en un sólo mensaje. Esto requiere un nombre de mensaje con dos palabras clave, por ejemplo "*`nombre:dirección:`*"

Existen tres tipos de mensajes: unarios, binarios y palabra clave.

### Mensajes Unarios

Un mensaje unario es similar a la llamada de una función con parámetro único. Este tipo de mensaje consiste de un nombre de mensaje y un operando. Los objetos se colocan antes del mensaje. Por lo tanto los mensajes unarios siguen la forma: **`ObjetoReceptor`** *`mensaje`*

Los siguientes son ejemplos de mensajes unarios:

`x` *`sin`* "devuelve el resultado de sen(x)"

**`Date`** *`tomorrow`* "devuelve una nueva instancia de la clase Date"

`5` *`factorial`* "devuelve el factorial de 5"

`'hola'` *`size`* " devuelve el tamaño del string 'hola' "

```
3 sqrt “raíz cuadrada de 3”

Float pi “obtener el valor de Pi”

true not “ niega la expresión true obteniendo como resultado false”
```

### Mensaje Binarios

Los mensajes Binarios son utilizados para especificar operaciones aritméticas, lógicas y de comparación. Un mensaje binario puede ser de uno o dos caracteres de longitud y puede contener cualquier combinación de los siguientes caracteres especiales:
`+ / \ * ~ < > @ % | & ? ! ,`
POr lo tanto, los mensajes Binarios siguen la estructura de:
`ObjetoReceptor caracter ObjetoParametro`

Los siguientes son ejemplos de expresiones en Smalltalk, utilizando mensajes binarios:

```
a + b “devuelve la suma de a y b”

a | b “devuelve la comparación lógica a o b”

a>=b “compara si a es mayor o igual que b, y devuelve true (verdadero) o false (false)”

100@100  "crea un objeto Point"
```

El primer ejemplo puede también ser interpretado como “el mensaje + se envía al objeto ‘a’ con el parámetro ‘b’ “

### Mensaje de Palabra Clave o Keyword

Un mensaje de Palabra CLave es equivalente a una llamada de un procedimiento con dos o más parámetros. Observe el siguiente ejemplo, el nombre del objeto, al cual el mensaje es enviado, se escribe primero, luego el nombre del mensaje (o nombre del método) y luego el parámetro que se pasa.
`UnObjeto unMensaje:parametro`
Los dos puntos (:) son una parte esencial del nombre del mensaje.
Cuando hay más de un parámetro, el nombre del mensaje debe aparecer para cada parámetro.
Por ejemplo: `unObjeto unNombre1:parametro1 unNombre2: parametro2`
Aquí:

- **unObjeto** es el objeto receptor
- *unNombre1:* es la primera parte del mensaje.
- parametro1 es pasado a *unNombre1:*
- *unNombre2:* es la segunda parte del nombre del mensaje

- parametro2 es pasado a *unNombre2:*

Ejemplos de mensajes de palabra clave:

```
Array new:20 "se envía el mensaje new: al objeto Array con el parámetro 20"

UnDia month: esteMes tear:EsteAnio "Establece los datos privados del objeto UnDia, month= esteMes y year= esteAnio"

Alumno nombre:'Juan' direccion:'French 414' "Establece el valor de las variables de instancia del objeto Alumno de acuerdo a los parámetros"

1 to: 10  "(1 to: 10) crea un intervalo"

12 between: 8 and: 15  "pregunta si el objeto 12 se encuentra entre 8 y 15"
```

## Creando nuevas Instancias

Para crear nuevas instancias de una clase, el mensaje new es enviado a la clase. Por ejemplo: **Alumno** *new*
Cada clase automáticamente contiene el mensaje new. El mensaje *new* devuelve un puntero a la nueva instancia.
En el siguiente ejemplo, la variable global `MiAlumno` apunta a un nuevo objeto Alumno:

```
MiAlumno:= Alumno new.
```

## Formato de un Mensaje

En general, una expresión en Smalltalk consiste en el nombre del objeto que recibe el mensaje, seguido por el nombre del mensaje. Por ejemplo: **UnObjeto** *unMensaje*
Para aquellos mensajes que necesitan pasar argumentos al objeto receptor, como los mensajes palabra clave, al nombre del objeto receptor le sigue el nombre del mensaje y su argumento. Los argumentos se separan por medio de un espacio en blanco.

```
UnObjeto unNumero: 1 unNombre: 'Pedro'
```

### Orden de Ejecución de los Mensajes

Las reglas que usa Smalltalk en la decisión del orden de ejecución de los mensajes, puede resumirse en los siguientes pasos:

1. Smalltalk ejecuta mensajes de izquierda a derecha
2. El resultado de un mensaje reemplaza el mensaje en la sentencia
3. Smalltalk ejecuta primero todas las expresiones que aparecen entre paréntesis, comenzando por la izquierda, y por aquella que está anidada.

4. Dentro de una expresión, los mensajes unarios se ejecutan primero, luego los mensajes binarios, y finalmente los mensajes de palabra clave; siempre de izquierda a derecha.
5. Smalltalk ejecuta todos los mensajes binarios de izquierda a derecha, independientemente de las operaciones que realicen. Esto significa que no hay una orden especial para ejecutar operaciones aritméticas.
6. Una expresión puede incluir el nombre de una variable Smalltalk reemplaza el nombre de la variable con el objeto al que ella apunta.

### Ejemplo 1

```
2 squared + 3 squared raisedTo: 2
```

En este caso, se evalúan en este orden: `((2 squared ) + (3 squared )) raisedTo: 2`

1. Los mensajes "*squared*" (mensajes unarios), `2 squared` devolviendo el objeto 4, `3 squared` devolviendo el objeto 9.
2. El mensaje "+" (mensaje binario), que se envía al objeto 4 con el objeto 9 como parametro. El mensaje "+" devuelve el objeto 13.
3. El mensaje *raisedTo:* (mensaje de palabra clave), que se envía al objeto 13 con el objeto 2 como parametro.

### Ejemplo 2

```
3 + 4 * 2
```

En este caso tenemos solamente mensajes binarios. Esto no da el resultado 11, es decir `3 + 8`, sino que da 14: Smalltalk evalúa los mensajes binarios, como ya se dijo, de izquierda a derecha: primero suma `(3 + 4)`, luego multiplica `(7 * 2)`.
Esto es porque "+" y "*" son solo mensajes comunes y corrientes que se envían a objetos, que en este caso representan números, smalltalk no conoce las reglas de la precedencia de las matemáticas. Esto debería haberse escrito "`(3 + 4) * 2`" o "`3 + (4 * 2)`" según lo que se quiera hacer (generalmente se utilizan paréntesis por claridad incluso si lo que se quiso escribir fue "`(3 + 4) * 2`".

### Ejemplo 3

```
2 raisedTo: 2 raisedTo: 2
```

Podríamos pensar que esto devuelve 2 elevado al cuadrado dos veces seguidas, pero no: smalltalk entiende que estamos enviando un único mensaje "`raisedTo:raisedTo:`" con dos palabras claves (este mensaje no existe).
Entonces, debería haberse escrito así: `(2 raisedTo: 2) raisedTo: 2`

### Ejemplo 4

```
valorNuevo := valorViejo raisedTo: 2 + 3.
```

En este caso, valorNuevo va a hacer referencia al resultado de elevar a la quinta al valorViejo, porque la asignación tiene la precedencia más baja.

# Devolviendo un Valor

En Smalltalk, un mensaje siempre devolverá un valor. La devolución por defecto, es el receptor del objeto. Un método puede anular su valor de retorno por defecto, colocando un símbolo de intercalación **^** delante de la instrucción. Cuando se encuentra un **^**, el método finaliza la ejecución y devuelve el valor de la instrucción que le sigue el símbolo de retorno **^**.
Por ejemplo: *^instrucción*
Se retorna el resultado de l instrucción, donde instrucción puede ser cualquier instrucción válida.
Una expresión de retorno aparece normalmente adelante de la última instrucción de un método, o adelante de una instrucción que se ejecuta como parte de una instrucción del control. El siguiente ejemplo es una práctica importante, porque pone de manifiesto que la instrucción de retorno finaliza la ejecución del método en la que se encuentra.

```
y:= y +y
^y  "devuelve el valor de y"
```

La instrucción también puede ser escrita como:

```
^y:=y+7 "devuelve el valor de y"
```

Para devolver el valor de y+7 sin modificar el valor de y:

```
^y+7 "devuelve el valor de y+7, pero no modifica el valor de y"
```

Devolviendo un valor desde una instrucción condicional

```
a<b ifTrue:[^a]
       ifFalse:[^b]
```

Este ejemplo contiene más de una instrucción de retorno. Este código devuelve el valor de a si es mayor que b, caso contrario devuelve el valor de b.
En la instrucción ^a, la expresión es la variable a y el valor retornado de la expresión es el valor de a. El valor de una instrucción siempre es equivalente a la última expresión ejecutada en la instrucción.

# Bloques

Los bloques son corchetes que contienen ninguna o muchas expresiones, y un código que realiza iteraciones o ejecuciones condicionales. Por ejemplo, un bloque de código es utilizado para especificar qué ejecutar como resultado de una condición verdadera o falsa:

```
ifTrue:[x:=2]
ifFalse:[x:=5]
```

Un bloque puede pensarse como un mini-método dentro de un método. Las siguientes reglas se aplican a los bloques:

- Un bloque puede contener cualquier número de instrucciones válidas ejecutables, o cualquier número de comentarios.
- Cada instrucción debe terminar con un punto, excepto cuando se definen las variables temporales, y en la última instrucción del bloque, donde el punto es opcional.

- Un bloque tiene acceso a las mismas variables que el método al que pertenece.

Como un bloque es parte de un método no posee definición de interface de método. El bloque en la siguiente instrucción: `ifTrue:[x:=2]` es llamado bloque de cero-argumento; no puede aceptar ningún argumento. Sin embargo es posible definir un bloque que pueda tomar argumentos, como el siguiente: `[:variable1| código]` `[:variable1 :variable2| código]`
donde **`variable1`** y **`variable2`** son variables temporales, son válidas dentro del alcance del bloque. El nombre de la variable es precedido por dos puntos':'. `[:unNumero | x* unNumero]`
La variable `unNumero` está definida dentro del bloque. Esta instrucción multiplica una variable llamada x por el argumento pasado en el bloque. La variable x se debe estar dentro del alcance del método en donde se encuentra el bloque.

## Elementos sintácticos

Las expresiones se componen de las siguientes piezas:

I. seis palabras reservas o pseudo-variables: self, super, nil, true, false y thisContext;
II. expresiones constantes para objetos literales incluyendo números, caracteres, cadenas de caracteres, símbolos y arrays;
III. declaraciones de variables;
IV. asignaciones;
V. bloques de cierre;
VI. mensajes.

Podemos ver ejemplos de los distintos elementos sintácticos en la Tabla:

| **Sintaxis** | **Lo que representa** |
|---|---|
| puntoInicial | un nombre de variable |
| **Transcript** | un nombre de variable global |
| self | una pseudo-variable |
| 1 | entero decimal |
| -1234 | entero negativo |
| -1/7 | fraccion negativa |
| 2r101 | entero binario |
| 1.5 | número en coma flotante |
| -2.34e10 | número de punto flotante negativo |

| | |
|---|---|
| 2.4e7 | notación exponencial |
| $a | el carácter 'a' |
| 'Hola' | la cadena de caracteres "Hola" |
| #Hola | el símbolo #Hola |
| #(1 2 3) | un array literal |
| {1. 2. 1+2} | un array dinámico |
| "un comentario" | un comentario |
| \| x y \| | declaración de las variables x e y |
| x := 1 | asignar 1 a x |
| [ x + y ] | un bloque que se evalúa como x+y |
| **Transcript** *show:* 'hola'. **Transcript** *cr* | separator de expresiones (.) |
| **Transcript** *show:* 'hola'; *cr* | mensaje en cascada (;) |

## Operaciones Aritméticas

Una instrucción que realiza una operación aritmética en Smalltalk tiene la siguiente forma: `numero operacion numero` donde una operación puede ser una de las siguientes:

| Operador | Operación |
|---|---|
| + | suma |
| - | resta |
| * | multiplicación |
| / | división |
| // | división entera (cociente) |
| \\ | resto de una división |

## Concepto de Nil

En Smalltalk, nil es un objeto que significa “nada”. Inicialmente todas la variables apuntan a nil. Cualquier variable puede ser apuntada a nil durante la ejecución, con una instrucción como la siguiente: `unaVariable:=nil.`

Nil también puede usarse como un valor de retorno para indicar que una operación no fue exitosa.

## Comparaciones Lógicas

Una instrucción de comparación tiene el siguiente formato: `valor` *`comparación`* `valor` donde valor puede ser cualquier expresión que resulte en un valor que pueda ser comparado, tal como números, cadenas, caracteres y símbolos; y comparación puede ser cualquier operación válida de comparación.
Algunos ejemplos son:

| Comparación | Significado |
|---|---|
| > | mayor que |
| < | menor que |
| = | igual en valor |
| ~= | desigual en valor |
| >= | mayor o igual que |
| <= | menor o igual que |
| == | el mismo objeto que |

Las comparaciones lógicas devuelven un valor que puede ser true (verdadero ) o false (falso), que son instancias de las clases True y False, respectivamente.

## Expresiones Booleanas “y” “o” lógicos

En Smalltalk, las expresiones booleanas pueden ser combinadas en un resultado, usando la operación o o la operación y. Estas dos funciones pueden ser utilizadas como mensajes binarios o mensajes palabras claves.

## Mensajes Binarios

El mensaje binario para el y lógico es &, y para el o lógico es |
Por ejemplo:

```
(a>0)&(b<0) "devuelve true si a es positivo y b es negativo. Caso contrario devuelve false"

(a>0) | (b<0) "devuelve true si a es positivo y/o b es negativo. Caso contrario devuelve false"
```

Una sentencia puede contener un número ilimitado de mensajes binarios. por ejemplo:

```
|x y z|
x:=3.
y:=5.
z:=7.
(x>0)&(y<0)|(x>y)&(y=z)
```

## Mensajes de Palabra Clave

El mensaje de palabra clave para y lógico es *and:*, y para o lógico es *or:*. El formato para estos mensajes es:

```
booleano and: [código]
booleano or:[código]
```

El `booleano` es cualquier expresión cuyo valor resulte `true` o `false`. El bloque de código encerrado entre corchetes debe devolver un valor `true` o `false`. los métodos *and:* y *or:*. combinan los dos valores booleanos y devuelve el resultado correspondiente.

Existe una diferencia entre los mensajes binarios & y |, y los mensajes de palabra clave *and:* y *or:*, respectivamente. Lo mensajes palabra clave son considerados caminos cortos porque utilizan evaluación diferida. el código en el bloque no es evaluado hasta que el valor del receptor booleano no es determinado como true o false.

En el caso del mensaje *and:*, si el receptor evalúa false, entonces el código del bloque no se ejecuta, ya que en una operación and con un false siempre es false.

En el caso del mensaje *or:*, si el receptor evalúa true, entonces el código del bloque no se ejecuta, ya que en una operación or con true siempre es true.

El siguiente es un ejemplo utilizando mensajes de palabra clave:

```
|x y z|
x:=3.
y:=5.
z:=7.
```

```
(((x>0)and:[y<0]) or:[x>y]) and:[y=z]
```

## "O" Exclusivo

Un booleano también soporta la función de o exclusivo proveyendo el mensaje de palabra clave xor:. Este es idéntico en formato a los mensajes de palabra clave and: y or:, excepto que el argumento debe ser otro booleano, y no un bloque de código. Por ejemplo:

```
|a b|
a:=1.
b:=2
(a>1) xor:(b<0)
```

Este ejemplo retorna el valor true.

## Not

El mensaje unario not provee la función not. Este mensaje invierte el valor booleano (true se vuelve false o false se vuelve true). El formato es: `booleano not`
Ejemplo:

```
(5>1)not "El valor de retorno es false"
```

Es importante asegurarse que un valor booleano inmediatamente le precede la operación *not*. cuando esté inseguro encierra la expresión entre paréntesis. esto asegura que la lógica ocurra en el orden correcto. por ejemplo, la expresión: `5>1 not` tiene dos mensajes: el mensaje binario `>` y el mensaje unario *not*. Smalltalk ejecuta los mensajes unarios primero, lo que causa que el número 1 reciba el mensaje *not*. Claramente esto es la correcta ejecución de la secuencia, y provocará un error en tiempo de ejecución.

## Lógica Condicional

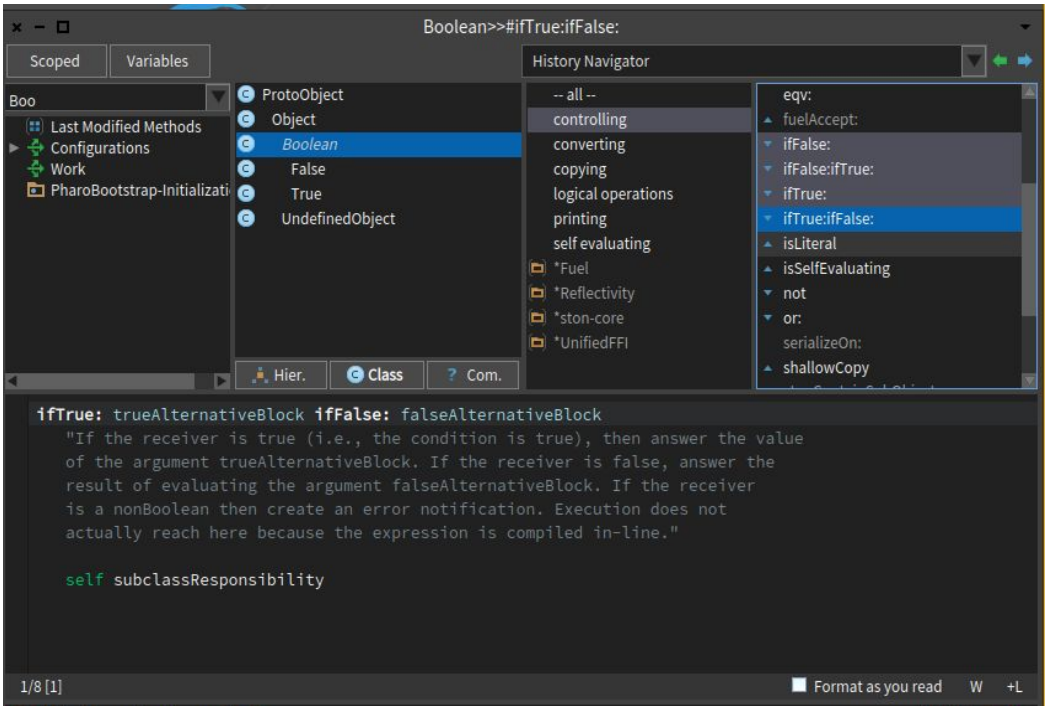
La lógica condicional permite la ejecución del código dependiendo de un valor booleano. existen varios mensajes de palabra clave que proveen esta función, por ejemplo: `booleano ifTrue:[código] ifFalse:[código]` donde `booleano` es cualquier expresión que resulte `true` o `false`. La expresión `[código]` puede ser cualquier bloque cero-argumento.
El mensaje de palabra clave *ifTrue:ifFalse* ejecuta un bloque diferente dependiendo del valor del booleano. Por ejemplo:

```
|x y nuevoValor|
x:=1.
y:=2.
(x>y) ifTrue:[nuevoValor:=x]
      ifFalse:[nuevoValor:=y].
^nuevoValor
```

Este ejemplo determina valor de la variable `nuevoValor` al valor del mayor entre `x` o `y`, en caso `y`, y devuelve el valor en `nuevoValor`.

Si revisamos en "System Browser"en Pharo, dentro de la implementación de la Clase Boolean, se podrán observar otros métodos de lógica condicional además del propuesto, como ser:

- `booleano ifTrue:[Código]`
- `booleano ifFalse:[Código]`
- `booleano ifFalse:[Código] ifTrue:[Código]`




# Estructuras Iterativas

Smalltalk soporta cuatro tipos tradicionales de iteraciones. Ellos son:

- Hacer algo n números de veces -> *timesRepeat:*
- Hacer algo hasta que se encuentra con una condición false o una condición true -> *whileFalse:* o *whileTrue:*
- Hacer algo usando un índice, comenzando con un valor inicial, y finalizando en un valor final -> *to:do:*

## timesRepeat:

El mensaje *timesRepeat:* ejecuta un bloque de código un número específico de veces. El formato del mensaje es: `número timesRepeat:[código]` donde `número` puede ser cualquier expresión que resulte un entero, y `código` es un bloque de código de cero-argumento.
Ejemplo:

```
"agrega 1 a la variable x tres veces"
|x|
x:=2.
3 timesRepeat:[x:=x+1].
^x.
```

El resultado es 5.

## whileTrue: y whileFalse:

Estos dos mensajes realizan la misma operación, excepto que uno se ejecuta por `true` y el otro por `false`. El formato del mensaje es `[booleano] whileFalse:[código]` o `[booleano] whileTrue:[código]`.
Un `booleano` puede ser cualquier expresión que resulte en un valor de `true` o `false`; debe estar encerrado en un bloque. la expresión `[código]` es un bloque de código de cero-argumento.Ejemplos:

```
"itera mientras x es menor  que y"
|x y|
x:=5.
y:=0.
[x<y] whileFalse:[y:=y+1].
^y

"itera hasta que y es mayor que x"
|x y|
x:=5.
y:=0.
[u<=x] whileTrue:[y:=y+1].
^y
```

## to:do:

El mensaje to:do: ejecuta un bloque múltiple veces, basado en un valor inicial y un valor final. El formato del mensaje es: `numero1 to: numero2 do:[:variable| código]` donde `numero1` y `numero2` pueden ser cualquier expresión que resulte en un número, y `[:variable| código]` es un bloque de código de un-argumento. El bloque se ejecuta para cada número perteneciente al rango entre `numero1` y `numero2`, inclusive. (Este formato es utilizado generalmente con enteros, que varían en el rango de a 1 a la vez). El argumento del bloque de un-argumento equivale al valor actual en el rango.
Ejemplo:

```
"Ejecuta este bloque 3 veces con i referenciado a cada valor entre el
rango de 1 a 3. Al final x valdrá 6"
|x|
x:=0.
1 to:3 do:[:i| x:=x+i].
^x
```

otro ejemplo:

```
"multiplica x por los números entre 5 y 10 inclusive. Al final x valdrá 151200"
|x|
x:=1.
5 to: 10 do:[:incremento| x:= x* incremento].
^x
```

Este último ejemplo, ejecuta el bloque de código 6 veces (desde 5 a 10). la primera vez que se ejecuta el bloque, el valor de incremento es 5, la segunda vez 6, y así sucesivamente, hasta que la última vez es 10.

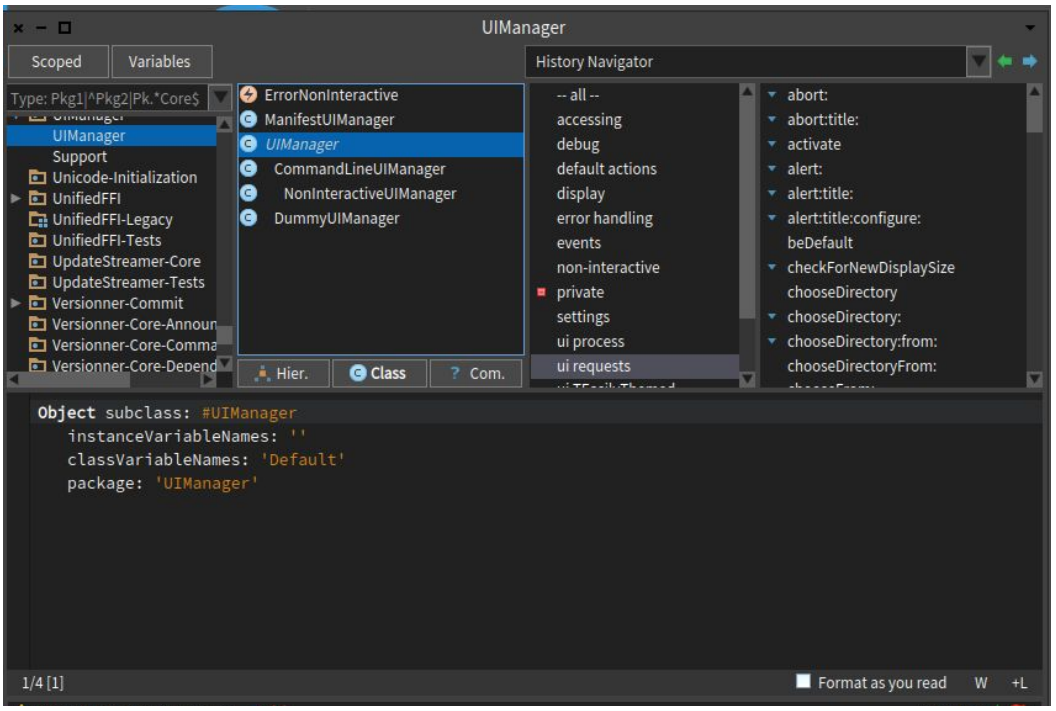
Tomando el primer ejemplo presentado, podríamos querer barrer un intervalo de 1 a 10, pero que evaluará la expresión del bloque cada cierto momento, como ser cada 2 valores, para estas situaciones podremos usar el mensaje `to:by.do:`, obteniendo una expresión como: `numero1 to:numero2 by:deCada numero2 do:[:variable| código]` , donde deCada es el parámetro que será un número o una expresión que evaluada devolverá un número y determinará "cada cuanto va a ejecutar el bloque". Por lo que podremos tener el siguiente ejemplo:

```
|x|
x:=0.
1 to:10 by:2 do:[:i| x:=x+i].
^x
```

En el ejemplo, el bloque se ejecuta 5 veces (i= 1, 3, 5, 7 y 9), debido a que el parámetro pasado en "`by:2`" es cada 2.

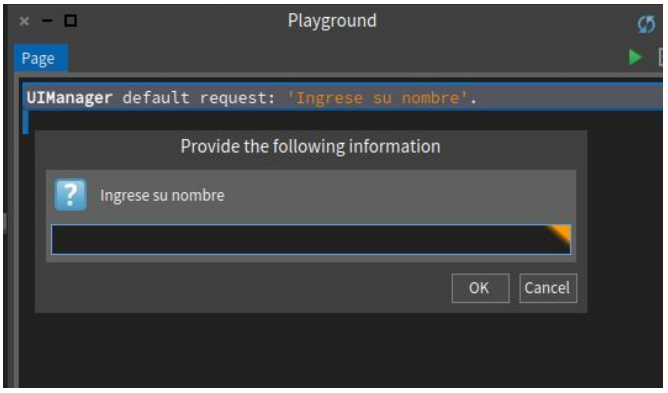
## Solicitar, Obtener, ingresar y mostrar resultados

Hay situaciones en las que se necesita solicitar por pantalla, confirmar o mostrar valores o información. Para ello, particularmente para los ejercicios de las primeras unidades se puede hacer uso de la clase **UIManager** de Pharo.
Para saber qué métodos tiene implementados, iremos a System Browser y buscaremos la clase **UIManager**.

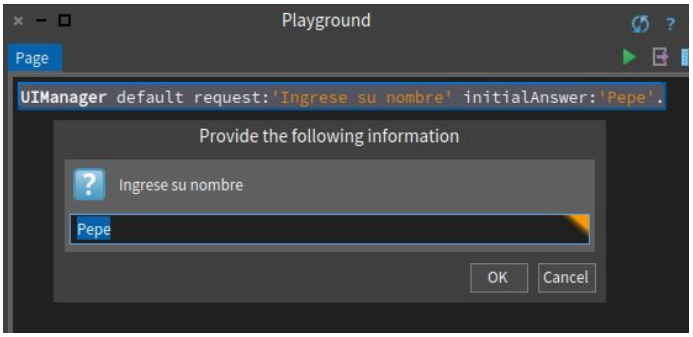



En la columna de la derecha, podrán revisar todos los métodos implementados que soporta. Algunos de ellos que más se utilizan podrán ser por ejemplo:

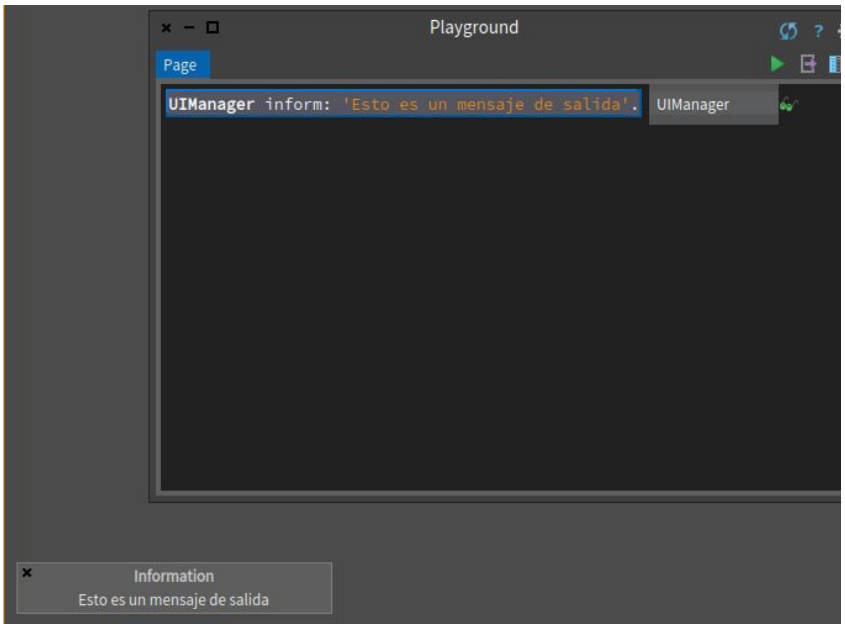
```
UIManager default request: 'Ingrese su nombre'.
```



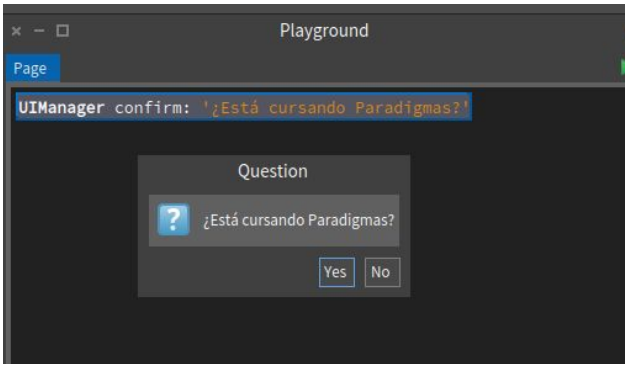

```
UIManager default request:'Ingrese su nombre' initialAnswer:'Pepe'.
```




```
UIManager inform: 'Esto es un mensaje de salida'.
```



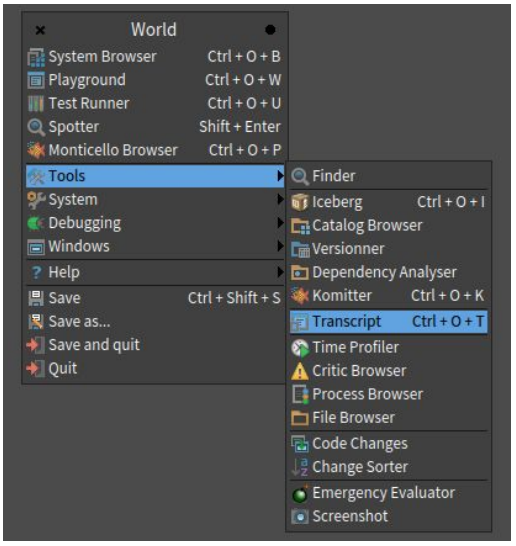

```
UIManager confirm: '¿Está cursando Paradigmas?'
```




Cada una de ellas y las demás implementadas en la clase, pueden ser personalizadas a gusto y adaptarlo a la necesidad que se presente.

## Ventana Transcript

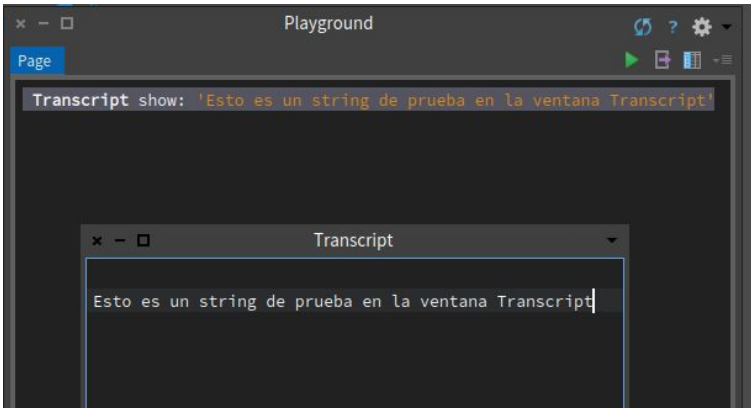
Hay en casos en donde es conveniente usar la ventana de Transcript. Esta ventana desde Pharo es accesible de la siguiente manera:




Para poder mostrar resultados en dicha pantalla, desde el Playground, solo consiste en enviar un mensaje show: a la variable Global reservada **Transcript**, con el string a mostrar, por ejemplo:

```
Transcript show: 'Esto es un string de prueba en la ventana Transcript'
```

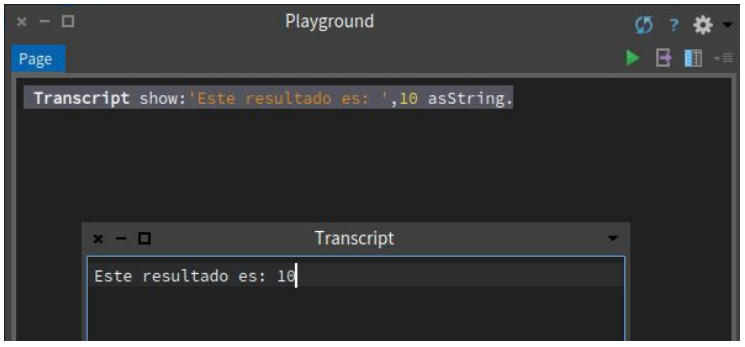
Obtendremos como resultado en la ventana del Transcript:




Hay que tener en cuenta que lo que se va a enviar como parámetro al mensaje show: deberá de ser string, por lo que si queremos mostrar como por ejemplo el resultado de una operación, debemos primero convertirlo a string y si es necesario concatenar con otros strings, como por ejemplo:

```
Transcript show:'Este resultado es: ',10 asString.
```

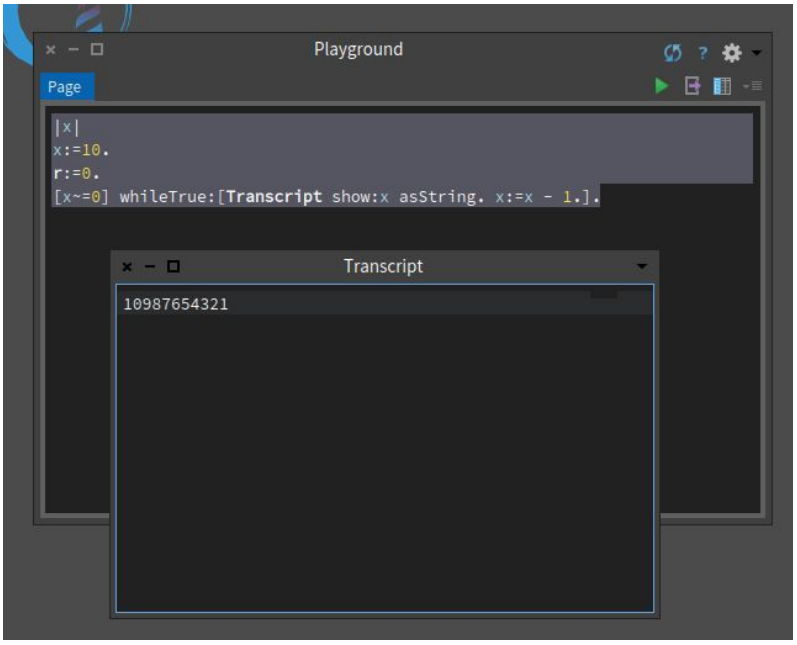
Obtendremos como resultado en la ventana del Transcript:



Incorporando a un ejercicio con iteración donde debemos mostrar los valores de x que van a ir decrementándose hasta 0, lo haremos mostrando en dicha ventana de la siguiente manera:

```
|x|
x:=10.
[x~=0] whileTrue:[Transcript show:x asString. x:=x - 1.].
```

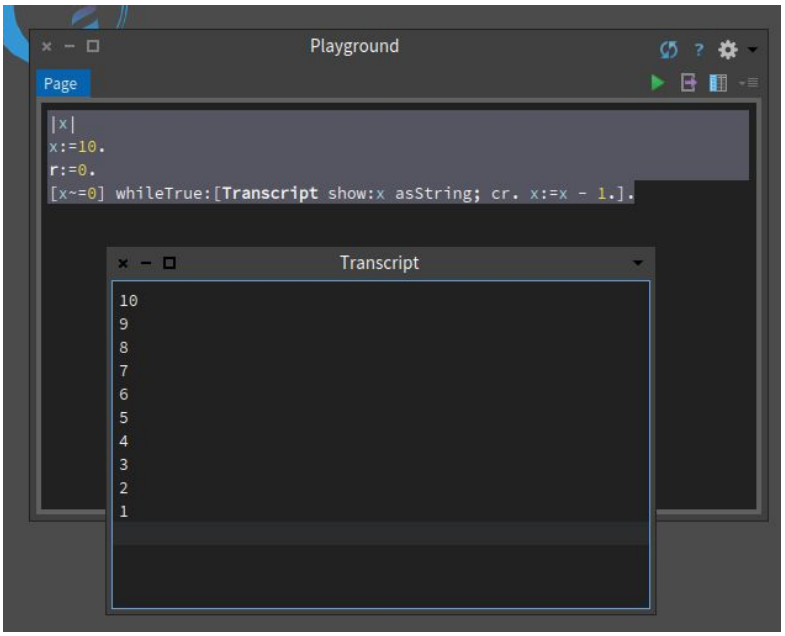
Notar que al ejecutar obtendremos un resultado en la ventana del Transcript de la siguiente manera:




Como podremos ver no es legible, para este caso podremos proceder a realizar saltos de lineas para poder obtener un resultado más prolijo, para ello acudimos al mensaje *cr* en conjunto con ; para un mensaje en cascada. Por lo que el código nos quedará:

```
|x|
x:=10.
[x~=0] whileTrue:[Transcript show:x asString; cr. x:=x - 1].
```

El resultado que obtendremos será:



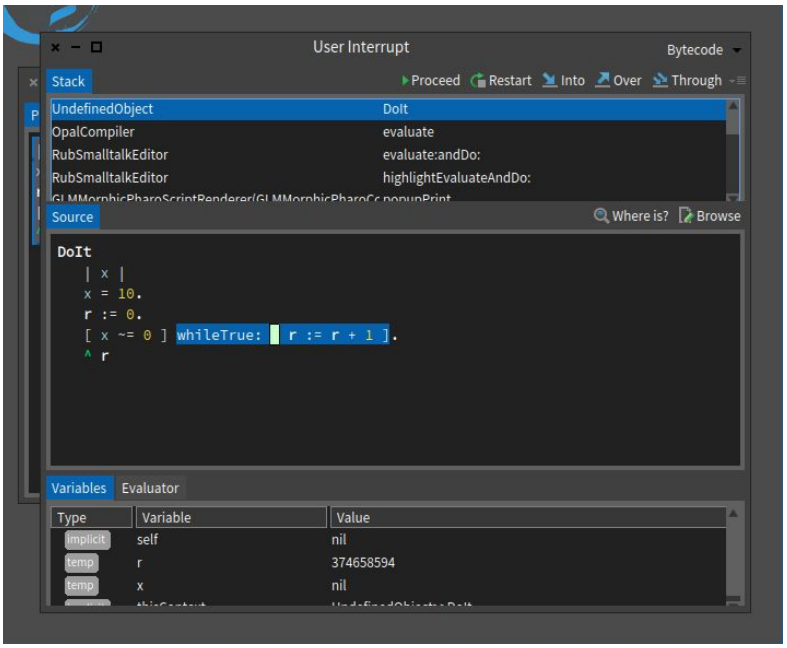

# Problemas habituales

## Bucles infinitos

Muchas veces caemos por error en una iteración que nunca termina cuando estamos codificando una solución (bucle infinito), lo que al ejecutar hará que en Pharo en particular se cuelgue. En estos casos es recomendable detener la ejecución y para eso acudimos a las teclas (**Alt+.**), de esta manera detendremos la ejecución.
Por ejemplo, el siguiente código generará un bucle infinito debido a que si prestamos atención nunca estamos decrementando la variable x y por lo tanto la condición del whileTrue siempre se mantendrá igual, x no va a variar y por lo tanto el bloque después del mensaje *whileTrue:* se seguirá ejecutando.

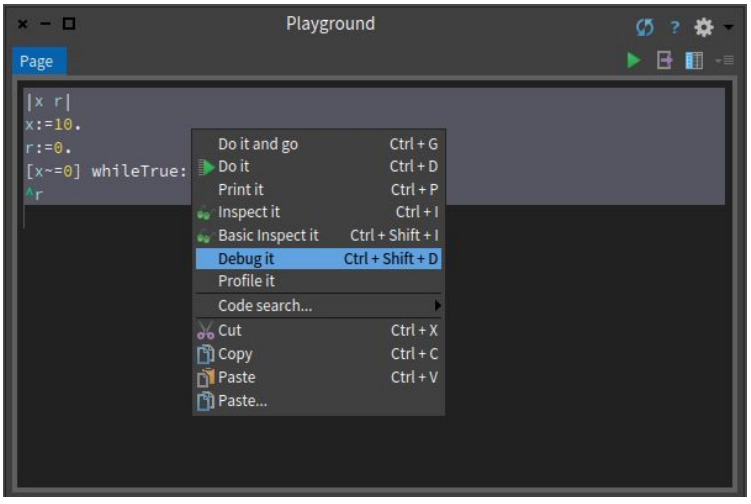
```
|x r|
x:=10.
r:=0.
[x~=0] whileTrue:[r:=r+1].
^r
```

Al detener con las teclas (**Alt+.**) nos saldrá una ventana similar a lo que sigue.

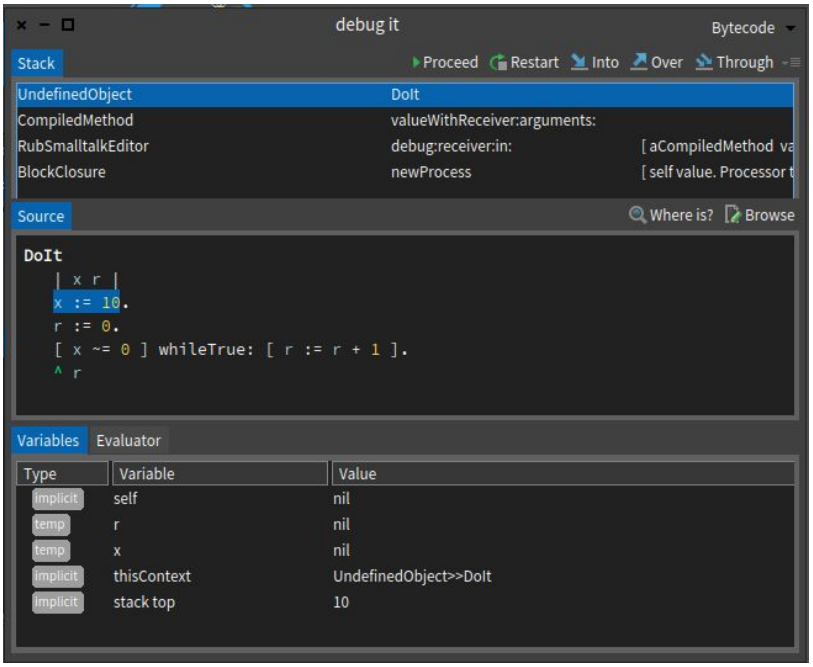


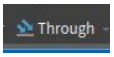

## Debug

Ocurre ante ciertas circunstancias en donde podríamos no tener problemas de sintaxis, pero si en comportamiento (no está realizando lo que deseo). Por lo cual recurrimos a las herramientas de pruebas o de debug, para analizar el comportamiento que se presenta. Para el mismo mismo en vez de ejecutar nuestro código procedemos a seleccionar nuestro código en cuestión y botón derecho del mouse y elegimos:

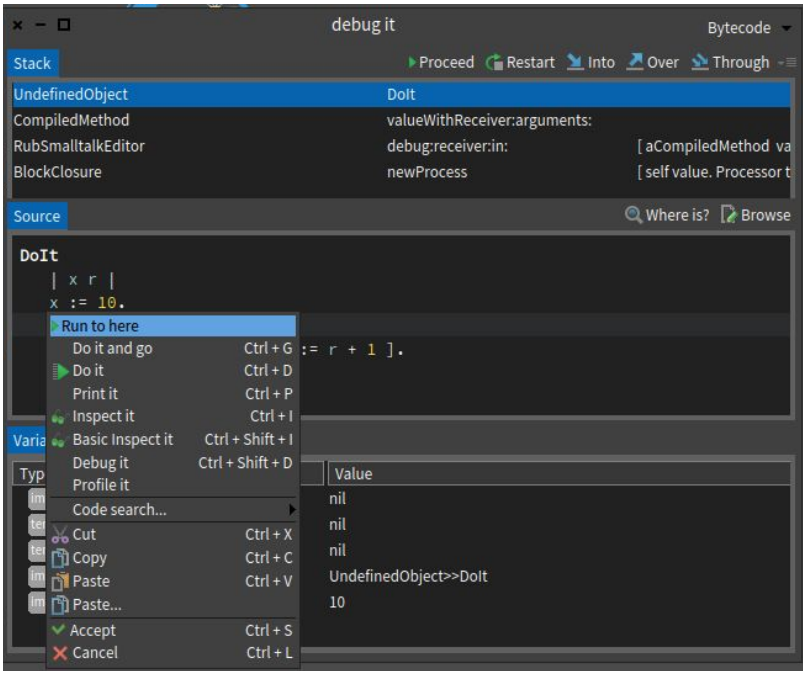



hora tendremos la pantalla de debug, en donde:




En la sección del medio tendremos nuestro código, en la sección inferior nuestras variables definidas y las del motor de Pharo.
La herramienta nos permite realizar una ejecución paso a paso, esto basta solamente con 

. A medida que vayamos avanzando veremos el cambio en las variables de la sección inferior.
Ocurre ante ciertas circunstancias en donde deseamos arrancar o ver el estado en un cierto punto, para no ejecutar desde el inicio podremos pararnos en la línea hasta donde queramos ejecutar y desde ahí presionando botón derecho del mouse elegimos:



De esta manera muy simple y básica podremos ir debugueando nuestro código y corrigiendo. Existen muchas funcionalidades presentes que quedan a cargo del lector inspeccionar.

## Anexo: Resumen de mensajes más utilizados en clases

| Expresión en Pharo | Explicación |
|---|---|
| Expresión ifTrue: [Sentencias] | |
| Expresión ifFalse: [Sentencias] | |
| Expresión ifTrue: [Sentencias] ifFalse:[Sentencias] | |
| [Expresión] whileTrue:[Sentencias] | |
| unEntero timesRepeat:[Sentencias] | |
| Expresión to:Expresión do:[:indiceBucle\| Sentencias] | |
| Expresión to:Expresión by:Expresión do:[:indiceBucle\| Sentencias] | |
| Expresión do:[:indice\| Sentencias] | |
| \|var1 var2\| | definición de variables var1 y var2 |
| Variable:=Expresión | asignación de una expresión a una variable definida |
| ^Expresión | expresión de retorno o salida |
| "esto es un comentario" | añade un comentario |
| true, false | constantes lógicas |
| Expresión = Expresión | preguntar por igualdad |
| Expresión ~= Expresión | preguntar por distintos |
| Expresión == Expresión | idéntico |
| Expresión ~~ Expresión | no idéntico |
| Expresión > Expresión | preguntar por mayor |
| Expresión >= Expresión | preguntar por mayor e igual |
| Expresión < Expresión | preguntar por menor |
| Expresión <= Expresión | preguntar por menor e igual |
| Expresión \| Expresión | operación o |
| Expresión & Expresión | operación y |

| | |
|---|---|
| Expresión or:[ Expresión ] | operación lógica or |
| Expresión and:[ Expresión ] | operación lógica and |
| Expresión xor:( Expresión ) | operación lógica xor |
| 23.5e3 | constante flotante |
| 1999 | un entero |
| 2/3 | expresión fraccionaria |
| + , -, *, / | operadores de suma, resta,multiplicación y división |
| Expresión // Expresión | divide y trunca hacia -infinito |
| Expresión \\ Expresión | devuelve el resto de la operación |
| Expresión negated | obtener una expresión negada |
| Expresión abs | obtener el valor absoluto de una expresión |
| Expresión sqrt | raíz cuadrada |
| unaBase**unExponente | una expresión elevada a un entero |
| Expresión squared | elevado al cuadrado |
| Expresión factorial | obtener su factorial |
| Expresión not | una expresión opuesta |
| Expresión isNil | preguntar si es nil |
| Expresión isNotNil | preguntar si no es nil |
| Expresión isNumber | preguntar si es in integer |
| Expresión isFloat | preguntar si es un real o flotante |
| Expresión isArray | preguntar si es un arreglo |
| Expresión isVowel | preguntar si es una vocal |
| Expresión isUpperCase | preguntar si se encuentra en mayúsculas |
| Expresión isLowerCase | preguntar si se encuentra en minúsculas |
| Expresión isLetter | preguntar si es una letra |
| Expresión asCharacter | convertir una expresión a caracter |
| Expresión asString | convertir una expresión a string |
| Expresión asUppercase | convertir una expresión a mayúsculas |
| Expresión asLowerCase | convertir una expresión a minúsculas |
| unString substrings | convertir una string a un arreglo de substrings (quitando espacios en blancos) |
| unString substrings: unSubstrings | convierte un strings en un arreglo de substrings quitando el string pasado por parámetro |

| | |
|---|---|
| Caracter asciiValue | obtener de un caracter su equivalente en el código ascii |
| Expresión reversed | obtener el reverso de la expresión |
| Expresión size | obtener el tamaño de una expresión.Uso en colecciones |
| Variable inspect | inspeccionar el contenido de una variable en un momento dado donde se llama |
| Expresión class | obtiene la clase del objeto en cuestión |
| UIManager default request: 'mensaje de solicitud' | solicitar valores por pantalla, la salida es un string |
| UIManager inform: 'mensaje de salida' | devolver en una ventana un mensaje |
| Transcript show: 'cadena' | colocar en la ventana del Transcript un string |
| Transcript nextPut: unCaracter | colocar en la ventana del Transcript un caracter |
| Transcript cr | colocar en la ventana del Transcript un salto de línea |
| Transcript space | colocar en la ventana del Transcript un espacio |
| Transcript tab | colocar en la ventana del Transcript una tabulación |
| unObjeto printOn:unaVentana | dado un objeto imprime sobre la ventana señalada |
| Array new: tamañoDeLaLista | Crea una nueva instancia de la clase array con el tamaño ingresado(posiciones con valores en nil) |
| OrderedCollection new | Crea una nueva instancia de la clase OrderedCollection |
| Dictionary new | Crea una nueva instancia de la clase Dictionary |
| Array new | Crea una nueva instancia de la clase array con el tamaño ingresado(posiciones con valores en nil) |
| unaColeccion size | obtiene en valor entero el tamaño de una colección dada |
| una Coleccion at: clave | dada una colección devuelve el contenido de la posición indicada |
| unaColeccion at: clave put:unvalor | dada una colección en la posición indicada coloca un objeto |
| unaColeccionOrdenada add:unObjeto | añade a la colección un objeto |
| unaColeccionOrdenada addFirst:unObjeto | añade al inicio de una colección un objeto |
| unaColeccionOrdenada addLast:unObjeto | añade a la colección al final de la misma un objeto |
| unaColeccionOrdenada removeFirst | remueve el primer objeto de la colección |
| unaColeccionOrdenada removeLast | remueve el último objeto de la colección. |
| Expresión atAllPut: Expresión | dada una colección coloca en todas las posiciones un |

| | objeto |
|---|---|
| Expresión occurrencesOf: Caracter | de una expresión obtiene las ocurrencias del caracter indicado |
| String' indexOfSubCollection: 'ing' | de una expresión, obtiene la posición donde se encuentre lo pasado por parámetro |
| Expresión between:unNumero and:unNumero | dado un rango de números evalúa si se encuentra entre ellos. |

***Autor: Fede A.***
***Historial de revisiones:***

| ***Versión*** | ***Revisado por*** | ***Fecha*** | ***Detalles*** |
|---|---|---|---|
| *1.0.0* | *Fede A.* | *18-08-17* | *Versión inicial. Objeto, mensaje, tipos de mensajes, orden de evaluación bloque,* |
| *1.1.0* | *Fede A.* | *19-08-17* | *Se edita. Métodos, creación instancia, características de la POO, ejemplos de tipos de mensajes* |
| *1.2.0* | *Fede A.* | *20-08-17* | *Se edita. Mensajes más utilizados, operaciones lógicas, operaciones aritméticas, operador not, or, and y xor. Operadores condicionales. Estructuras iterativas* |
| *1.3.0* | *Fede A.* | *21-08-17* | *Se edita. Se completa formato de la versión 1.2.0 y se añaden imágenes ilustrativas* |
| *1.4.0* | *Fede A.* | *24-08-17* | *Se edita. Agrega ventana Transcript y problemas habituales, bucle infinito* |
| *1.5.0* | *Fede A.* | *29-09-17* | *Se edita. Corrección de error en resultados y se anexa el debug.* |