



# **UNIDAD Nº 3**

## **PARADIGMA DE PROGRAMACIÓN**

### **CON ORIENTACIÓN A OBJETOS**

**EL PRESENTE CAPÍTULO HA SIDO ELABORADO, CORREGIDO Y COMPAGINADO POR:**

**ESP. ING. COLACIOPPO NICOLÁS**  
**MGTR. ING. CORSO, CYNTHIA**  
**ESP. ING. GUZMAN, ANALIA**  
**ESP. ING. LIGORRIA, KARINA**  
**ESP. ING. LIGORRIA, LAURA**  
**DR. ING. MARCISZACK, MARCELO**  
**ING. ROMANI, GERMAN**



## ÍNDICE

<i>Objetivos de la Unidad</i> .....	6
<i>Contenidos Abordados</i> .....	6
1. Historia .....	7
2. PARADIGMA ORIENTADO A OBJETOS .....	9
2.1 Conceptos Introdutorios .....	9
2.2 Características de la POO .....	10
2.3 Objeto .....	13
2.4 Los Objetos en el Paradigma Orientado a Objetos .....	15
2.4.1 Abstracción .....	15
2.4.2 Identidad .....	15
2.4.3 Comportamiento .....	15
2.4.4 Capacidad de Inspección .....	16
2.5 Colaboraciones: Mensaje y método .....	17
2.6 Colaboradores Internos / Estado interno .....	21
2.7 Encapsulamiento.....	21
2.8 Interpretación de las colaboraciones: Colaboradores externos .....	22
2.9 Polimorfismo.....	23
2.10 Mensajes a mí mismo – self .....	24
2.11 Destrucción, Creación y Representación del conocimiento .....	25
2.12 Clases .....	28
2.12.1 Definición de comportamiento común .....	28
2.12.2 Cómo se crean objetos.....	28
2.12.3 Clases e instancias.....	29
2.12.4 Forma del código Smalltalk .....	29
2.13 Relaciones entre clases .....	30
2.13.1 Asociación .....	30
2.13.2 Agregación / Composición .....	31
2.13.3 Generalización / Especialización .....	32
2.14 Reutilización.....	33



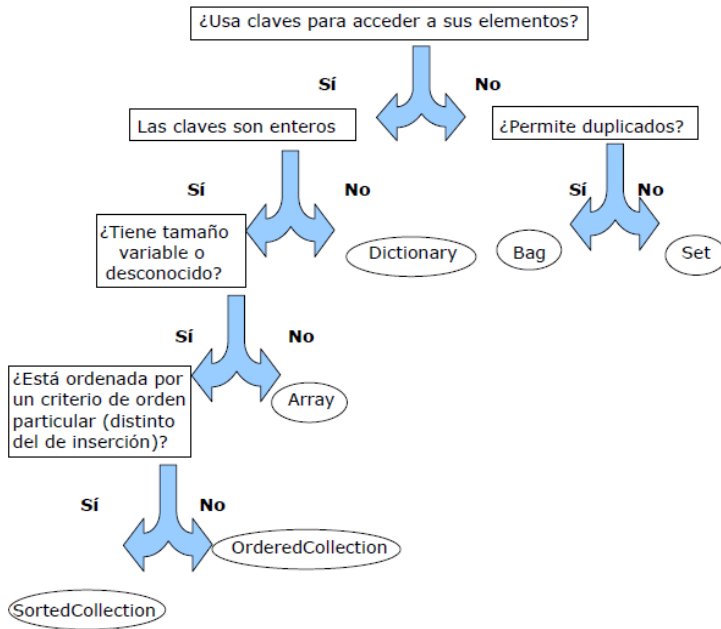
2.15	Herencia .....	33
2.15.1	Estrategias para implementar herencia .....	33
2.15.2	Modelos de implementación .....	35
2.16	Polimorfismo.....	36
2.17	Ambiente .....	37
2.18	Software.....	38
3.	Implementación del POO con Smalltalk .....	39
3.1	Introducción.....	39
3.2	Aspectos importantes de Smalltalk [Budd 91].....	40
3.2.1	Asignación dinámica de memoria.....	40
3.2.2	Asignación de referencias.....	41
3.2.3	Asignación dinámica de tipos .....	41
3.2.4	Objetos polimórficos .....	41
3.3	Sintaxis y Semántica del Lenguaje Smalltalk .....	42
3.3.1	Comentarios .....	42
3.3.2	Operador de asignación (:=) .....	42
3.3.3	Literales y Variables .....	42
3.3.3.1	Variables Locales o Temporales .....	43
3.3.3.2	Variables Globales.....	43
3.3.4	Pseudo-VARIABLES .....	44
3.3.4.1	Objeto especial nil.....	44
3.3.4.2	true.....	45
3.3.4.3	false .....	45
3.3.4.4	self.....	45
3.3.4.5	super .....	45
3.3.4.6	thisContext .....	45
3.3.5	Símbolos .....	45
3.4	Implementación de Conceptos del POO en Smalltalk .....	46
3.4.1	Clases.....	46
3.4.1.1	Variables de Instancia .....	47
3.4.1.2	Variables de Clase .....	47
3.4.2	Instancias de una clase .....	48



3.4.3	Métodos .....	49
3.4.3.1	Métodos de acceso y modificación .....	50
3.4.3.2	Métodos de inicialización .....	51
3.4.3.2.1	Inicialización de Variables de Instancia.....	51
3.4.3.2.2	Inicialización de Variables de Clase.....	52
3.4.4	Paso de Mensajes .....	52
3.4.4.1	Pasos Implicados en la Ejecución de un Mensaje .....	55
3.4.4.2	Envío de más de un Mensaje a un Objeto.....	56
3.4.4.3	Reglas de Prioridad de Mensajes .....	56
3.4.5	Resolución de un Caso de Estudio .....	57
3.5	Implementación de Composición en Smalltalk.....	59
3.5.1	Resolución de un Caso de Estudio .....	59
3.6	Implementación de Herencia en Smalltalk .....	62
3.6.1.1	Definir una Subclase .....	63
3.6.1.2	Uso de super .....	64
3.6.1.3	Herencia de variables .....	64
3.6.1.4	Inicialización de Atributos en una Clase Hija.....	65
3.6.1.5	Herencia de métodos .....	66
3.6.1.6	Enlace Dinámico .....	66
3.6.1.7	Clases abstractas .....	67
3.6.2	Resolución de un Caso de Estudio .....	68
3.7	Implementación de Polimorfismo en Smalltalk .....	70
3.7.1	Resolución de un Caso de Estudio .....	71
3.8	Biblioteca de Clases de Smalltalk .....	75
3.8.1	Clase Object .....	75
3.8.2	Objeto indefinido.....	77
3.8.3	Magnitudes.....	77
3.8.3.1	Números.....	79
3.8.3.1.1	Enteros.....	81
3.8.3.1.2	Fracciones .....	81
3.8.3.2	Caracteres .....	82
3.8.3.3	Fechas y Horas.....	82



3.8.3.3.1	Clase Date .....	82
3.8.3.3.2	Clase Time .....	84
3.8.4	Booleanos .....	86
3.8.5	Bloques .....	87
3.8.5.1	Estructuras de Control .....	89
3.8.5.1.1	Estructuras condicionales .....	89
3.8.5.1.2	Estructuras repetitivas determinadas.....	90
3.8.5.1.3	Estructuras repetitivas indeterminadas (o condicionales) .....	91
3.8.5.1.4	Resolución de Caso de Estudio .....	92
3.9	Colecciones en Smalltalk.....	96
3.9.1	Introducción .....	96
3.9.2	Set.....	98
3.9.3	Bag.....	99
3.9.4	OrderedCollection .....	100
3.9.5	SortedCollection .....	102
3.9.6	Array .....	104
3.9.7	De un Tipo a Otro .....	105
3.9.8	Operaciones sobre Colecciones.....	106
3.9.8.1	Creación de Instancias.....	106
3.9.8.2	Consulta .....	107
3.9.8.3	Añadir y Remover Objetos .....	109
3.9.8.4	Recorrido.....	111
3.9.9	Dictionary .....	113
3.9.9.1	Inserción y Remoción de Elementos .....	115
3.9.9.2	Recuperación de Elementos.....	117
3.9.9.3	Recorrido.....	122
3.9.10	Cómo elegir la Colección más adecuada? .....	122



SortedCollection ..... 122

4. Anexo I: Ejemplo ..... 123

    Ejemplo: La Golondrina ..... 123

5. Bibliografía ..... 135



### **OBJETIVOS DE LA UNIDAD**

Que el alumno comprenda acabadamente los mecanismos fundamentales que dan origen a este paradigma.

Que el alumno utilice para la resolución de problemas un lenguaje con OO puro en su concepción y representación.

### **CONTENIDOS ABORDADOS**

Conceptos fundamentales. Abstracción de datos y ocultamiento de la información. Estructura de un objeto. Métodos y mensajes. Clasificación. Clase. Concepto de generalización-especialización. Herencia, simple y múltiple. Polimorfismo.

Conceptos del modelo de objetos en SmallTalk. Desarrollo de la sintaxis de objetos en Smalltalk. Expresiones literales. Caracteres, secuencia de caracteres, símbolos y números. Expresiones de asignación y variables. Asignación. Tipos de variables. Variables privadas: de instancia nombradas e indexadas, argumentos y temporales. Variables compartidas: de clase, globales y pool. Variables especiales: self y super. Expresiones de mensaje. Sintaxis de un mensaje. Tipos de mensajes: unario, binario y palabra clave. Orden de precedencia en la evaluación de expresiones. Expresión de mensajes en cascada. Expresiones de bloque. Clase Context. Bloques con y sin argumentos. Evaluación de bloques. Métodos y expresiones de retorno. Sintaxis de la definición de un método. Significado de la expresión de retorno. Métodos de clase e instancia. Implementación de Composición en Smalltalk. Implementación de Herencia en Smalltalk (Definición de una Subclase, uso de super, herencia de variables, inicialización de atributos en una clase Hija, herencia de métodos, clases abstractas). Implementación de Polimorfismo en Smalltalk. Colecciones en Smalltalk: Introducción, jerarquía, colecciones básicas (Set, Bag, OrderedCollection, SortedCollection, Array, Dictionary), operaciones básicas, conversión entre colecciones.

Lenguaje asociado: Smalltalk. Pharo 5.0. Imagen, ambiente de objetos, definición y uso de clases y objetos.

## **1. HISTORIA**

Simula ´67 (1967): Precursor de la tecnología de objetos, tal cual se la conoce actualmente. Fue el primero de los lenguajes orientado a objetos. Varios años después de su desarrollo, casi todos los lenguajes modernos comenzaron a utilizar sus principios de orientación a objetos. Así fue como se popularizaron términos como clases, objetos, instancias, herencia, polimorfismo, etc.

Año 1969: Alan Kay (personaje fundamental de la informática) Matemático, Psicólogo, Computólogo y Músico Estudió como mejorar el proceso cognitivo utilizando herramientas informáticas. Estudió dentro del MIT como los niños llegaban a conceptualizaciones y abstracciones complejas a través de juegos con LOGO.



En la década del 60 estaban en auge los Mainframes; Alan Kay propuso que computadora se necesitaría para favorecer el proceso cognitivo. Definió conceptualmente a DYNABOOK (Computadora personal tipo agenda, que tuviera todo lo necesario para comunicarse con otras personas) Esta debía contener todo lo necesario, sino que siempre disponible, lo que constituyo las bases de los ordenadores portátiles y tablet PC actuales, considerado por algunos como el arquitecto de los sistemas modernos de ventanas GUI (Interfaz Gráfica de usuario).

Año 1970: Xerox funda el Centro de Investigación de Palo Alto (PARC - Palo Alto Research Center). Alan Kay fue uno de los miembros principales del centro, desarrollando prototipos de estaciones de trabajo en red, usando el lenguaje de programación SMALLTALK. Estas invenciones fueron posteriormente comercializadas por APPLE en el APPLE MACINTOSH.

Alan Kay se dedica a construir su DINABOOK (Hw y Sw). Hardware, pequeño, autónomo, con almacenamiento y capacidad gráfica + mouse, diseñando entonces el concepto de interfaz gráfica tal cual lo conocemos en la actualidad. En forma conjunta hubo que generar un proyecto de Software que debía ser intuitivo, fácil de usar y de entender. Este proceso fue liderado por Adele Goldberg, tomando como idea a las propuestas de Simula y surge SMALLTAK (1972).

Año 1976: Se entrega la 1er versión a Empresas, Universidades y Laboratorios. Lo importante es que el mismo responde e implementa las ideas de Alan Kay. Smalltalk es el único ambiente de objetos puro que ha tenido difusión comercial

Año 1980: Xerox PARC da por finalizado el proyecto. En este punto comienza a haber distintas corrientes e implementaciones del POO. Adele Goldberg funda una empresa que comercializa a SMALLTALK como entorno de desarrollo para grandes empresas.

Año 1984: aparece SMALLTALK-V que fue pensado para utilizarse en una arquitectura de PC-IBM básica.

Año 1986 - 1990: Aparece C++ es un diseñado a por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso Lenguaje de programación "C", con mecanismos que permitan la manipulación de Objetos. En ese sentido, desde el punto de vista de los Lenguajes orientados a Objetos, el C++ es un lenguaje híbrido. Esta idea no fue considerada como muy brillante, pero este nuevo lenguaje tuvo un éxito rotundo.

James Rumbaugh, propone la metodología OMT (Object Modeling Technique) que es una de las metodologías de análisis y diseño orientadas a objetos, más maduras y eficientes que existen en la actualidad. La gran virtud que aporta esta metodología es su carácter de abierta (no propietaria), que le permite ser de dominio público.

Bertrand Meyer: Hace un enfoque diferente de la Orientación a Objetos el que se soporta desde la Teoría de la Computación, desarrollando Eiffel. El cuál es más OO que C++ pero mucho menos que SMALLTALK, el cuál es admitido en sus principios constitutivos pero que aporta e incorpora mucho de Ingeniería de Software, permitiendo construir software de alta calidad y seguro. No tiene gran aceptación en software comercial.

Brad Cox desarrolla Objective-C (más cerca de Smalltalk que C). Es un lenguaje de programación orientado a objetos creado como un superconjunto de C para que implementase un modelo de objetos parecido al de Smalltalk. Originalmente fue creado por Brad Cox y la corporación StepStone en 1980. En 1988 fue adoptado como lenguaje de programación de NEXTSTEP.

Luego aparecen los metodólogos (Jacobson, Booch, Coad, Jourdon, Palmer, Rebeca Wirsf-Brook) y más tarde UML (Jacobson, Rumbaugh, Booch).

Año 1997: Alan Kay ya en Disney colaboró en la creación del proyecto SQUEAK, el entorno OO, basado en SMALLTALK para la creación y experimentación multimedia. Trabaja en forma conjunta con Ian Ingalls (Arquitecto de la Máquina virtual SMALLTALK). SQUEAK es una máquina virtual SMALLTALK construida en





SMALLTALK. Como aspecto sobresaliente es que: Es un proyecto abierto y corre en todas las plataformas de Hardware existente.

En marzo de 2008 comenzó el proyecto **Pharo**, como una bifurcación de Squeak 3.9. La primera versión beta 1.0 fue liberada el 31 de Julio de 2009.

Actualidad: Recientemente Alan Kay, comenzó a trabajar en el “Proyecto Croquet”, que busca ofrecer un entorno 3D en red de código libre para el desarrollo colaborativo.

La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, abstracción, polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existe variedad de lenguajes de programación que soportan la orientación a objetos.

## 2. PARADIGMA ORIENTADO A OBJETOS

En este paradigma, se puede partir de una definición básica que caracteriza al Paradigma: Un programa es un conjunto de Objetos que colaboran entre sí enviándose mensajes.

**PARADIGMA ORIENTADO A OBJETOS**

**OBJETOS + MENSAJE = PROGRAMA**

- Pensar en programar como una entidad viva en la computadora, en donde los objetos evolucionan, modificándose a sí mismos de acuerdo a ciertos factores.

### 2.1 CONCEPTOS INTRODUCTORIOS

A continuación, se describen un conjunto básico de definiciones, de manera de comprender el enfoque adoptado para el desarrollo del presente documento.

Programar con Orientación a Objetos: Para construir un programa usando orientación a objetos, el primer concepto con el que voy a trabajar, y lo primero que va a saltar a la vista, es el del término de objeto.

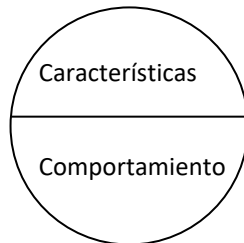
Rápidamente, un objeto es una unidad de software de la cual lo que me importa es: qué le puedo preguntar y/o pedir, y a qué otros objetos conoce. Los objetos responden a los pedidos interactuando con los otros objetos que conoce, y así se va armando una aplicación.

O sea, en vez de enfocarse en rutinas, variables, funciones, punteros, etc.; nos introduce a pensar en objetos: con qué objetos voy a interactuar, sus características, qué les voy a poder preguntar, con qué otros objetos tienen que interactuar estos, y otras cosas que surgen a partir de estas.

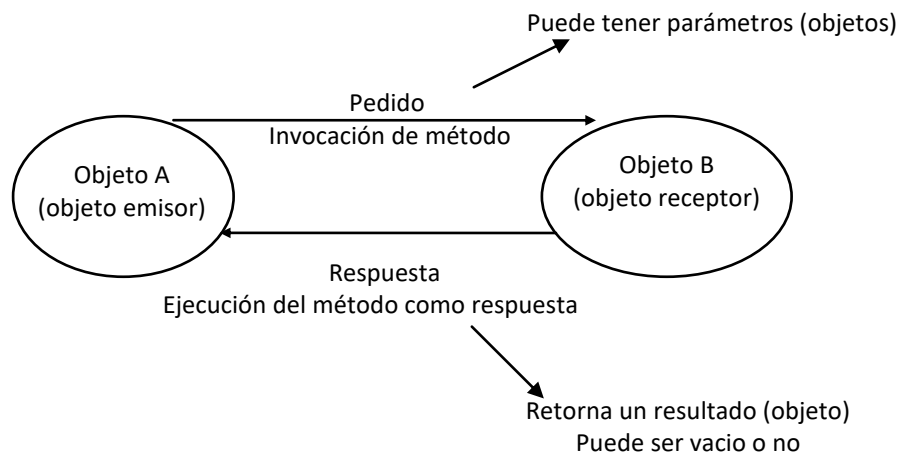
De hecho, el resultado de la tarea va a ser la descripción de estos objetos, cómo se conocen entre ellos, y cómo interactúan.



Para resumir, un objeto posee características y comportamiento:



Y se relaciona con otros objetos a través de mensajes:



## 2.2 CARACTERÍSTICAS DE LA POO

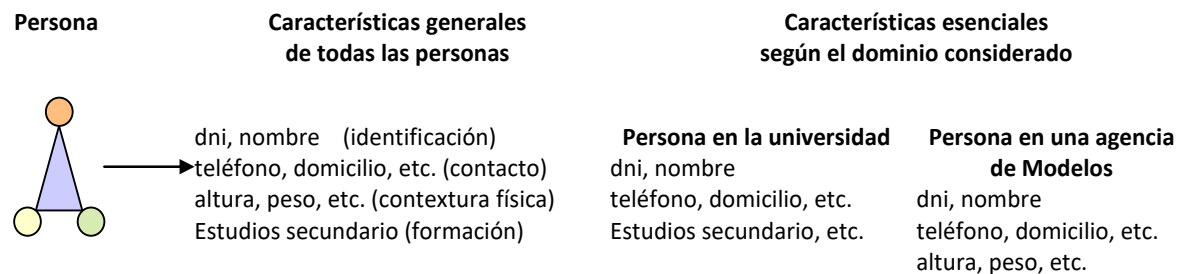
Existe un acuerdo acerca de qué características contempla la "orientación a objetos", las características siguientes son las más importantes:

- **Abstracción:** denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar *cómo* se implementan estas características. Los procesos, las funciones o los

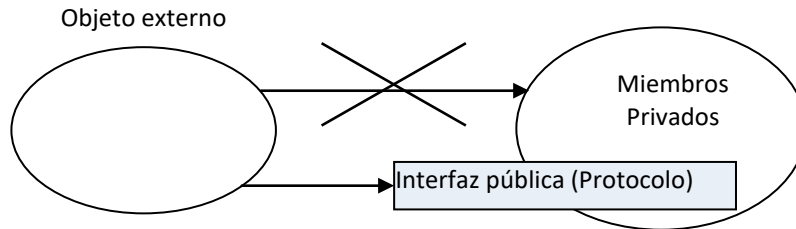


métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

Por ejemplo gráficamente:



- **Modularidad:** Se denomina Modularidad a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la Modularidad de diversas formas.
- **Encapsulamiento:** Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Es común que se confunda este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.
- **Principio de ocultación:** Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una *interfaz* (protocolo) a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción.



- **Cohesión y acoplamiento:** la cohesión mide las responsabilidades asignadas a cada objeto, mientras que el acoplamiento mide las relaciones entre los objetos. En el Modelado Orientado a Objetos, lo que se busca es tener ALTA cohesión (cada objeto se responsabilice por una sola cosa) y BAJO Acoplamiento (Poca o nula interdependencia)
- **Herencia:** las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en *clases* y estas en *árboles* o *enrejados* que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay *herencia múltiple*.
- **Polimorfismo:** comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama *asignación tardía* o *asignación dinámica*. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.
- **Recolección de basura:** la recolección de basura o *garbage collector* es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse manualmente.



## 2.3 OBJETO

Es una abstracción que utilizaremos como representación computacional de una entidad de la realidad tiene unas propiedades particulares (atributos) y una forma de operar con ellas (métodos).

Un objeto representa alguna entidad o ente de la vida real. Puede representar:

- Lo que está vivo: personas, animales, plantas, células, pulmones;
- Las organizaciones: un equipo de fútbol, un ministerio, un grupo de investigación, un club, una cátedra, una empresa, un ecosistema;
- Cosas en lo que se toca es sólo un soporte: un libro, una película, un artículo en una revista, un contrato, una ley;

Cosas bien abstractas: un número, un conjunto, un nombre, una letra, una función matemática

### OBSERVADOR

Para comprender mejor el concepto de objeto emisor y receptor y porque del envío de mensajes, introduciremos el concepto de Observador.

Un objeto es cualquier ente o entidad que tiene alguna utilidad/significado/sentido para el que tiene que trabajar con él, esto quiere decir que nos va a interesar quien va a trabajar/interactuar con el objeto, ese "alguien" por ahora lo vamos a llamar "observador" y vamos a suponer que los observadores son personas.

Distintos observadores van a ver distintos objetos. Es probable que en la visión de un carpintero que describe su negocio aparezcan sillas y no aves, mientras que en la de un ornitólogo (biólogo especializado en aves) lo más probable es que pase lo contrario.

Un observador tampoco es alguien en abstracto, es alguien haciendo algo/en un rol; los objetos que va a ver son aquellos con los que necesita interactuar de alguna forma para poder cumplir su rol. Esto se logra, asumiendo el Rol de analista y produciendo un recorte de la realidad.

Armar modelos y trabajar en base a ellos es la forma que encontramos las personas para poder pensar acerca del mundo complejo en el que vivimos

Por lo tanto un Objeto será una representación esencial de entes de la realidad, el cuál además tendrá definido ciertos comportamientos, que serán requeridos por otros objetos que interactúen con él.

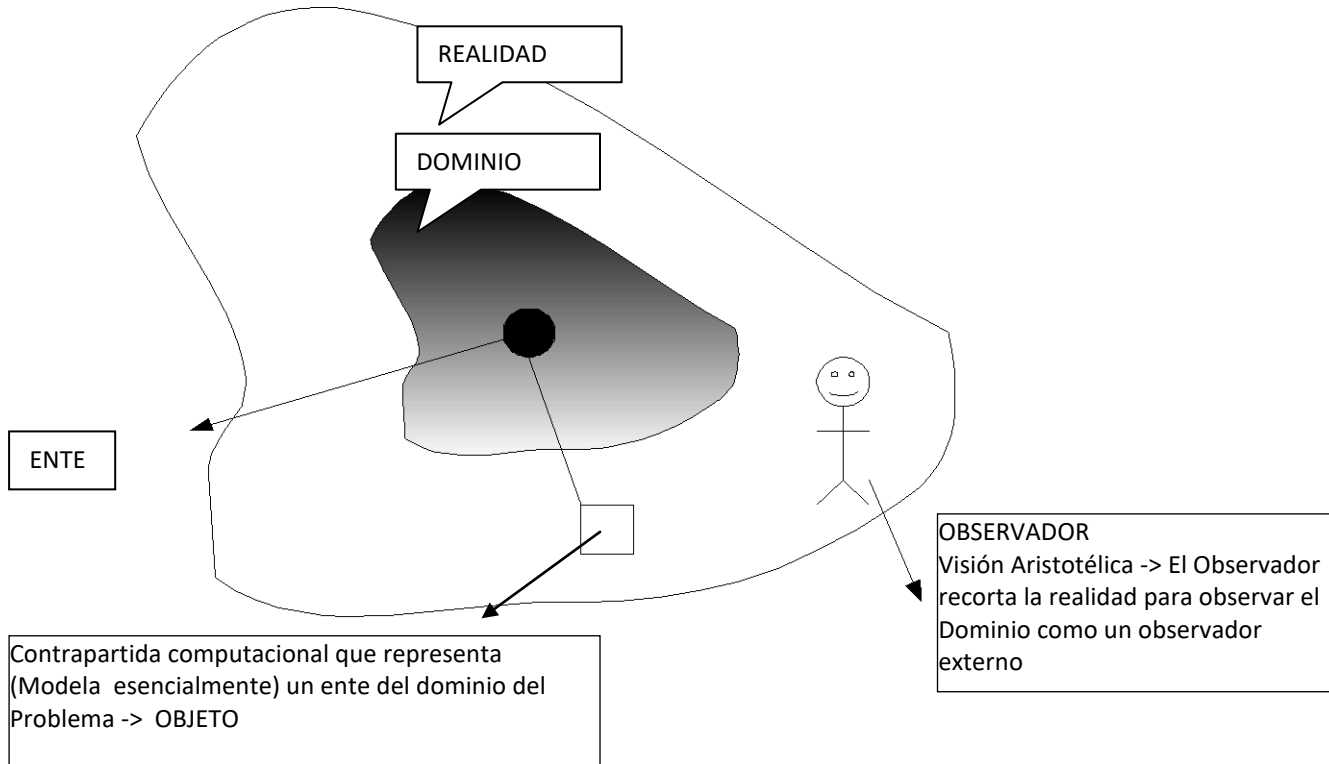


FIG. 5.1. PROCESO DE ABSTRACCIÓN EN POO

En resumen, no podemos definir Objeto si no hablamos de un Sujeto. No podemos hablar de un Observado si no hay un Observador. No se puede definir un término sin el otro.

Entonces, siempre que hablamos de un Objeto estamos diciendo también que ese objeto existe porque hay alguien que lo observa, alguien que reflexiona sobre su existencia.

Si recordamos que un programa orientado a objetos es un conjunto de Objetos que colaboran entre sí enviándose mensajes, podremos inferir que existirán algunos objetos que serán observados y otros observadores.

Ahora la figura del observador cambió, ya no es una persona que lo modela, sino un objeto que es capaz de interactuar con otro, por lo tanto habrá que analizar en el conjunto de objetos cuales son los que interviene.



## 2.4 LOS OBJETOS EN EL PARADIGMA ORIENTADO A OBJETOS

Un objeto dentro de Paradigma Orientado a Objetos debe poseer:

- **Abstracción:** Representación esencial de la Realidad.
- **Identidad:** Todo objeto es igual a sí mismo, y solo a sí mismo.
- **Comportamiento:** Descripto a través de un protocolo
- **Capacidad de inspección**

### 2.4.1 ABSTRACCIÓN

Un Objeto es una abstracción que utilizamos como una representación computacional de entes de la realidad, el cuál responde a dos características:

- **Esenciales:** aquellas que hacen que el ente sea lo que es, que no pueden cambiar, que si falta algo ya no es lo que era.
- **Accidentales:** Un ente puede poseerlos, pero si no los tuviera o se lo cambiara, no dejaría de ser lo que es. (color, tamaño, etc.)

#### Representación esencial

Es un proceso de abstracción, el cual consiste en separar estas dos características claramente, describiendo la esencia del ente.

Un Objeto es una abstracción que utilizamos como representación computacional de entes de la realidad.

Una abstracción no simplifica la realidad, sino que modela sus características esenciales representadas a través de su comportamiento

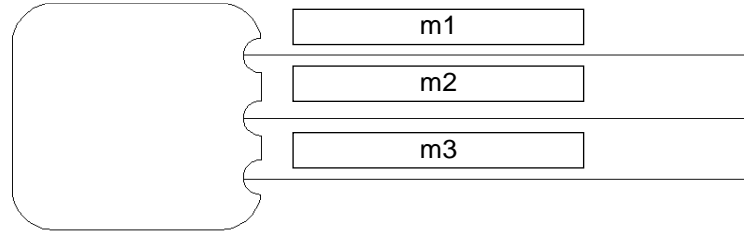
Los objetos en la orientación a objetos son las representaciones computacionales de los entes con los que vamos a interactuar.

### 2.4.2 IDENTIDAD

La identidad es una característica intrínseca de los objetos. Un buen nombre cierra una idea. Identidad es diferente a igualdad. Para verificar una igualdad se deben comparar dos objetos y con un criterio definido.

### 2.4.3 COMPORTAMIENTO

Un objeto también queda representado por su comportamiento, el cual estará definido por el conjunto de mensajes que el objeto pueda responder (para que sirva, como utilizarlo, etc.). Este conjunto de mensajes que el objeto puede responder se lo denomina Protocolo.



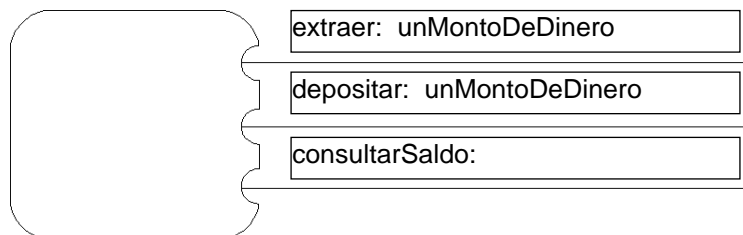
**FIG. 5.2. REPRESENTACIÓN GRÁFICA DE UN OBJETO**

Los mensajes, son el vocabulario que el objeto es capaz de entender, es lo que hay que saber para poder comunicarse con él.

### Protocolo

Conjunto de mensajes que un objeto puede responder. O planteado desde otro punto de vista las reglas a cumplir para poder comunicarse con un objeto.

A través del protocolo de un objeto, definimos la utilidad del ente que estamos representando, o sea su esencia.



**FIG. 5.3 REPRESENTACIÓN DE UN OBJETO CON SU PROTOCOLO**

El vocabulario que entiende el objeto (Su Protocolo) es el LENGUAJE DEL DOMINIO, y por lo tanto no debe estar expresado en términos de computación sino del dominio.

El protocolo es invariante; si he comprendido la realidad y la he modelado como comportamiento de un ente a través de mensajes, se constituye en un punto invariante del modelo que representa mi programa.

La especificación del protocolo de un objeto es una decisión de análisis, jamás de diseño.

#### **2.4.4 CAPACIDAD DE INSPECCIÓN**

Poder ver en todo momento quiénes son sus colaboradores, y en un determinado instante eventualmente cambiárselo.





## 2.5 COLABORACIONES: MENSAJE Y MÉTODO

¿Qué hace un objeto cuando recibe un mensaje? - Lo único que puede hacer es enviar un mensaje a otro objeto, eso se llama colaborar.

La colaboración se produce cuando los objetos se envían mensajes.

Un método se asocia a un mensaje que el objeto puede responder.

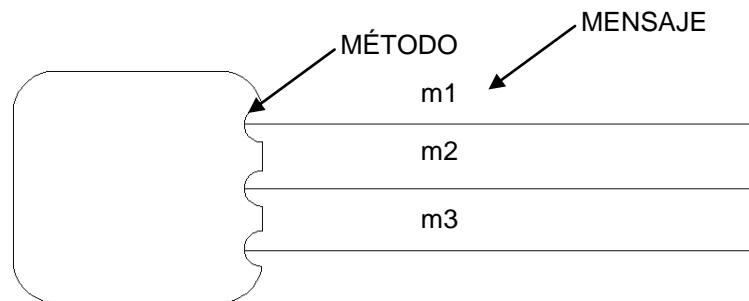
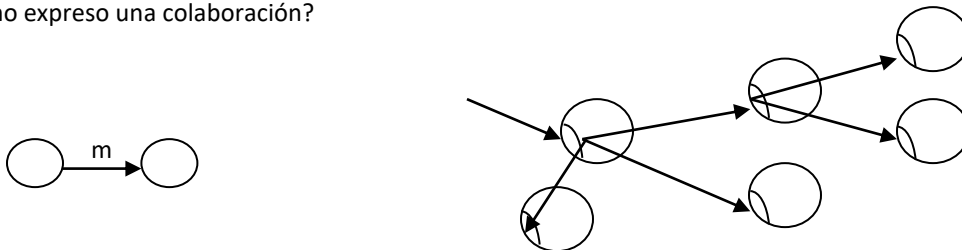


FIG. 5.4. MÉTODO Y MENSAJE

MÉTODOS: conjunto de colaboraciones que un objeto tendrá con otros.

La idea esencial, es que el método le indica al objeto que hacer para cada mensaje que puede recibir.

¿Cómo expreso una colaboración?



### 5.5. FORMA GRÁFICA DE EXPRESAR UNA COLABORACIÓN

¿Y cómo hace un observador para usar un objeto?

Le envía mensajes. Un mensaje es cada una de las formas posibles de interactuar con un objeto. Para pedirle algo a un objeto, lo que hago es enviarle un mensaje. Cada objeto entiende un conjunto acotado de mensajes (Protocolo)



Sí es posible (y altamente deseable, como veremos mas adelante) que haya muchos objetos capaces de entender el mismo mensaje, incluso aunque sean bastante distintos entre sí. Eso permite ir armando un vocabulario mediante el cual uno interactúa con los objetos.

En cada acción de envío de mensaje:

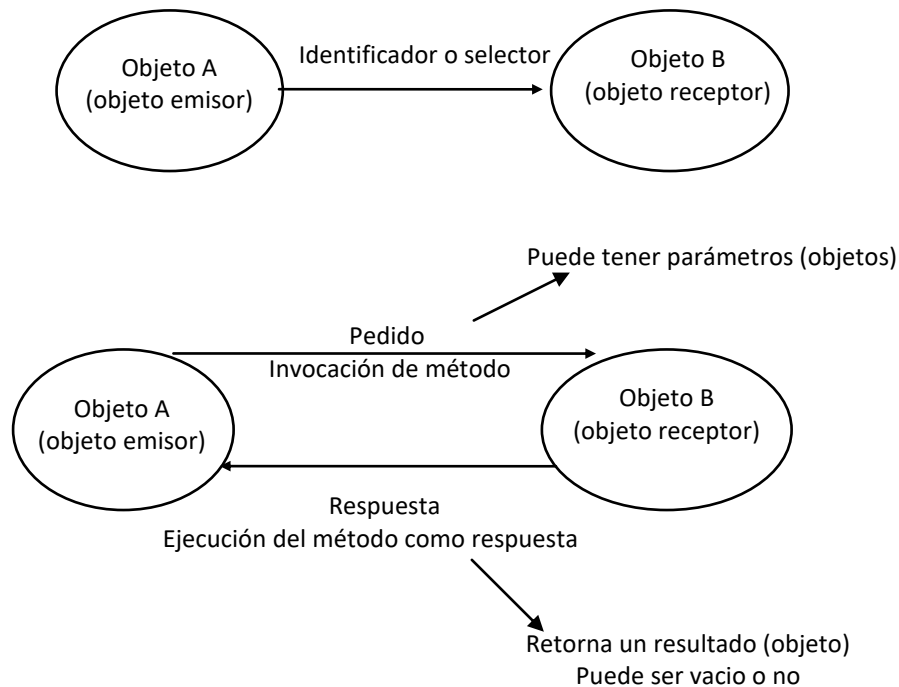
- hay un emisor
- hay un receptor
- hay un nombre, que identifica el mensaje que estoy enviando entre todos los que entiende el receptor.

Como estamos hablando de software, un nombre toma la forma de un identificador. En la jerga de Smalltalk, a este nombre se lo llama selector.

Pasamos en limpio las definiciones prolijas de mensaje y envío de mensaje

mensaje: cada una de las formas posibles de interactuar con un objeto, que se identifica por un nombre y puede llevar parámetros.

envío de mensaje: cada interacción con un objeto. Tiene: emisor, receptor, selector, eventualmente parámetros (que son objetos), y eventualmente un resultado (que es otro objeto).





Por ejemplo, en Smalltalk para enviarle un mensaje a un objeto es:

**{objeto destino mensaje.}**

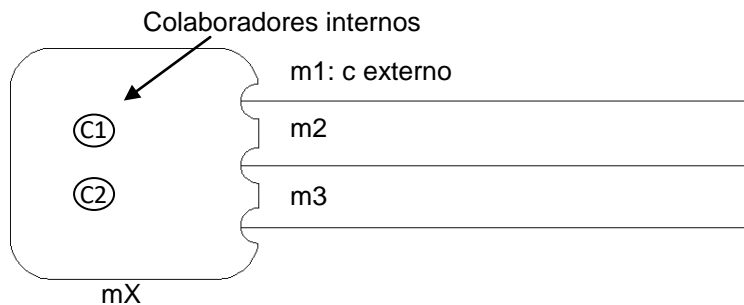
Donde “objeto destino” es el nombre que le di al objeto en el entorno en donde estoy escribiendo el código. Y “mensaje” será el nombre del selector del mensaje propiamente dicho.

Para poder especificar un MÉTODO debo pedir identificar con mis colaboradores.

Podemos pensar en dos tipos de colaboradores:

- Habituales (internos)
- Eventuales (externos)

Un colaborador externo se va a definir especialmente para alguno de los mensajes.



**FIG. 5.6. FORMA GRÁFICA DE EXPRESAR UNA COLABORACIÓN: EXTERNAS E INTERNAS**

Lo que yo conozco de mis colaboradores es su nombre (como yo me refiero a ellos, en general un **rol**);

El Objeto  $mX$  puede colaborar con  $c1$  y  $c2$  (siempre) y con  $c$  externo (siempre que responda a  $m1$ ).

El protocolo del objeto es PÚBLICO, y la definición de métodos y colaboradores es PRIVADA, es interna al objeto. Esta parte interna es la implementación del objeto, y está encapsulada. Puede ser cambiada en cualquier momento mientras se mantenga invariante su protocolo público. Esto es lo que se conoce como ENCAPSULAMIENTO.

Cuando un objeto envía un mensaje a otro, SIEMPRE recibe como respuesta un OBJETO. En SmallTalk ampliamos el concepto de colaboración.

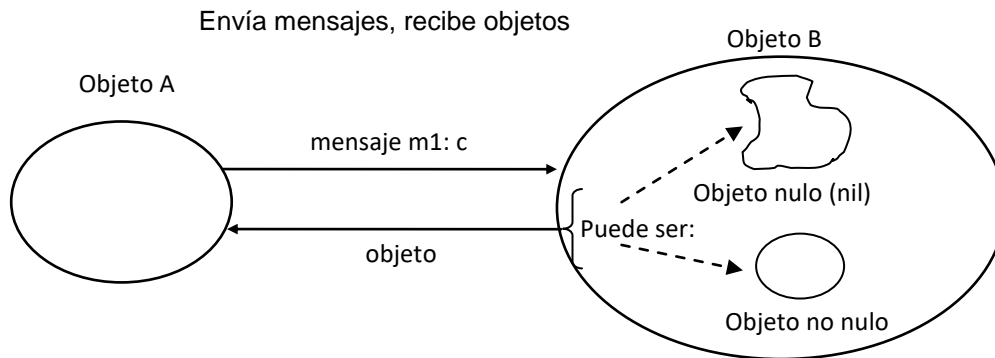


FIG. 5.7. RESPUESTA A MENSAJES

### Aspecto sintáctico

{objeto mensaje. }	Colaboración
^	devolver (objetos)
:=	asignación (a un nombre en determinado objeto)

Un objeto conoce a otro cuando puede hacer una asignación de un nombre a un objeto.

El método es donde se escribe el código que alguien deberá programar, donde “programar” quiere decir: indicar qué mensajes va a entender el objeto (su protocolo), y para cada uno de estos mensajes (selectores) existe un Método donde se escribe el código que se va a ejecutar cuando el Objeto reciba este Mensaje.

O sea: fue un programador el que decidió que el Objeto va a entender un determinado mensaje y escribió el código asociado a ese mensaje. Ese código es el que se va a ejecutar cuando se invoque el mensaje.

La sección de código que se asocia a un mensaje se llama método. Un método tiene un nombre, que es el del mensaje correspondiente; y un cuerpo, que es el código que se ejecuta.

Un detalle: en la jerga de objetos, se usa mucho el término “evaluar” en vez de “ejecutar”.

Con todo esto, pasamos la definición en limpio junto con un corolario

**Método:** sección de código que se evalúa cuando un objeto recibe un mensaje. Se asocia al mensaje mediante su nombre.

**Corolario:** casi todo el código que escribamos va a estar en métodos, que definen qué mensajes entiende cada objeto, y determina qué hacer cuando los recibe.



La única forma que tengo de interactuar con un objeto es enviándole mensajes. Los nombres de mensaje con un parámetro terminan con dos puntos, eso indica que atrás viene un parámetro; los dos puntos forman parte del nombre.

## 2.6 COLABORADORES INTERNOS / ESTADO INTERNO

El conjunto de variables que va a contener un objeto se llama Colaborador Interno o estado interno.

Una “variable” es el nombre de un objeto, entonces lo que me estoy guardando en realidad es el objeto que está nombrando cada variable. Esto va a ir quedando claro a medida que avancemos. Por ahora alcanza que quede claro que una variable no es un nombre de una sección de memoria.

En resumen, el estado interno de un objeto: es el conjunto de variables que contiene, y se define al programar el Objeto.

En la convención Smalltalk, los selectores que piden información tienen el nombre de lo que piden sin ningún agregado; en principio conviene adherir a las convenciones del lenguaje/entorno que usamos. Además, se escribe menos y el código queda más legible.

Una característica interesante de muchos entornos Smalltalk es que puedo hacer que los objetos que creo queden “vivos” en un entorno que a su vez vive dentro del Smalltalk. Puedo crear objetos y después interactuar con ellos cuando quiero.

## 2.7 ENCAPSULAMIENTO

Se puede cambiar la implementación de un objeto, y el usuario de ese objeto no se enteró.

¿Cómo logramos esto? Porque no cambiamos el protocolo del objeto, o sea el conjunto de mensajes que entiende; y lo único que ve el usuario del objeto son los mensajes que entiende, no ve cómo el objeto lo implementa “adentro suyo” (métodos y estado interno).

Trabajando en Smalltalk, el “Observador” no tiene forma de conocer la forma del estado interno de un objeto, ni sus valores; y tampoco puede acceder a detalles sobre cuál es la implementación que está detrás de los mensajes que le envía. Lo único que conoce es el comportamiento que el objeto exhibe (Protocolo); siendo lo único que necesita conocer para poder interactuar con el objeto.

Esta idea de que un observador no ve todos los aspectos de un objeto sino solamente aquellos que le sirven para interactuar con él se llama encapsulamiento; la idea del nombre es que los aspectos internos del objeto se encapsulan de forma tal que los observadores no pueden verlos.

La idea de encapsulamiento también puede observarse en la relación que nosotros tenemos con él.

Al encapsular un objeto, estoy al mismo tiempo acotando y explicitando (haciendo explícitas) las formas posibles de interacción; sólo se puede interactuar con un objeto mediante el comportamiento que exhibe. Esto nos da varias ventajas que pasamos a comentar.

Como las formas de interacción son acotadas y las maneja quien programa el objeto, se hace más sencillo probar si un objeto se comporta correctamente.

También resulta más controlable cualquier cambio que haga en la implementación, siempre que respete el conjunto de mensajes definidos (protocolo)



A su vez, como es más sencillo probar si un objeto “funciona bien” o no, se simplifica darnos cuenta si con un cambio de implementación “rompemos algo”.

Otro efecto es que al explicitar el comportamiento lo estamos documentando, entonces para alguien que quiera usar el objeto que construimos es fácil saber qué usos le puede dar y cómo usarlo. De alguna forma el comportamiento que exhibe el objeto forma un “manual de uso”.

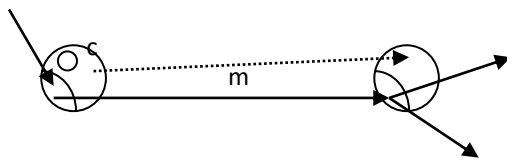
Eso se potencia si seguimos la sabia política de poner abajo del nombre de cada método un comentario que explica qué se puede esperar, como lo hicimos en los ejemplos de código. Es una convención de Smalltalk que definitivamente conviene respetar en el código que escribamos nosotros.

En Smalltalk no hay forma de que un usuario del objeto acceda a la variable (Colaborador Interno), entonces es necesario. Además, el definir el mensaje permite hacer cambios de implementación que resulta lo más conveniente.

Resumiendo encapsulamiento: quien usa un objeto sólo ve lo que necesita para poder interactuar con él, que el comportamiento que exhibe. Los detalles internos quedan encapsulados en el objeto. Y así quedan acotadas y explicitadas las formas posibles de interactuar con un objeto.

## 2.8 INTERPRETACIÓN DE LAS COLABORACIONES: COLABORADORES EXTERNOS

Volvemos a repasar ¿qué es una colaboración?

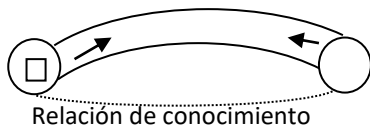


Para que la colaboración sea posible, el objeto de la izquierda debe conocer al objeto de la derecha

( .....→ )

Esto significa que el de la izquierda pone un nombre y, por ejemplo, asigna la identificación de algún objeto.

Podemos pensar la colaboración como un canal por el cual envía colaboradores y recibe un objeto como resultado. En este momento termina la colaboración y se cierra el canal (esto es una metáfora, los objetos no viajan de un lado al otro, sino que se manejan referencias, viajan las identidades de los objetos).



Si envío un colaborador externo, es porque el destino es quien colabora con él. Si recibo un objeto, es porque quiero colaborar con él (enviarle mensajes).

El receptor del mensaje **no tiene idea** de quien le envía el mensaje (no puede dialogar, no es cliente – servidor). Esto es bueno porque está totalmente desacoplado; los objetos están preparados para responder a cualquiera que le envíe el mensaje.

El emisor, asume que existe un objeto con capacidad de responder al mensaje que envió.

El único error que podría ocurrir es que un objeto reciba un mensaje que no entiende:

- a) Mensaje equivocado al objeto correcto
- b) Mensaje correcto al objeto equivocado.

Esto acota y facilita el manejo/descubrimiento de errores.

La colaboración -> un contrato entre emisor y receptor. El emisor debe escribir bien el mensaje y enviar un colaborador externo que sepa hacer cosas que requiere el receptor. Por su lado el receptor debe devolver un objeto que sepa hacer cosas que requiere el emisor.



COLABORACIÓN

TIPO, es el conjunto de mensajes que un objeto debe saber responder.

$$e \in T_1 = \{ m_1, m_2, \dots, m_n \}$$

(el objeto se debe conformar con el tipo  $T_1$ )

El tipo es comportamiento esperado. Tipo es en definitiva un protocolo definido en un contexto.

- Programa: aquello que se está ejecutando

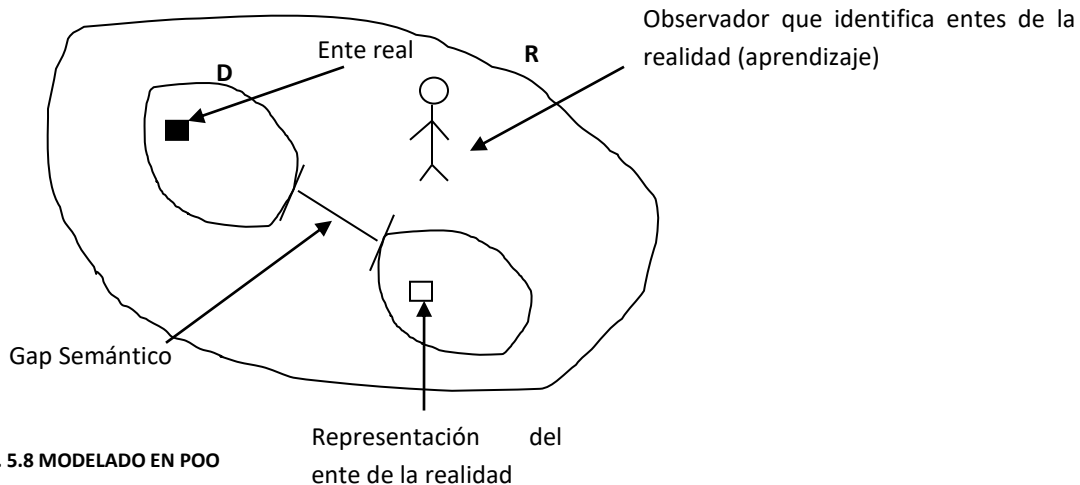


FIG. 5.8 MODELADO EN POO

## 2.9 POLIMORFISMO

Dos o más objetos son polimórficos respecto de un conjunto de mensajes si todos ellos pueden responder de manera semánticamente equivalente, aún si su implementación es distinta.



Se dice que un objeto es polimórfico con otro cuando un determinado observador no puede distinguirlos entre sí. Como en Smalltalk no se puede observar un objeto más que enviándole mensajes, un objeto es polimórfico con otro cuando un determinado observador les envía un conjunto de mensajes y ambos responden de manera indistinguible.

→ De alguna manera los objetos polimórficos son intercambiables.

Tipo= {m1, m2, ...} conjunto de mensajes con un nombre (es un protocolo, no un tipo tradicional).

$e \in T_1$  : El objeto  $e$  conforma con el protocolo T1 (puede responder los mensajes elementos de T1). La “interface” de java hace “algo” parecido.

Para que un Lenguaje, pueda interpretar el Polimorfismo, debe necesariamente poseer binding dinámico o tardío (Ligadura en tiempo de ejecución).

Los objetos polimórficos son intercambiables. Disponiendo así de una enorme flexibilidad en el desarrollo. El acoplamiento entre objetos se hace a través del protocolo, o sea acoplamiento por mensajes.

## 2.10 MENSAJES A MÍ MISMO – self

Para explicar este concepto, lo haremos a través de un ejemplo. Tenemos ahora un objeto que representa a un cliente de un banco tal como lo ve el oficial de crédito.

El cliente entiende dos mensajes montoTopeDescubierto y montoCreditoAutomatico, que devuelven hasta cuánto el cliente puede girar en descubierto, y hasta cuánto se le puede dar de crédito sin necesidad de aprobación. Los dos métodos correspondientes obtienen los valores haciendo cuentas complicadísimas.

Ahora el oficial nos pide a nosotros, que programamos al cliente, que le agreguemos la capacidad de entender el mensaje montoTotalCredito, que es la suma de los dos anteriores.

¿Cómo escribo el método, qué tengo que poner? Veamos.

Cuando escribo un método, estoy parado en el objeto que va a recibir el mensaje que estoy habilitando, en este caso el cliente.

Si soy el cliente y me preguntan mi montoTotalCredito, lo que tengo que devolver es la suma entre mi montoTopeDescubierto y mi montoCreditoAutomatico.

Estos dos montos ¿cómo los obtengo? Enviándome los dos mensajes a mí mismo, para no repetir las cuentas que están en esos dos métodos.

¿Y cómo hago para enviarme mensajes a mí mismo? Fácil, usando la palabra clave self que incluye el Smalltalk. Self es exactamente “yo mismo”, enviarle un mensaje a self es enviármelo a mí, donde





"yo" soy el objeto que recibió el mensaje por el que se disparó el método.

Entonces queda

montoTotalCredito

"El monto total de crédito que me asignaron"

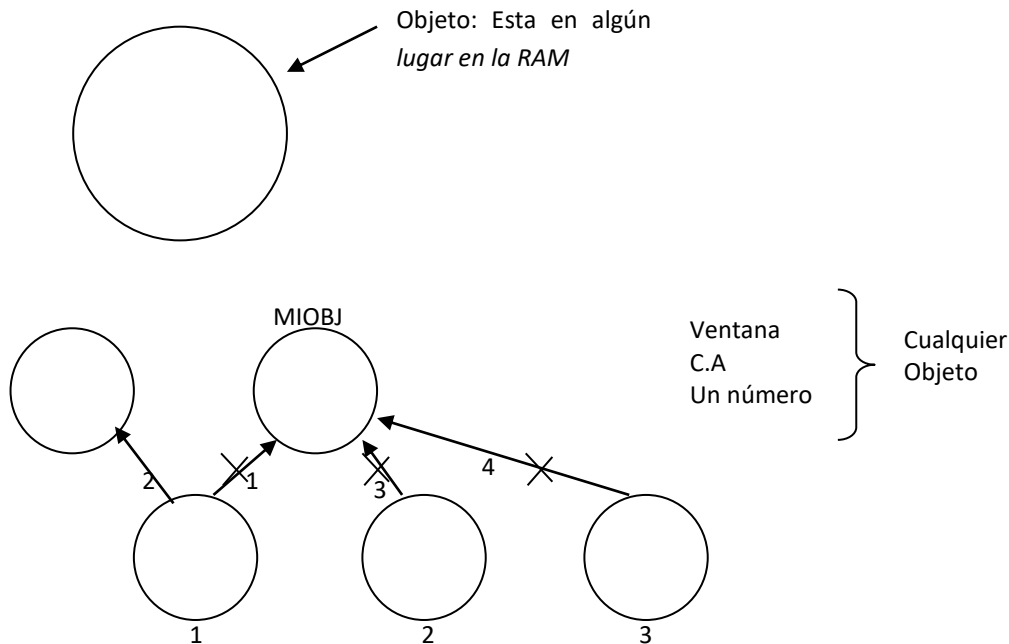
$\wedge$ self montoTopeDescubierto + self montoCreditoAutomatico

La definición formal:

Self: en un método, referencia al objeto que recibió el mensaje correspondiente, que sirve para enviarme mensajes "a mí mismo"

## 2.11 DESTRUCCIÓN, CREACIÓN Y REPRESENTACIÓN DEL CONOCIMIENTO

### DESTRUCCION



Analicemos a MIOBJ de la figura anterior - Nadie lo referencia, nadie lo conoce, entonces, nadie lo puede alcanzar... Ok ¿Quién lo destruye?

Alternativas:

a) que algún otro objeto lo destruye

b) que el mismo se suicide

Tengamos en cuenta que entre 1, 2 y 3 no hay conciencia de que comparten la característica de tener relación con MIOBJ.

Entonces no le puedo decir que 1 mate a MIOBJ pues 1 no sabe que existen 2 y 3.

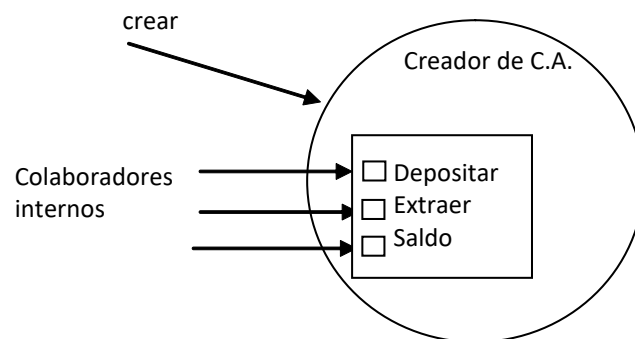
Tengo que tener un objeto que sepa que un MIOBJ se convirtió en basura. Entonces aparece la figura del garbage collector (recolector de basura).

Smalltalk, Java, Python, entre otros, tienen un Garbage Collector que automáticamente libera memoria, el mismo se encarga de destruir los objetos que no están referenciados, es decir, cuando un objeto deja de estar referenciado es desalojado y liberada la memoria.

## CREACIÓN

Analizaremos dos diferentes posibilidades:

1) Que lo cree otro objeto



Crear: Una Caja de Ahorro

El creador de CA "sabe". Tiene el conocimiento de cómo crear CA. Es una fábrica de CA. Y le mando el simple mensaje CREAR.

El conocimiento interno de un objeto es el conjunto de colaboradores internos que él tiene.

2) Que el Objeto sea Clonado

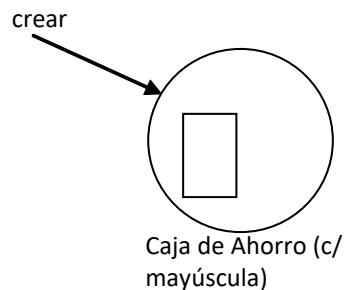


Suponiendo que tengo una CA le mando el mensaje clone y él hace una copia igual pero con identidad diferente. En el momento de nacer el nuevo objeto era igual al que estoy clonando, luego ya está en condiciones y sabe como recibir mensajes y hacer movimientos.

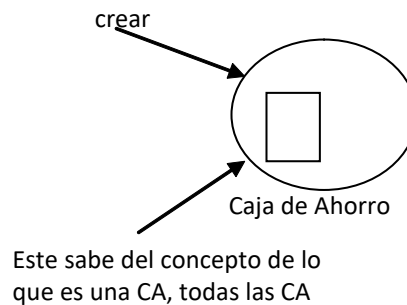
Entonces puedo hacer un clon básico y luego le agregamos comportamientos para que a partir de un objeto básico pase a ser una CA (o lo que sea)

El creador de CA no sabe COMO crear un CA sino que sabe QUE es un CA. O sea, ¿qué le puedo preguntar?, ¿cuál es el protocolo de esa CA?. Además le puedo “ampliar” ese conocimiento y decirle que ahora las CA puedan informar sus movimientos.

Este objeto creador REPRESENTA al CONOCIMIENTO o el CONCEPTO DE CA.



Hemos dado un salto de abstracción. No sólo representan entes de la realidad sino conceptos de la realidad.



Una Caja al recibir un mensaje saldo le puede pedir a Caja de Ahorro que me diga cómo responder al mensaje SALDO. Ergo Caja de Ahorro es un REPOSITORIO de todos los comportamientos de todos los métodos de todos los objetos usados a partir del CONCEPTO que posee Caja de Ahorro

Este último, el Clonado o POO basado en prototipos podría tener CA con comportamiento distinto, en cambio en el claseado todas las CA tienen el mismo comportamiento.

En el clonado puedo conocer un conocimiento inicial y luego voy expresando como aumento mi conocimiento de manera inductiva. Si aparece alguno nuevo, lo copio y le pongo las diferencias.



En la implementación por Clases puedo abstraer y definir la “clase” con las todas las características comunes.

### REPRESENTACIÓN DEL CONOCIMIENTO

El conocimiento está en los objetos y el protocolo que sabe responder. Desde ahora y en lo que se basa la potencialidad del paradigma es que puedo **Inspeccionar** un objeto, ver sus métodos y ver sus colaboradores internos. En este contexto el del POO puro, el concepto de Programar será INSEPCIONAR “CLASES” (objetos que representan clases.)

## **2.12 CLASES**

Hasta ahora hemos trabajado con objetos que creamos y tienen su propio comportamiento, el cuál es distinto al de todos los demás.

Es hora de hacernos algunas preguntas ¿Qué pasa si quiero tener objetos que se comporten de la misma manera? ¿Tengo que codificar lo mismo de nuevo cada vez?

Como ya hemos vistos en creación de objetos, la forma más habitual (aunque no la única – Existen dos formas: clases y prototipos) de resolver ese problema es a partir del concepto de clase. En lugar de definir cada objeto por separado, defino una clase con las características que serán comunes a los objetos, y luego voy a crear los objetos a partir de esta clase.

### **2.12.1 DEFINICIÓN DE COMPORTAMIENTO COMÚN**

La idea de clase nos permite definir un molde para luego a partir de ahí crear muchos objetos que se comporten de la misma manera. Ahora, ¿qué es lo que compartirían todos los objetos?, y ¿qué es lo que no queremos que compartan?

Lo primero en que pensamos es que todos los objetos compartan los mismos métodos.

Por otro lado, cada objeto tendrá la necesidad de poder representar un determinado estado, el cual se reflejará de acuerdo a su estado interno.

.Hasta acá vemos que la clase me estaría definiendo dos cosas:

- El comportamiento común (métodos) a todos los objetos
- El conjunto de variables que tiene que tener cada objeto.

### **2.12.2 CÓMO SE CREAN OBJETOS**



Hasta aquí en ningún momento nos ocupamos del problema de la creación de los objetos; la aparición del concepto de clase nos permite describir con precisión cuándo y cómo se crean objetos.

El hecho de crear la clase y definir sus variables y métodos no crea ningún objeto. Una clase será el molde a partir del cual se crearán objetos a instancia de la clase.

```
ObjetoCreado := NombreClase new
```

Esto me devolverá un objeto nuevo, de la clase NombreClase. Decimos que este nuevo objeto es instancia de la clase, ya que fue creado utilizando a la clase como molde.

### 2.12.3 CLASES E INSTANCIAS

Volvamos un poco a la palabra instancia. Dijimos que el objeto creado por la sentencia:

```
NombreClase new
```

es una instancia de la clase NombreClase, y que en realidad es un objeto. La palabra instancia se usa cuando queremos remarcar la relación entre los objetos y las clases,

Terminemos esta sección bajando las definiciones de clase e instancia.

Clase: molde a partir del cual se crean los objetos; y en el que se definen los métodos y el conjunto de variables que tendrán los objetos que se creen a partir del molde

Instancia: cada objeto es instancia de la clase que se usó como molde para crearlo. Cada objeto es instancia de exactamente una clase.

Las instancias de una clase entienden, y tienen la capacidad de responder los mensajes para los cuales hay métodos definidos en la clase.

### 2.12.4 FORMA DEL CÓDIGO SMALLTALK

Con lo que vimos sobre clases e instancias, ahora sí podemos escribir código Smalltalk usando las herramientas estándar, e interactuar con los objetos que quedan definidos.

- Se definen clases.
- Los métodos se escriben en las clases, y las variables se definen para las clases. O sea, todo el código está en las clases, excepto el que ponemos en ventanas de interacción.



- Los objetos se crean instanciándolos, para lo cual se usa la sentencia: Clase new.
- Los objetos no tienen nombre propio.

## 2.13 RELACIONES ENTRE CLASES

Ya definido el concepto de Clase, es necesario explicar cómo se pueden interrelacionar dos o más clases (cada una con características y objetivos diferentes).

Según G. Booch [Booch, 94] existen tres clases básicas de relaciones:

- Asociación (conexión entre clases)
- Agregación / Composición (relaciones de pertenencia)
- Generalización/especialización (relaciones de herencia)

### 2.13.1 ASOCIACIÓN

Es una relación entre clases. Diremos que dos (o más) clases tiene una relación de asociación cuando una de ellas tenga que requerir o utilizar alguno de los servicios (es decir, acceder a alguna de las propiedades o métodos) de las otras. Se da cuando una clase usa a otra clase para realizar algo.

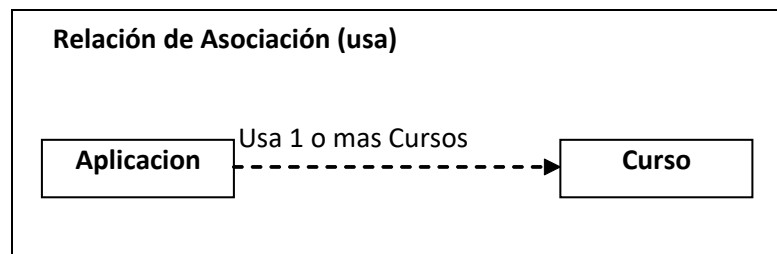
Las relaciones de asociación crean enlaces entre objetos. Estos enlaces no tienen por qué ser permanentes (en la mayoría de los casos, no lo son). Los objetos deben tener entidad fuera de la relación (a diferencia de las relaciones de composición).

Esta relación permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Para validar la asociación, la frase “Usa un”, debe tener sentido, por ejemplo:

- El ingeniero *usa* una computadora
- El cliente *usa* tarjeta de crédito.

Un ejemplo gráfico:





### 2.13.2 AGREGACIÓN / COMPOSICIÓN

Esta relación es utilizada cuando una clase se compone de otras clases, es un tipo de asociación que indica que una clase es parte de otra clase.

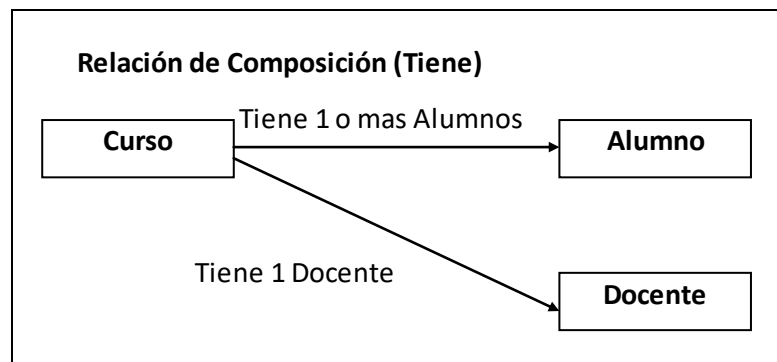
Se presenta entre una clase TODO y una clase PARTE que es componente de TODO. La implementación de este tipo de relación se consigue definiendo como atributo un objeto de la otra clase que es parte-de.

Los objetos de la clase TODO son objetos contenedores. Un objeto contenedor es aquel que contiene otros objetos.

Es un tipo de relación **dependiente** en dónde un objeto más complejo es conformado por objetos más pequeños. En esta situación, la frase "Tiene un", debe tener sentido:

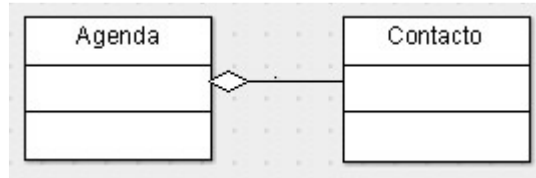
- **El auto tiene llantas**
- **La pc tiene un teclado**

Un ejemplo gráfico:



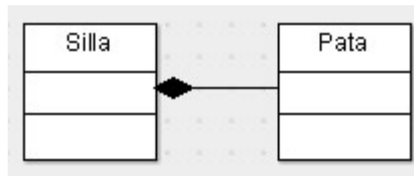
Hay dos tipos de especializaciones de esta relación entre clases.

- **Agregación:** La agregación implica una composición débil, si una clase se compone de otras y quitamos alguna de ellas, entonces la primera seguirá funcionando normalmente, por ejemplo un objeto de tipo Agenda tiene una lista de objetos de tipo Contacto, si quitamos algún objeto de tipo Contacto no afectamos la estructura básica del objeto de tipo Agenda.



- **Composición:** La Composición es una forma fuerte de composición, donde la vida de la clase contenida debe coincidir con la vida de la clase contenedor. Los componentes constituyen una parte del objeto compuesto.

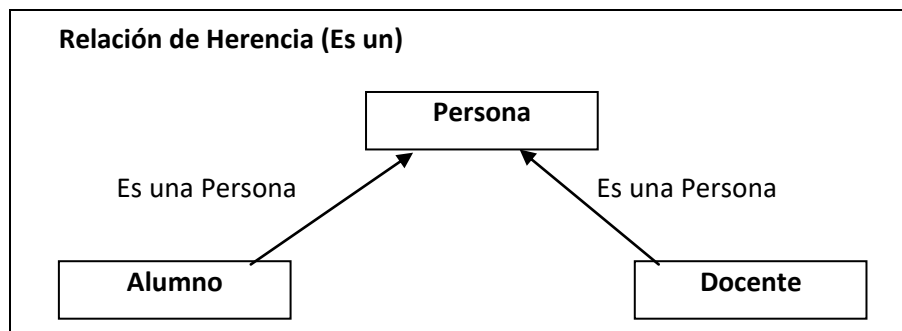
El tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye, por ejemplo si tenemos un objeto de tipo Silla, que está compuesto de cuatro objetos de tipo Pata, si eliminamos un objeto de tipo pata, quedará con tres, lo cual rompe la definición original que tenía el tipo Silla, La silla no puede existir sin sus patas.



### 2.13.3 GENERALIZACIÓN / ESPECIALIZACIÓN

De todas las relaciones posibles entre las distintas clases y objetos, hay que destacar por su importancia en O.O la relación de **herencia**, ésta es una relación entre clases que comparten su estructura y el comportamiento, la estudiaremos más en detalle en la sección 2.15.

Un ejemplo gráfico:







## 2.14 REUTILIZACIÓN

Hay dos mecanismos en POO para aplicar reutilización, es decir, para construir clases utilizando otras clases:

- **Composición:** Una clase posee objetos de otras clases (relación *tiene un*). Se puede reutilizar los atributos y métodos de otras clases, a través de la invocación de los mensajes correspondientes.
- **Herencia:** Se pueden crear clases nuevas a partir de clases preexistentes (relación *es un*). Se puede reutilizar los atributos y métodos de otras clases como si fueran propios.

## 2.15 HERENCIA

La **herencia** es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. Es la característica clave de los sistemas orientados a objeto para propiciar entre otros aspectos la reusabilidad.

### 2.15.1 ESTRATEGIAS PARA IMPLEMENTAR HERENCIA

En general el concepto de herencia, se refiere al mecanismo por el cual los objetos comparten comportamiento. En el mundo de las clases se denomina **herencia**, mientras que en el de los prototipos recibe el nombre de **delegación**; en esta forma alterna de implementar conocimiento común, la distinción entre clases e instancias no es necesaria: **cualquier objeto pueden ser un prototipo**.

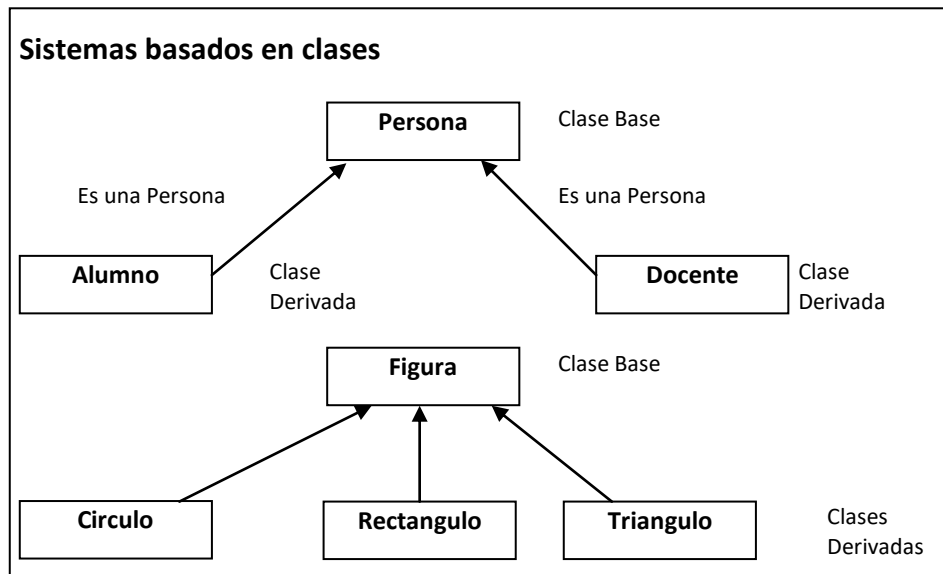
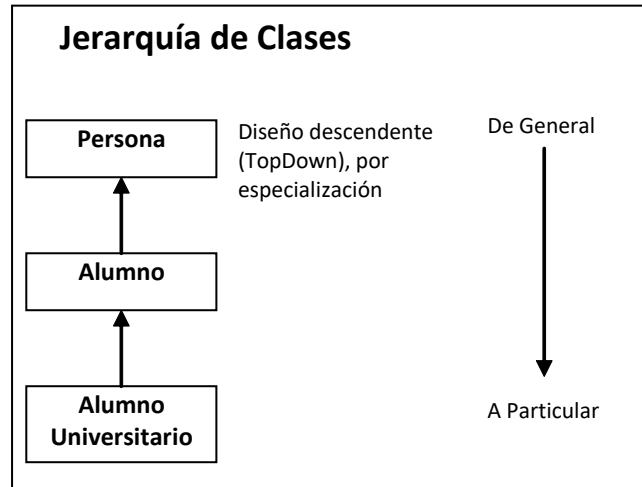
Es por esto que existen dos estrategias básicas, para implementar el modelo de herencia: Clases-Herencia y Prototipos-Delegación:

#### 2.15.1.1 SISTEMAS BASADOS EN CLASES

Mediante una clase (molde, plantilla) es posible definir la estructura y el comportamiento de un conjunto de objetos, en donde se detallan los atributos y métodos que compartiran todas las instancias de la misma determinando su estado y comportamiento.

Las clases se organizan jerárquicamente, y una nueva clase puede reutilizar la estructura y comportamiento de otras previamente definidas.

Lenguajes que lo implementan: Smalltalk, Haskell, C++, Java, entre otros



#### 2.15.1.2 SISTEMAS BASADOS EN PROTOTIPOS

Las clases no suponen una necesidad práctica en el desarrollo orientado a objetos, los lenguajes basados en prototipos prescinden de ellas y sus funciones se llevan a cabo mediante mecanismos alternativos.

La herencia se obtiene a través de la clonación de objetos ya existentes, que sirven de prototipos, extendiendo sus funcionalidades.



Es un estilo de programación orientada a objetos en el cual, las "clases" no están presentes.

Lenguajes que lo implementan: Self, JavaScript, entre otros.

### 2.15.1.3 COMPARACIÓN DE MODELOS

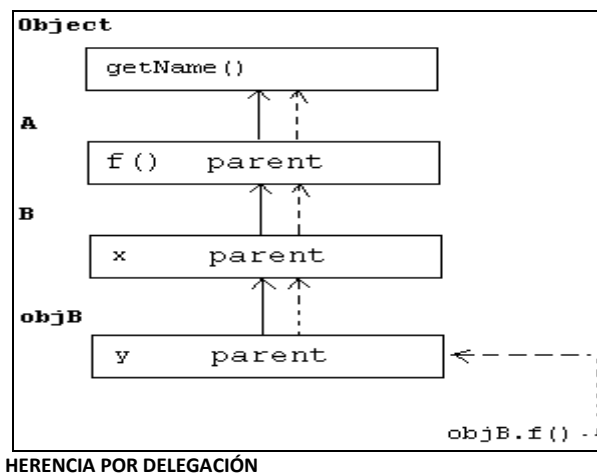
Sistema basado en clases	Sistema basado en prototipos
Los objetos pueden ser de dos tipos generales, las clases y las instancias. Las clases definen la disposición y la funcionalidad básicas de los objetos, y las instancias son objetos "utilizables" basados en los patrones de una clase particular.	Las clases actúan como colecciones de comportamiento (métodos) y estructuras que son iguales para todas las instancias, mientras que las instancias llevan los datos de los objetos.
Hay una distinción entre la estructura y el comportamiento.	Hay una distinción del estado.

## 2.15.2 MODELOS DE IMPLEMENTACIÓN

### 2.15.2.1 HERENCIA POR DELEGACIÓN

El objeto tiene uno o más atributos parent, de forma que cuando no puede responder a un mensaje, le reenvía éste a su padre.

Existe una cadena de objetos apuntando a sus padres, hasta llegar a un objeto padre de todos.

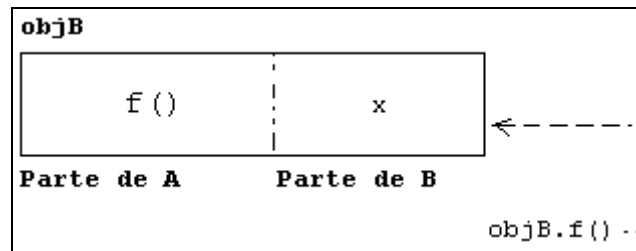




### 2.15.2.2 HERENCIA POR CONCATENACIÓN

El objeto está compuesto por las partes que define cada una de las clases de las que hereda.

Todos los atributos y métodos heredados están disponibles en el mismo objeto (si bien se necesita la clase para poder interpretarlos).



HERENCIA POR CONCATENACIÓN

### 2.15.2.3 COMPARACIÓN DE MODELOS

Para representar en máquina, características y comportamientos compartidos por elementos de un conjunto, denominados objetos, se propone: el tratamiento tradicional de definir objetos clase que empaquetan el conocimiento necesario para fabricar objetos instancia a partir de ellas, y la definición de objetos prototípicos que puedan clonarse para crear otros nuevos.

Los dos modelos difieren básicamente en “a quién se le atribuye el comportamiento” y “el momento en que se fijan los patrones de comunicación entre objetos”.

Sobre la herencia implementada por delegación o por concatenación, se puede decir que con respecto a la forma de aplicar el comportamiento compartido, y a la forma de implementarlo, resulta más flexible y por lo tanto otorga características dinámicas, lo que facilita la exploración de dominios en conjunto con una metodología apoyada en los prototipos. Mientras que la herencia por concatenación es más rígida en su definición pero más uniforme en su aplicación y por ende es posible, establecer una jerarquía de herencia y de tipos correspondientes lo que facilita la construcción de herramientas que soporten este mecanismo. Por lo tanto se puede afirmar que concatenación y delegación pueden ser implementadas en ambos ambientes ya sean de clases o prototipos y el mecanismo de implementar la herencia estará en función del propósito y funcionalidad que se persiga en la construcción del sistema.

Tener en cuenta que la herencia por concatenación es más eficiente porque hereda solo lo que es diferente, no duplica.

Lenguajes: Self y Eiffel.

## 2.16 POLIMORFISMO

La palabra polimorfismo proviene del griego y significa que posee varias formas diferentes. Este es uno de los conceptos esenciales de una programación orientada a objetos. Así como la herencia está relacionada con las clases y su jerarquía, el polimorfismo se relaciona con los métodos.



Esta propiedad, como su mismo nombre sugiere múltiples formas, se refiere a la posibilidad de acceder a un variado rango de funciones distintas a través de la misma interfaz. O sea, que, en la práctica, un mismo identificador puede tener distintas formas (distintos cuerpos de función, distintos comportamientos) dependiendo, en general, del contexto en el que se halle inserto.

Otra definición:

Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en “tiempo de ejecución”, esta última característica se llama **asignación tardía** o **asignación dinámica**. Algunos lenguajes proporcionan medios más estáticos (en “tiempo de compilación”) de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

En general, hay tres tipos de polimorfismo:

- Polimorfismo de sobrecarga
- Polimorfismo paramétrico (también llamado polimorfismo de plantillas)
- Polimorfismo de inclusión (también llamado redefinición o subtipado)

En este curso estudiaremos el polimorfismo de inclusión o subtipado.

#### POLIMORFISMO DE SUBTIPADO

La habilidad para redefinir un método en clases que se hereda de una clase base se llama **especialización**. Por lo tanto, se puede llamar un método de objeto sin tener que conocer su tipo intrínseco: esto es **polimorfismo de subtipado**. Permite no tomar en cuenta detalles de las clases especializadas de una familia de objetos, enmascarándolos con una interfaz común (siendo esta la clase básica).

Imagine un juego de ajedrez con los objetos *rey*, *reina*, *alfil*, *caballo*, *torre* y *peón*, cada uno heredando el objeto *pieza*.

El método *movimiento* podría, usando polimorfismo de subtipado, hacer el movimiento correspondiente de acuerdo a la clase objeto que se llama. Esto permite al programa realizar el *movimiento\_de\_pieza* sin tener que verse conectado con cada tipo de pieza en particular.

## 2.17 AMBIENTE

.Para que haya objetos que interactúen, se envíen mensajes y se conozcan entre sí; tienen que estar en algún lugar, tiene que haber algo que los contenga. A ese lugar lo llamamos ambiente de objetos.

Cualquier versión de Smalltalk, es un ambiente de objetos, en el cual viven los objetos con los que trabajamos y modelamos. Cuando creamos un objeto, se agrega al ambiente.

Ahora, para que un ambiente no crezca indefinidamente, tiene que haber alguna forma de sacar objetos. ¿Cómo hago para sacar un objeto del ambiente?. La respuesta para la mayoría de los lenguajes que soportan la orientación a objetos, incluido Smalltalk, es “el ambiente se arregla”. Dicho en forma sencilla, se



da cuenta cuando un objeto no tiene referencias, y lo saca. Esta característica es la que se conoce como garbage collector.

Hay una relación fuerte entre un ambiente de objetos y la memoria de la PC donde corre, no conviene pensar en el ambiente como “el formato que una aplicación con objetos le da a la memoria”. Algunos ambientes Smalltalk swapean a disco, existen ambientes distribuidos, hay técnicas para “deshidratar” objetos de un ambiente en una base de datos relacional y después volver a “hidratarlos” cuando hacen falta.

## 2.18 SOFTWARE

El software según el paradigma de objetos: Objetos que viven en un ambiente y que interactúan entre sí enviándose mensajes.

A partir de esta definición queda más claro que programar consiste en definir qué objetos necesito, definir los mensajes que va a entender y que van a formar su comportamiento, y escribir el código que soporta cada objeto definido, en donde se incluye enganchar mediante las variables de cada objeto las referencias necesarias para que cada objeto conozca a quienes debe enviarle mensajes.

Hacer software en gran escala hay mucho más de lo que me tengo que ocupar: manejo de memoria, base de datos, performance, etc. Hay proyectos de software nada pequeños que se desarrollaron o se desarrollan usando los conceptos del paradigma en una forma bastante razonable, y que se comportan bastante bien tanto en performance como en uso de recursos<sup>10</sup>. Cuando hubo problemas, más bien se debieron al uso incompleto o inadecuado de estos conceptos, y no al revés.

Hay muchísimo para decir sobre esto, por ahora destacamos dos factores:

- los conceptos de objetos, estos básicos que aparecieron hasta ahora: objeto, mensaje, polimorfismo, interacción, referencias; bien usados, permiten manejar modelos bastante complejos sin que se te vayan de las manos. Esto permite que haya menos necesidad de hacer chequeos redundantes, repetir cosas, etc., lo que en escala provoca que el software construido con objetos se hace más eficiente, y no menos.
- en un software bien construido con objetos los problemas de performance y uso de memoria terminan quedando bastante focalizados en lugares bien específicos; al trabajar sobre estos puntos, se logran con relativamente poco trabajo ganancias dramáticas (que un programa corra 10 veces más rápido después de un trabajo de performance no es algo raro).



### 3. IMPLEMENTACIÓN DEL POO CON SMALLTALK

El objetivo de la presente sección es introducirnos en:

- la sintaxis y semántica del lenguaje Smalltalk,
- la biblioteca Smalltalk: clases fundamentales de Smalltalk, tales como: cadenas, números, caracteres, colecciones y otras,
- el ambiente de programación de Smalltalk (playground, inspect it, transcript y system browser) para crear nuevas aplicaciones Smalltalk,
- los conceptos fundamentales del lenguaje: manejo de clases y objetos, mensajes, clases y herencia.

Para ello tomamos como libro cabecera el del autor Wilf Lalonde, Descubra Smalltalk'97. Este autor es uno de los pioneros en el uso productivo de Smalltalk y centra la atención de su libro en el dialecto de Smalltalk/V desarrollado para Microsoft Windows. En el desarrollo de la Asignatura y ejercitación práctica se utilizará el ambiente de Smalltalk - Pharo 5.0.

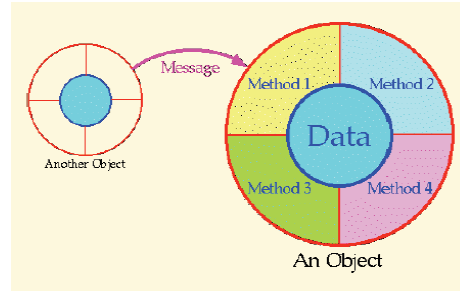
#### 3.1 INTRODUCCIÓN

Smalltalk es un lenguaje fuertemente adherido al paradigma orientado a objetos. Ha servido tanto como modelo de extensión de los lenguajes procedimentales tradicionales como de base para una nueva generación de lenguajes que admiten herencia. [LaLonde 97].

Smalltalk es un **lenguaje orientado a objetos puro**, todas las entidades que maneja son objetos. La programación en Smalltalk consiste en: [GIPSI 96].

- **Crear clases.**
- **Crear instancias.**
- **Especificar la secuencia de mensajes entre objetos.**

Toda la programación en Smalltalk se realiza mediante el envío de un mensaje a un objeto para invocar uno de sus métodos.



Smalltalk no es un lenguaje de investigación, existen distintos ejemplos de su uso en aplicaciones críticas, abarcando diversos dominios, tales como el proceso de transacciones con tarjetas de crédito y la supervisión de mercados internacionales de valores [Booch 97].

Smalltalk es un ambiente completo de desarrollo de programas. Éste integra de una manera consistente características tales como un editor, un compilador, un debugger, utilitarios de impresión, un sistema de ventanas y un manejador de código fuente. [GIPSI 96].

Smalltalk es un lenguaje altamente interactivo y se implementa como intérprete. Esto permite a los usuarios trabajar en muchas cosas diferentes a la vez dado que pueden suspender su actividad en una ventana y continuar otra actividad distinta en otra ventana simplemente moviendo el apuntador de una a otra ventana. Las ventanas pueden también moverse alrededor de la pantalla usando el ratón. [Cáceres González 04].

Las ventanas que utilizaremos en el presente material son:

- **Transcript:** ventana principal del sistema Smalltalk/V, que es un editor de texto utilizado para evaluar expresiones. El propio entorno también la utiliza para mostrar mensajes al programador.
- **Playground:** contienen un editor de texto y pueden utilizarse para evaluar expresiones.
- **Inspect it:** permiten observar la estructura interna de un objeto y modificar el valor de las mismas.
- **System Browser:** permite inspeccionar y editar la biblioteca de clases del ambiente.

## 3.2 ASPECTOS IMPORTANTES DE SMALLTALK [BUDD 91]

### 3.2.1 ASIGNACIÓN DINÁMICA DE MEMORIA

En Smalltalk solo disponemos de objetos, los cuales son manipulados a través de apuntadores/referencias.

Todas las referencias tienen un tamaño fijo constante y se le asigna espacio en la pila. Los objetos referenciados se almacenan en otro sector de la memoria, llamado heap (montículo), que no está sujeto al protocolo de asignación de la pila.





### 3.2.2 ASIGNACIÓN DE REFERENCIAS

Smalltalk usa semántica de referencias para asignación.

Una asignación a una referencia, cambia el valor de la referencia por el valor contenido en otra referencia. De esta manera, si tenemos dos variables no solo tendrán el mismo valor sino que referencian al mismo objeto. Por ejemplo, `x := y`.

### 3.2.3 ASIGNACIÓN DINÁMICA DE TIPOS

Para conocer la diferencia entre asignación estática y dinámica de tipos, es necesario que distingamos entre identificadores, tipos y valores. Un identificador o variable, es simplemente un nombre que utiliza el programador para manipular un espacio de memoria. Valor describe los contenidos actuales de la memoria del computador asociados a la variable. Dependiendo del lenguaje, un tipo se puede asociar con una variable o con un valor.

En los lenguajes con asignación de estática de tipos, como Java cuando trabaja con tipos primitivos de datos, los tipos se asocian con las variables al momento de ser declaradas. Por ejemplo (en Java): `int var`

En Smalltalk, las variables no se declaran con tipos específicos. De esta manera, no es posible asignar a una variable un valor ilegal.

En un lenguaje con asignación dinámica de tipos, cada valor debe llevar consigo una identificación que permite conocer la naturaleza exacta del valor. Esta identificación queda ligada al valor a través de la asignación. Con el paso del tiempo, a un identificador se le pueden pasar valores de muchos tipos diferentes.

Por ejemplo, en Smalltalk, podríamos tener:

```
| identificadorVariable |          Declaración del identificador de una variable sin indicar el tipo
identificadorVariable := 10.
identificadorVariable := "Hola".
```

### 3.2.4 OBJETOS POLIMÓRFICOS

Al ser un lenguaje con asignación dinámica todos los objetos son potencialmente polimórficos. Cualquier objeto puede tener valores de cualquier tipo.



### 3.3 SINTAXIS Y SEMÁNTICA DEL LENGUAJE SMALLTALK

#### 3.3.1 COMENTARIOS

Esta técnica permite comentar lo que estamos haciendo. Cualquier cosa entre comillas dobles es un comentario y Smalltalk lo ignora.

Sintaxis:

```
"Un comentario"
```

#### 3.3.2 OPERADOR DE ASIGNACIÓN (:=)

El operador := se denomina operador de ligadura o asignación porque asigna la variable de la izquierda al objeto calculado por la expresión de la derecha. Una vez efectuada la asignación, la evaluación de la variable devuelve el objeto asignado. [LaLonde 97].

Es un operador especial, no es un mensaje.

Por ejemplo:

```
temperaturaHoy := 22.
```

#### 3.3.3 LITERALES Y VARIABLES

En Smalltalk, los literales y variables son nombres que se dan a los objetos para luego hacer referencia a ellos.

Los **literales** [Cuenca] hacen referencia siempre al mismo objeto. Por ejemplo:

<code>\$a</code>	denota el carácter a.
<code>'hola'</code>	denota al string con las letras h, o, l y a.
<code>10</code>	denota al número 10.
<code>#hola</code>	denota al símbolo (string constante) con las letras h, o, l y a.



Las **variables**, en cambio, pueden hacer referencia a un objeto determinado en un instante de tiempo y, momentos después, a otro objeto completamente diferente. En un instante de tiempo, una variable solo puede hacer referencia a un objeto. [LaLonde 97].

El nombre de una variable debe comenzar con una letra y se encuentra compuesto por una secuencia de letras y dígitos.

En Smalltalk, podemos distinguir los siguientes tipos de variables: variables de instancia, variables globales, variables locales o temporales, argumentos de métodos, variables contenedoras y variables globales. Por ahora, describiremos las variables locales y globales.

### 3.3.3.1 VARIABLES LOCALES O TEMPORALES

Se utilizan para hacer referencia temporal a objetos que serán utilizados más adelante [Cuenca].

Para poder utilizar una variable local esta debe estar previamente declarada, para lo cual se listan sus nombres encerrados entre barras:

```
| variable1 variable2 variable3 |
```

El identificador de una variable local debe comenzar con minúscula.

Asignación de un objeto a una variable local:

```
variable1 := expresión.
```

Ejemplo:

```
| temperaturaAyer temperaturaHoy |  
temperaturaAyer := 10.  
temperaturaHoy := 22.
```

### 3.3.3.2 VARIABLES GLOBALES.

Son variables que pueden accederse desde cualquier parte del entorno y son persistentes entre diferentes sesiones de trabajo [Cuenca]. Si asignamos una variable global a un objeto, almacenamos la imagen y luego reiniciamos Smalltalk una semana más tarde, la variable estará aún asignada al objeto. [LaLonde 97].



Por ejemplo, las clases se identifican mediante variables globales, como Integer, Number, Magnitude y Object. Otra variable global es Transcript, que hace referencia a la ventana que encontramos en Smalltalk. [LaLonde 97].

El nombre de una variable global debe comenzar con mayúsculas [Cuenca].

Para **crear una variable global** se debe:

*Declaración de una variable global:*

```
Smalltalk at: #MiMejorAmiga put: ' Objeto de muestra '
```

Asignación de la variable global a un objeto:

```
MiMejorAmiga:= 'Gladys Garcia'
```

Las variables globales pueden recibir cambios:

```
MiMejorAmiga:= ' Vera '
```

**Eliminación de la declaración de una variable global:**

```
Smalltalk removeKey: #MiMejorAmiga
```

Otra opción para la creación de variables globales es la siguiente [Cuenca]:

- Escribir el nombre de la variable global en la ventana Transcript.
- Seleccionar y evaluar ese nombre.
- Confirmar la creación de la variable global.
- Asignar algún valor a la variable global (caso contrario, su valor será el objeto nil).
- Todas las variables globales se almacenan en un diccionario llamado Smalltalk (que es también una variable global).

### 3.3.4 PSEUDO-VARIABLES

Es un identificador que referencia a un objeto, no se les puede asignar un objeto y siempre referencian al mismo objeto.

#### 3.3.4.1 OBJETO ESPECIAL nil

Referencia a un objeto que representa la “nada” o “vacío”. Las variables que no han sido asignadas a un objeto referencian a nil. Es una instancia de la clase UndefinedObject. Si investigamos la clase UndefinedObject encontramos que es clase hija de la clase Object, no posee atributos y entiende los mensajes: **isNil** y **notNil**. [LaLonde 97].



```
nil isNil    cuyo resultado es: true
nil notNil   cuyo resultado es: false
25 isNil    cuyo resultado es: false
25 notNil   cuyo resultado es: true
```

### 3.3.4.2 true

Referencia a un objeto que representa el verdadero lógico [Gómez Deck 07].

### 3.3.4.3 false

Referencia a un objeto que representa el falso lógico [Gómez Deck 07].

### 3.3.4.4 self

Referencia al objeto receptor de un mensaje: objeto actual. Se utiliza cuando como parte de la implementación de un método es necesario invocar a un método de la misma clase. No se utiliza self para hacer referencia a un atributo de la clase.

### 3.3.4.5 super

Indica que debe ejecutarse el método de la clase base del objeto actual. [Gómez Deck 07].

### 3.3.4.6 thisContext

Referencia al objeto contexto-de-ejecución que tiene toda la información referente a la activación de un método [Gómez Deck 07].

## 3.3.5 SÍMBOLOS

Los símbolos (Symbol) son cadenas de caracteres (String) usadas como nombres de clases, métodos, etc. No pueden existir símbolos con el mismo conjunto de caracteres.

Sintaxis:

```
#unSímbolo.
```

Ejemplo:

```
#'un símbolo con espacios en blanco'.
```



## 3.4 IMPLEMENTACIÓN DE CONCEPTOS DEL POO EN SMALLTALK

### 3.4.1 CLASES

Una clase contiene elementos llamados miembros, que pueden ser atributos (variables de instancias), variables de clase y poolDictionaries.

La sintaxis para la definición de una clase en Smalltalk es:

```
Object subclass: #NombreClase
  instanceVariableNames: ' '
  classVariableNames: ' '
  poolDictionaries: ' '
```

Donde:

- **Object subclass:** indica la clase madre o base de la cual es clase hija la clase que se está creando.
- **#NombreClase:** indica el identificador que se le asigna a la clase. Este identificador debe comenzar con mayúscula y si se encuentra constituido por más de una palabra, cada palabra debe comenzar con mayúsculas. El nombre de la clase debe ser representativo de los objetos que permite instanciar.
- **Variables de instancia:** son los atributos del objeto y existen durante el tiempo de vida del objeto instancia de la clase [Goldberg/ Robson 83]. Los nombres de atributos sólo pueden ser referidos desde el interior de los métodos de su clase.
- **Variables de clase:** son compartidas por todas las instancias de una clase [Goldberg/ Robson 83]. En consecuencia, hay una sola instancia de cada una de estas variables. Por otra parte, la clase, sus rosubclases, las subclases de éstas, y así sucesivamente, y cada una de las instancias de ésta (y de las subclases) pueden acceder a los objetos de las variables de clase [Lalonde 97].
- **Pool dictionaries:** son variables contenedoras. Todas las variables mantenidas por los diccionarios comunes son accesibles desde las instancias y la clase que tiene el diccionario común en su definición. [Lalonde 97].

Si debiéramos desarrollar una aplicación bancaria, y se detecta que uno de los conceptos principales es el de CuentaBancaria, nos llevará a pensar en la necesidad de diseñar una clase que provea objetos para representar las cuentas del banco.

Si la información a mantener de cada cuenta fuera: número, titular, saldo y cantidad de cuentas que posee el banco, la definición de la clase en Smalltalk sería la siguiente:



```
Object subclass: #CuentaBancaria
    instanceVariableNames: ' numero titular saldo '
    classVariableNames: ' cantidadCuentas '
    poolDictionaries: ' '
```

## Continuación de Tipos de variables en Smalltalk

### 3.4.1.1 VARIABLES DE INSTANCIA

Representan los atributos de un objeto y se listan en la definición de la clase:

```
Object subclass: #CuentaBancaria
    instanceVariableNames: ' '
    classVariableNames: ' '
    package: 'EjemploVariables'
```

Una convención entre desarrolladores Smalltalk, indica que si el nombre de una variable se encuentra compuesto de más de una palabra (a excepción de la primera palabra), cada una de ellas debe comenzar con una letra mayúscula.

Por ejemplo, si tuviéramos una clase que representa una CuentaBancaria con los atributos número, titular y saldo:

```
Object subclass: #CuentaBancaria
    instanceVariableNames: 'numero titular saldo '
    classVariableNames: ' '
    package: 'EjemploVariables'
```

### 3.4.1.2 VARIABLES DE CLASE

Son compartidas por todas las instancias de una clase y se listan en la definición de la clase:

```
Object subclass: #CuentaBancaria
```



```
instanceVariableNames: ' '  
  
classVariableNames: ' '  
  
package: 'EjemploVariables'
```

El identificador de una variable de clase debe comenzar con una letra mayúscula.

Spongamos que la clase CuentaBancaria posee una tasa de interés común para todas las cuentas bancarias y además posee una variable de clase que determina la cantidad de instancias que se han creado a partir de la clase. La declaración sería la siguiente:

```
Object subclass: #CuentaBancaria  
  
instanceVariableNames: ' '  
  
classVariableNames: 'tasaInteres cantidadInstancias '  
  
package: 'EjemploVariables'
```

### 3.4.2 INSTANCIAS DE UNA CLASE

Para la creación de la instancia de una clase se debe enviar el mensaje **new** a la clase a instanciar.

Por ejemplo:

```
unObjeto:= Object new  
  
unaCuenta:= CuentaBancaria new  
  
otraCuenta:= CuentaBancaria new
```

Cuando se crea un objeto, todas sus variables de instancia (atributos) se inicializan en nil. Lo correcto es que inmediatamente después de instanciar un objeto, sus variables de instancia asuman valores iniciales válidos a través del método de inicialización.

No todos los objetos se pueden crear enviando el mensaje new, así por ejemplo, si le enviamos el mensaje new a la clase Fraction el sistema nos visualizará un mensaje de error. Para ello es necesario enviar el mensaje de división a un entero, como: 1/2.

```
| variable |  
  
variable := 1/2.
```





### 3.4.3 MÉTODOS

Un método en Smalltalk consta de dos partes:

- **Un patrón del mensaje** (también denominado cabecera del método) con el nombre del selector que identifica el método y los nombres de sus parámetros, denominados parámetros del método o colaboradores externos.
- **Cuerpo del método**, que se encuentra constituido por tres componentes:
  - Un comentario para describir la actuación de ese método concreto.
  - Una lista de variables temporales o locales que puedan ser utilizadas en las sentencias.
  - Las sentencias que responden al mensaje enviado.

Las variables temporales o locales creadas en los métodos, son creadas para una tarea específica y se encuentran disponibles solo mientras dura la actividad para la cual han sido creadas [Goldberg/ Robson 83].

Sintaxis

```
patronDelMensaje  
"comentario"  
|variablesTemporales|  
sentencias.
```

Por ejemplo, si la clase CuentaBancaria posee un método que le permite incrementar su saldo en un 20%, el método sería:

```
Depositar  
"Acredita en la cuenta un porcentaje fijo del 20%"  
saldo := saldo + (saldo * 0.20).
```

#### Argumentos de métodos

Permiten la participación de colaboradores externos en los métodos. Dichos parámetros/argumentos son necesarios para completar la funcionalidad del método<sup>i</sup>.

El identificador de una variable local debe comenzar con minúscula.

```
Sintaxis de un método sin argumentos
```



```
patronDelMensaje
“comentario”
|variablesTemporales|
sentencias.

Sintaxis de un método con argumentos

patronDelMensaje: colaboradorExterno1 <nombreArgumento: colaboradorexterno2>
“comentario”
|variablesTemporales|
sentencias.
```

Nota: Los caracteres <> indican opcionalidad (no forman parte de la sintaxis del método)

Ejemplo de un método con un parámetro:

```
depositar: importe
“Acredita el importe indicado en la cuenta”
saldo := saldo + importe.
```

Ejemplo de un método con más de un parámetro usando variables locales:

```
depositar: importe porcentaje: porc
“Acredita en la cuenta un importe y porcentaje pasados por parámetros”
|porcentaje|
porcentaje:= saldo * porc.
saldo := saldo + (importe + porcentaje).
```

### 3.4.3.1 MÉTODOS DE ACCESO Y MODIFICACIÓN

Para respetar la propiedad del POO: Encapsulamiento, la única forma de obtener información de un objeto es enviarle un mensaje solicitándole dicha información. En consecuencia, el objeto debe disponer de un método asociado a dicho mensaje que le permita devolver información sobre el objeto. Este método recibe el nombre de “**método de acceso**” (o accesor).



Por correspondencia, un método definido para cargar o modificar los atributos de un objeto, recibe el nombre de “**método de modificación**” (o modificador).

Por convenio, los programadores de Smalltalk, como nombres de métodos de acceso y modificación utilizan los mismos nombres que los de sus correspondientes nombres de atributos.

En consecuencia, si la clase CuentaBancaria posee el atributo numero, los nombres de los métodos de acceso y de modificación serán:

#### Método de Modificación

titular: unTitular

“Método modificador del atributo titular”

titular:= unTitular.

#### Método de Acceso

titular

“Método de acceso al atributo titular”

^titular.

“Retorno del objeto desde el método”

### 3.4.3.2 MÉTODOS DE INICIALIZACIÓN

#### 3.4.3.2.1 INICIALIZACIÓN DE VARIABLES DE INSTANCIA

Como se indicó anteriormente, cuando se crea un objeto, todas sus variables de instancia (atributos) se inicializan en nil. El método de inicialización se invoca inmediatamente después de creada una instancia y asegura que el estado inicial del objeto sea válido.

La convención, entre desarrolladores de Smalltalk, es denominar **initialize** a dicho método.

Si consideramos la clase CuentaBancaria donde sus atributos son: numero, titular y saldo su método de inicialización tendría, por ejemplo, la siguiente forma:

**initialize**

“Inicializa la instancia”

numero:= 0.



```
titular:= ' '.  
saldo:= 0.
```

Este mensaje se envía al objeto de la siguiente manera:

```
unaCuenta := CuentaBancaria new initialize.
```

### 3.4.3.2 INICIALIZACIÓN DE VARIABLES DE CLASE

Para la inicialización de variables de clase se debe disponer de un método de clase que tenga como objetivo inicializar las variables de clase. Se puede crear un solo método que inicialice todas las variables de clase o un método por variable de clase.

Este mensaje debe ser enviado a la clase antes de la creación de sus instancias, siempre que sus instancias así lo requieran.

Por ejemplo, si la clase Cuenta Bancaria tuviera la variable de clase *cantidadInstancias* y se desea inicializarla en cero:

#### **inicializarVariablesDeClase**

“Inicializa la variable de clase cantidadInstancias”

```
cantidadInstancias isNil
```

```
ifTrue: [ cantidadInstancias := 0].
```

Este mensaje se envía a la clase de la siguiente manera:

```
CuentaBancaria inicializarVariablesDeClase
```

### 3.4.4 PASO DE MENSAJES

“Cuando se envía un mensaje a un objeto, se busca en la clase receptora el método correspondiente. Si se encuentra, se hace una copia y se ejecuta la copia. Si no se encuentra, el proceso se repite en la superclase”. [LaLonde 94].

“La única forma de acceder a un objeto es enviándole un mensaje”. [Cuenca].

Ejemplos:

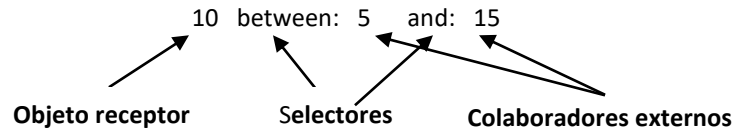
```
5 factorial      => 120      “envía el mensaje factorial al objeto 5”
```



8 + 2                            => 10                    “envía el mensaje +2 al objeto 8”

10 between: 5 and: 15       => true        “envía el mensaje between: 5 and: 15 al objeto 10”

**Componentes de un mensaje** [Cuenca]:



En los ejemplos anteriores:

**5 factorial**                    Receptor: **5**  
    Selector: **factorial**  
    Colaboradores externos: no tiene

**8 + 2**                            Receptor: **8**  
    Selector: **+**  
    Colaborador externo: **2**

**10 between:5 and:15** Receptor: **10**  
    Selector: **between: and:**  
    Colaboradores externos: **5 y 15**

Los mensajes en Smalltalk se clasifican en:

- **Mensajes Unarios**

Es similar a la llamada de una función sin parámetros. Se compone de un nombre de mensaje y un operando. El operando es el objeto al que se le envía el mensaje. Por ello, algunos lo interpretan como una función de un único parámetro.

Por ejemplo: **x sin** se puede interpretar como el **sin(x)**

Ejemplos de mensajes unarios:

```
5 factorial
Date tomorrow
```

- **Mensajes Binarios**

Son usados para especificar operaciones aritméticas, de comparación y lógicas.



Un mensaje binario se puede encontrar constituido por uno o dos caracteres y pueden contener una combinación de los siguientes caracteres especiales:

+ / \ \* - < > = @ % | & ? ! ,

Por ejemplo:

`a + b` retorna el resultado de sumar a y b

`a + b` se puede interpretar como que el mensaje + es enviado al objeto a con el parámetro b.

- **Mensajes de palabra clave**

Es equivalente a llamar a una función con uno o más parámetros.

Por ejemplo:

`a isKindOf: Integer`

Transcript show: 'El valor de x es:' , x

Los mensajes de palabra clave tienen variaciones algo más complejas.

Por ejemplo, podríamos preguntar:

`5 between: 1 and: 10` (el cual devuelve como resultado **true**)

Aquí, "between: 1 and: 10" es un mensaje compuesto por dos palabras claves, donde:

"between:" es una palabra clave y,

"and:" es otra palabra clave.

Por analogía con "1+2", podemos observar que 5, 1 y 10 son los **operandos** y "between: and:" es el **operador**. Obsérvese que los símbolos dos puntos forman parte del operador.

No hay límite en cuanto al número de operandos que puede tener un mensaje, pero entre cada operando deberá haber un palabra clave. Por lo tanto, lo siguiente no está permitido:

`5 between: 1 10` (**No permitido:** falta la palabra clave para el operando 10)

`5 between 1 and 10` (**No permitido:** faltan dos veces los dos puntos)

`5 between: 1 and 10` (**No permitido:** faltan dos puntos)



Expresiones como: **1 class**, **1 + 2**, y **1 between: 0 and: 10**, constan de un **receptor** (objeto receptor del mensaje): el objeto es **1** y en cada expresión se le envían, respectivamente, los mensajes:

- class
- +2
- between:0 and:10, respectivamente.

Todo mensaje en Smalltalk está compuesto de:

- Un **selector**: “class”, “+2” y “between:0 and:10”, respectivamente.
- y **parámetros**: que son los operandos del mensaje. Según nuestro ejemplo, respectivamente, ningún parámetro, un parámetro (2) y dos parámetros (0 y 10).

También nos podemos referir a los selectores como:

- Selectores unarios, por ejemplo: class
- Selectores binarios, por ejemplo: +
- Selectores de palabra clave, por ejemplo: between: and:

Ejemplo de envío de mensajes en Smalltalk

```
unaCuenta depositar: 3000- 500 porcentaje: 20+ 10
```

### 3.4.4.1 PASOS IMPLICADOS EN LA EJECUCIÓN DE UN MENSAJE

1. Evaluación de los parámetros del mensaje.

En el ejemplo:

“3000- 500” se transforma en “2500”

“20+ 10” se transforma en “30”

2. Envío del mensaje al receptor.  
En el ejemplo el mensaje “depositar: 3000- 500 porcentaje: 20+ 10” se envía al objeto “unaCuenta”.
3. Localización del método correspondiente a un objeto receptor particular y realización de una copia, el método se copia.
4. Emparejamiento del método con el patrón del mensaje perteneciente al método copia, de la forma:
  - a. Emparejamiento del objeto receptor codificado en la expresión con la variable especial self - también denominada pseudovariabla- que identifica al objeto receptor, al tiempo que se ejecuta el método copia.



En el ejemplo, se empareja “unaCuenta” con “self”.

- b. Emparejamiento del selector codificado en la expresión con la cabecera del método copia.  
En el ejemplo, la cláusula “depositar: porcentaje:” de la expresión se empareja con la cláusula “depositar: porcentaje:” que se encuentra en la cabecera del método copia.
  - c. Emparejamiento de los parámetros de la expresión, los parámetros del mensaje con sus correspondientes parámetros que se encuentran en la cabecera del método copia.  
En el ejemplo, el entero “2500” se empareja con el parámetro del método “importe” y el entero “30” se empareja con el parámetro del método “porcentaje” del método copia.
5. Ejecución del cuerpo del método copia hasta la localización de una expresión de respuesta. Si el método no posee una expresión de respuesta, se ejecutará una expresión de respuesta por omisión ^self.
  6. Devolución del objeto calculado en la expresión de respuesta al emisor del mensaje, para su uso en posteriores cálculos.

#### 3.4.4.2 ENVÍO DE MÁS DE UN MENSAJE A UN OBJETO

“Un punto y coma entre dos mensajes indica que el mensaje que se encuentra inmediatamente a continuación de éste se va a enviar al mismo objeto receptor al que se ha enviado el anterior”

Por ejemplo, si la clase Cuenta Bancaria contara con los métodos depositar y extraer:

```
unaCuenta    depositar:3000;  
              extraer:500;  
              depositar:400 porcentaje:20.
```

#### 3.4.4.3 REGLAS DE PRIORIDAD DE MENSAJES

Dados dos selectores consecutivos, el selector de la izquierda se evalúa antes si ambos tienen la misma prioridad. En caso contrario, se evalúa antes el selector con mayor prioridad.

- Selectores Unitarios: máxima prioridad.
- Selectores Binarios: la siguiente prioridad más alta.
- Selectores de Palabra Clave: mínima prioridad.

**Ejemplo 1 con selector binario:** La prioridad de los selectores unarios se realiza estrictamente de izquierda a derecha.

1 + 2 (devuelve como resultado el objeto 3)

**Ejemplo 2 con selector unitario:** La prioridad de los selectores unarios se realiza estrictamente de izquierda a derecha.

1 negated (devuelve como resultado el objeto -1)





-1 negated (devuelve como resultado el objeto 1)  
0 factorial (devuelve como resultado el objeto 1)  
1 factorial (devuelve como resultado el objeto 1)  
1 negated negated (devuelve como resultado el objeto 1)

Evaluación de esta expresión paso a paso:

1 negated negated  
-1 negated  
1

**Ejemplo 3** con mezcla de selectores unitarios y binarios: Los selectores unitarios tienen asignada una prioridad más alta que los selectores binarios.

1 negated + 2 (devuelve como resultado el objeto 1)

Evaluación paso a paso:

1 negated + 2  
-1 + 2  
1  
1 + 2 negated (devuelve como resultado el objeto -1)

Evaluación paso a paso:

1 + 2 negated  
-1 + 2  
1

**Ejemplo 4** con mezcla de selectores unitarios, binarios y de palabra clave:

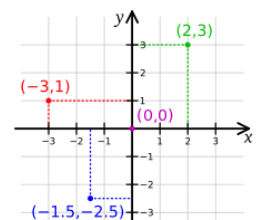
25 between: 10 + 3 factorial and: 5 factorial + 3 (devuelve como resultado el objeto true)

Evaluación paso a paso:

25 between: 10 + 3 factorial and: 5 factorial + 3  
25 between: 10 + 6 and: 5 factorial + 3 (porque se evalúa de izquierda a derecha)  
25 between: 16 and: 5 factorial + 3  
25 between: 16 and: 120 + 3  
25 between: 16 and: 123

**True**

### 3.4.5 RESOLUCIÓN DE UN CASO DE ESTUDIO





Desarrollar un programa que permita representar un **punto** en el plano, y calcule la distancia que existe entre el origen de las coordenadas (0,0) y un punto (x,y) determinado.

```
Object subclass: #Punto "Autor: Ing. Karina Ligorria"
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'CoordenadasCartesianas'

initialize
  x:=0.
  y:=0.

x
  "Answer the value of x"
  ^ x

x: anObject
  "Set the value of x"
  x := anObject

y
  "Answer the value of y"
  ^ y

y: anObject
  "Set the value of y"
  y := anObject

asString
  "Retorna los valores de x e y, en forma de coordenadas cartesianas"
  ^ '(' , x, ', ', y, ')'.

calcularDistancia
  "La distancia entre el origen y la coordenada x,y es igual a la
  hipotenusa: raiz cuadrada de la suma del cuadrado de x y cuadrado de y"
  ^((self x squared) + (self y squared) ) sqrt.
```

En Playground:

```
|p|
p:= Punto new initialize.
```



```
p x:10;
```

```
y:5.
```

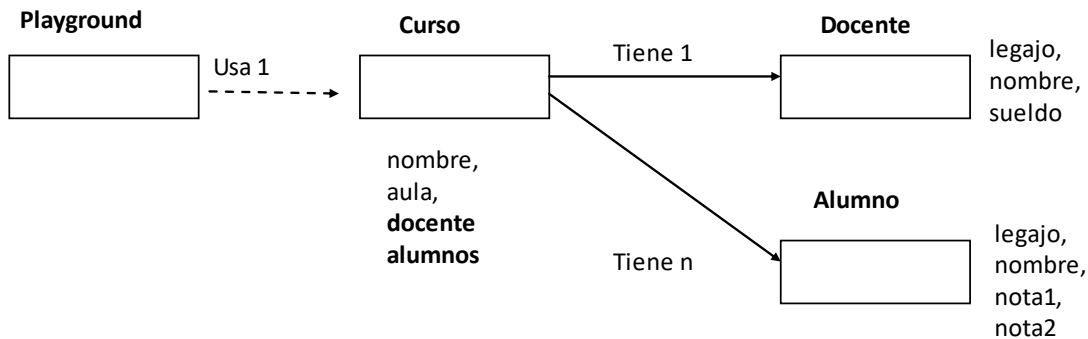
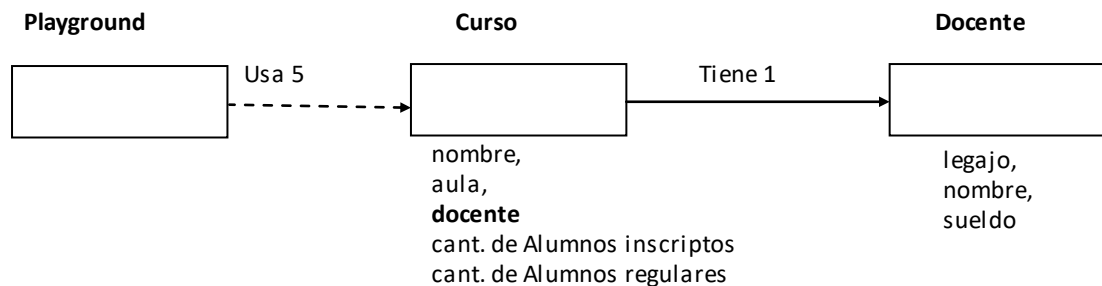
```
Transcript show: 'Distancia desde la coordenada (0,0) a ', p asString, ' es: ',  

p calcularDistancia asString.
```

### 3.5 IMPLEMENTACIÓN DE COMPOSICIÓN EN SMALLTALK

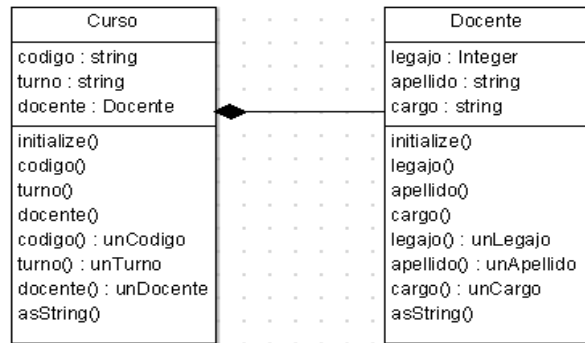
Como estudiamos en la primera sección la relación de composición se da cuando una clase posee objetos de otras clases (relación *tiene un*). Se puede reutilizar los atributos y métodos de otras clases, a través de la invocación de los mensajes correspondientes.

Algunos ejemplos gráficos:



#### 3.5.1 RESOLUCIÓN DE UN CASO DE ESTUDIO

A continuación plantearemos un ejemplo, utilizando el siguiente modelo:



### 1. Definición de la clase Docente

```
Object subclass: #Docente "Autor: Ing. Analía Guzman"
  instanceVariableNames: 'legajo apellido cargo'
  classVariableNames: ''
  package: 'ComposicionPPR'

initialize
  legajo := 0.
  apellido := ''.
  cargo := ''.

legajo
  ^legajo

legajo: anObject
  legajo := anObject

apellido
  ^apellido

apellido: anObject
  apellido := anObject

cargo
  ^cargo

cargo: anObject
  cargo := anObject

asString
  |res|

  res := String cr, 'Legajo: ', self legajo asString,
    ' - Apellido: ', self apellido asString,
    ' - Cargo: ', self cargo asString.

  ^ res
```



## 2. Definición de la clase Curso

```
Object subclass: #Curso "Autor: Ing. Analía Guzman"
  instanceVariableNames: 'codigo turno docente'
  classVariableNames: ''
  package: 'ComposicionPPR'

initialize
  codigo := ''.
  turno := ''.
  docente := nil.

codigo
  ^codigo

codigo: anObject
  codigo := anObject

turno
  ^turno

turno: anObject
  turno := anObject

docente
  ^docente.

docente: anObject
  docente := anObject.

asString
  |res|

  res := 'Codigo del curso: ', self codigo asString,
        'Turno: ', self turno asString,
        'Docente: ', self docente asString.
  ^ res
```

Se invoca el método **asString** de la clase Docente.

## 3. Invocación de objetos en Playground

Para mostrar la invocación de objetos en cascada, vamos a implementar en **Playground** como se debe hacer para mostrar el nombre del docente, por ejemplo.



En **Playground** para poder mostrar en Transcript el nombre del docente, se deberá seguir el proceso de invocación de mensajes, correspondiente a la relación definida, se deberá pedirle al curso el nombre del docente y este le pedirá al docente su nombre, vemos a continuación el ejemplo en **Playground**:

```
|unCurso unDocente|  
  
unDocente := Docente new initialize.  
  
unDocente legajo: 1;  
           apellido: 'Lopez';  
           cargo: 'Adjunto'.  
  
unCurso := Curso new initialize.  
  
unCurso codigo: '2K01';  
           turno: 'Noche';  
           docente: unDocente.           "objeto de tipo Docente"  
  
Transcript show: String cr, 'Los datos del curso son: ', unCurso asString.  
  
Transcript show: String cr, 'El nombre del docente del curso: ', unCurso codigo  
asString, ' es: ', (unCurso docente) nombre asString.
```



### 3.6 IMPLEMENTACIÓN DE HERENCIA EN SMALLTALK

Como estudiamos en la primera sección, La Herencia es la propiedad que permite a los objetos construirse a partir de otros objetos.

Una subclase o clase hija o clase derivada, es un nuevo tipo de objetos definido por el usuario que tiene la propiedad de heredar los atributos y métodos de una clase definida previamente, denominada superclase o clase madre o clase base [Ceballos, 2003].

Cuando se define una nueva clase, las variables de instancia proporcionadas por la nueva clase se añaden a las variables de instancia que se obtienen automáticamente de las clases que se encuentran más arriba en la jerarquía. La subclase puede añadir más variables de instancia pero no puede eliminar ninguno de los atributos heredados.



Una subclase puede, a su vez, ser una superclase de otra clase dando lugar a una jerarquía de clases. Por lo tanto, una clase puede ser una superclase directa de una subclase, si figura explícitamente en la definición de la subclase, o una superclase indirecta si está varios niveles arriba en la jerarquía de clases, y por lo tanto no figura explícitamente en el encabezado de la subclase [Ceballos, 2003].

Smalltalk admite herencia simple, es decir, que una clase hija solo puede tener una clase madre. Sin embargo, una clase madre puede tener tantas clases hijas como se requieran.

Cuando se crea una instancia de una subclase su estructura de atributos está conformada por los atributos propios más los atributos heredados, no teniendo acceso directo a los atributos heredados por respetar la propiedad de encapsulamiento de las superclases. En este caso, se deberá utilizar el método de acceso que corresponda de la superclase.

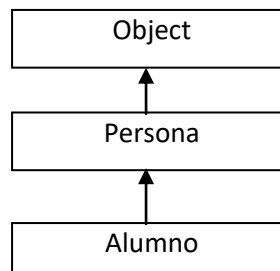
### 3.6.1.1 DEFINIR UNA SUBCLASE

La sintaxis para declarar una subclase es la siguiente:

```
ClaseBase subclass: #ClaseHija
instanceVariableNames: ''
classVariableNames: ''
package: 'HerenciaPPR'
```

En Smalltalk, los inspectores indican de dónde proceden las variables de instancia heredadas.

A continuación se presenta un ejemplo de cómo crear una jerarquía simple:



Se declara la clase base Persona, utilizando como clase base a la clase Object por defecto:

```
Object subclass: #Persona
instanceVariableNames: 'dni nombre telefono'
classVariableNames: ''
package: 'HerenciaPPR'
```

Se declara la clase derivada Alumno, utilizando como base a la clase Persona:

```
Persona subclass: #Alumno
```



```
instanceVariableNames: 'legajo promedio'  
classVariableNames: ''  
package: 'HerenciaPPR'
```

### 3.6.1.2 USO DE SUPER

El uso de super provoca que la búsqueda del método comience en la superclase del objeto receptor.

Cuando super es encontrado en la ejecución de un programa, Smalltalk busca el método en la superclase del receptor.

Por ejemplo:

```
super unMensaje. 'invoca a un mensaje de la clase base o superclase'  
self unMensaje. 'invoca a un mensaje de la propia clase'
```

### 3.6.1.3 HERENCIA DE VARIABLES

La jerarquía del lenguaje Smalltalk, ha sido diseñada para que las subclases hereden las variables de sus superclases. Las subclases también pueden poseer variables propias.

Las variables de instancia están definidas en la definición de la clase. Los datos de la instancia se mantienen en un área de datos creada por Smalltalk.

**Variables de Instancia:** Cada subclase tiene su propia copia (estado) de variables, tanto las propias como las heredadas.

Por ejemplo: el objeto alumno tiene las variables de instancia: dni, nombre, telefono, legajo y promedio. Estas variables incluyen las variables de instancia heredadas de Persona.





**Variables de Clase:** Una variable de clase es una variable que es compartida por todos los objetos de la clase.

Sólo existe una copia de la variable de clase en la jerarquía local de clases. Todos los objetos de la clase referencian a esta única copia de la variable.

Las variables de clase también permiten compartir la información a través de todas las subclases de la clase en que fueron declaradas.

### 3.6.1.4 INICIALIZACIÓN DE ATRIBUTOS EN UNA CLASE HIJA

Por defecto las variables de instancia se inicializan con el valor nil. A través del método `initialize` se inicializan las variables de instancia a un valor por defecto definido por el usuario.

Para inicializar los atributos propios y heredados de una clase hija, en el método de inicialización de la clase hija se deben inicializar en primer lugar los atributos heredados llamando al método **`initialize`** de la clase base y seguidamente sus propias variables de instancia.

Sintaxis del método de inicialización en la clase derivada:

#### **Initialize**

```
“Inicializar atributos heredados de su clase base”  
super initialize.  
“Inicializar atributos propios”  
atributoPropio1:= 0.  
atributoPropio2:= ' '.
```



### 3.6.1.5 HERENCIA DE MÉTODOS

La herencia de métodos es útil para permitir a una clase modificar su comportamiento respecto de su superclase. Esto puede ser hecho agregando nuevos métodos, o redefiniendo los métodos heredados.

- **Agregación de Métodos:**

Se puede agregar métodos de manera muy simple, incorporándolos, ya sean de instancia o de clase, en la definición de la clase. Todo objeto de la subclase soporta los métodos de su superclase, más los nuevos métodos.

- **Redefinición de Métodos:**

Redefinir un método heredado significa volverlo a escribir en la subclase con el mismo nombre y los mismos colaboradores externos que tenía en la superclase y su implementación adaptada a las necesidades de la subclase.

Una subclase puede redefinir (volver a definir) algún método existente en la superclase, con el objeto de proveer una implementación diferente, por ejemplo: método `asString`.

Para redefinir un método en la subclase, se tiene que declarar con la misma signatura (nombre y colaboradores externos).

En la invocación de métodos, si existen dos métodos con la misma signatura, uno en la subclase y otro en la superclase, se ejecutará siempre el de la subclase.

Usando el ejemplo de la clase `Alumno`, al definir el método `asString` se lo está redefiniendo, ya que tanto la clase base `Persona`, como la clase `Object` ya las tienen implementadas.

```
asString
|res|
res := super asString,
      ', Legajo: ', self legajo asString,
      ', Promedio: ', self promedio asString.
^res.
```

### 3.6.1.6 ENLACE DINÁMICO



Cuando un objeto recibe un mensaje se busca el método asociado al mensaje en la clase del objeto receptor del mensaje, si no se encuentra en la clase de dicho objeto se continua la búsqueda en la clase inmediata superior jerárquicamente y, así sucesivamente, hasta ejecutar el método correspondiente. En el caso de no existir dicho método, el sistema emite un error. Este procedimiento recibe el nombre de **enlace dinámico**.

### 3.6.1.7 CLASES ABSTRACTAS

Son clases genéricas que sirven para agrupar clases del mismo género. Definen comportamientos que se implementarán en las subclasses.

Definen un protocolo común para una jerarquía de clases que es independiente de las opciones de representación.

No se pueden instanciar, es decir no se pueden crear objetos a partir de ellas, ya que representan conceptos tan generales, con comportamientos que no siempre se pueden definir cómo será la implementación de los mismos.

Los métodos abstractos, son métodos que no poseen implementación, se definen en las clases abstractas.

Smalltalk no tiene una sintaxis dedicada para especificar que un método o una clase es abstracta. Por convención, el cuerpo de un método abstracto consiste en la expresión: `self subclassResponsibility`.

Este es conocido como un "método de marcador", e indica que subclasses tienen la responsabilidad de definir una versión concreta del método.

Los métodos `subclassResponsibility` siempre deben ser redefinidos, y por lo tanto no deben ser ejecutados. Si se olvida de redefinir uno, y se ejecuta, se produce una excepción.

Una clase se considera abstracta si uno de sus métodos es abstracto. Nada en realidad le impide crear una instancia de una clase abstracta; todo funcionará hasta que se invoca un método abstracto.

Por ejemplo, si queremos definir un método abstracto, llamado `size`, en una clase abstracta, definimos:

```
size
  ^ self subclassResponsibility.
```

También podemos definirlo sin cuerpo, pero no producirá una excepción si se invoca equivocadamente, por ejemplo:

size

“sin implementación”

Para finalizar, en alguna clase derivada se deberá redefinir e implementar el método, por ejemplo:

size

^ size

### 3.6.2 RESOLUCIÓN DE UN CASO DE ESTUDIO

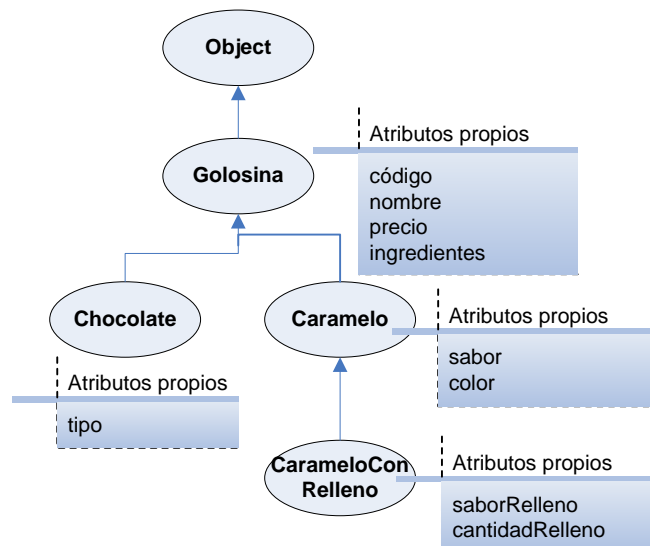
Una fábrica de dulces desea implementar un sistema de producción donde se manejen los datos de las golosinas específicamente de sus chocolates, caramelos sin relleno y con relleno.

De los chocolates se disponen los siguientes datos: código (entero), nombre, precio, ingredientes (cadena de caracteres) y tipo de chocolate (negro o blanco).

De los caramelos sin relleno: código (entero), nombre, precio, ingredientes (cadena de caracteres), sabor y color.

De los caramelos con relleno: código (entero), nombre, precio, ingredientes (cadena de caracteres), sabor del relleno y cantidad de relleno.

A partir de este relevamiento llegamos a la siguiente jerarquía de clases:





Codificación en Smalltalk (se generan los métodos de acceso y modificación automáticamente desde Pharo 5.0.).

```
Object subclass: #Golosina "Autor: Ing. Karina Ligorria"
  instanceVariableNames: 'codigo nombre precio ingredientes'
  classVariableNames: ''
  package: 'FabricaGolosinas'

initialize
  self codigo: 0. "El cero representa que no tiene codigo asignado aun"
  self nombre: 'Sin nombre asignado'.
  self precio: 0.0.
  self ingredientes: 'Sin ingredientes establecidos'.

asString
  ^String cr, 'Codigo: ', self código asString, ' Nombre:', self nombre
  asString, ' Precio: $', self precio asString, ' Ingredientes: ', self
  ingredientes asString.
```

```
Golosina subclass: #Chocolate "Autor: Ing. Karina Ligorria"
  instanceVariableNames: 'tipo'
  classVariableNames: ''
  package: 'FabricaGolosinas'

initialize
  super initialize.
  self tipo: 'Negro'. "Por defecto el chocolate es negro"

asString "Redefinición de métodos"
  ^String cr, 'CHOCOLATE: ', super asString, 'Tipo de chocolate: ' , self
  tipo asString.
```

```
Golosina subclass: #Caramelo "Autor: Ing. Karina Ligorria"
  instanceVariableNames: 'sabor color'
  classVariableNames: ''
  package: 'FabricaGolosinas'

initialize
  super initialize.
  self sabor: 'No asigando'.
  self color: 'No asignado'.

asString "Redefinición de métodos"
```



```
^String cr, 'CAMELO: ', super asString, ' Sabor: ' , self sabor  
asString, ' Color: ', self color asString.
```

```
Caramelo subclass: #CarameloConRelleno "Autor: Ing. Karina Ligorria"  
instanceVariableNames: 'saborRelleno cantidadRelleno'  
classVariableNames: ''  
package: 'FabricaGolosinas'  
  
initialize  
super initialize.  
self saborRelleno: 'No asignado'.  
self cantidadRelleno: 0.  
  
asString "Redefinición de métodos"  
  
^String cr, super asString, ' Sabor del relleno: ' , ' RELLENO: ', self  
saborRelleno asString, ' Cantidad de relleno: ' , self cantidadRelleno asString.
```

En **Playground**:

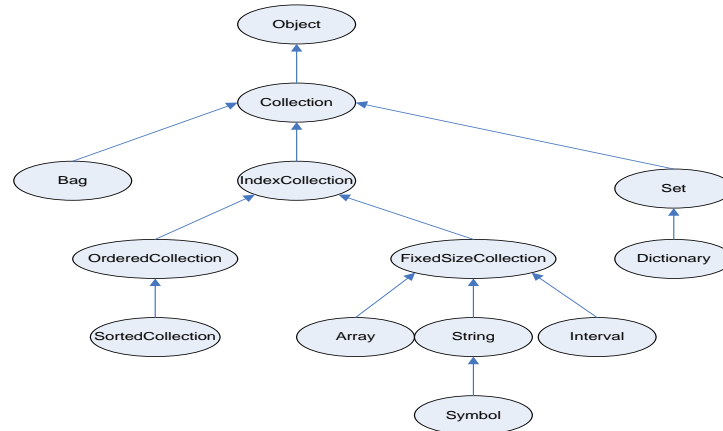
```
g:= CarameloConRelleno new initialize.  
  
g codigo: 1; "Método heredado de la clase Golosina"  
nombre: 'Gomita'; "Método heredado de la clase Golosina"  
precio: 1; "Método heredado de la clase Golosina"  
ingredientes: 'azúcar - agua - gelatina sin sabor - gelatina del sabor  
que se quiera'; "Metodo heredado de la clase Golosina"  
  
sabor: 'chocolate'; "Método heredado de la clase Caramelo"  
color: 'marron'; "Método heredado de la clase Caramelo"  
  
saborRelleno: 'Coco'; "Método propio"  
cantidadRelleno: 2. "Método propio"  
  
Transcript show: g asString. "Método redefinido en la clase CarameloRelleno"
```

### 3.7 IMPLEMENTACIÓN DE POLIMORFISMO EN SMALLTALK

- **Polimorfismo de subtipado**

En SmallTalk, el manejo de colecciones es polimórfica, posee una jerarquía y operaciones genéricas asociadas, pueden almacenar objetos de cualquier clase y presentan un protocolo unificado, por lo que todas las diferentes formas de colecciones provistas por el entorno responden a un mismo conjunto de mensajes básicos, lo que facilita su aprendizaje y el cambio de un tipo de colección a otra.

El siguiente gráfico muestra la jerarquía de colecciones de Smalltalk/V [Cuenca]:



### 3.7.1 RESOLUCIÓN DE UN CASO DE ESTUDIO

#### Enunciado

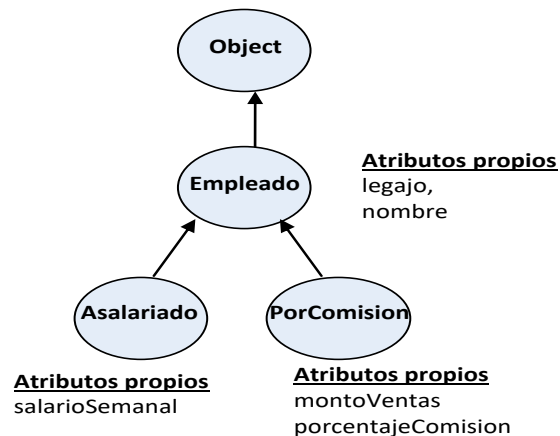
Para administrar los empleados de una empresa, se debe desarrollar un programa orientado a objetos que pueda manejar mediante clases las siguientes entidades descubiertas durante la etapa de análisis del sistema:

**Empleado:** que tiene los siguientes datos: legajo y nombre.

**Empleado Asalariado:** (es un Empleado) que tiene como atributo el monto del salario semanal. El cálculo del ingreso de este empleado se establece por el monto del salario semanal.

**Empleado por Comisión:** (es un Empleado) que tiene como datos: el monto de ventas por semana, y el porcentaje de comisión por ventas. El ingreso para estos empleados es el montoComision que se calcula como el porcentajeComision\*montoVentas/100.

Se acompaña un diagrama orientativo:





Se pide:

1. Implementar las clases Empleado, Asalariado y PorComision que tengan los métodos para inicializar, los métodos de acceso y modificadores, el método asString, que retorna una cadena con los atributos y el método montoIngreso, que retorna el ingreso que le corresponde según lo enunciado anteriormente.
2. Realizar las pruebas de funcionamiento instanciando algunos objetos e invocando sus métodos.

### Resolución

1. Codificación en Smalltalk (se generan los métodos de acceso y modificación automáticamente desde Pharo 5.0.).

```
Object subclass: #Empleado "Autor: Ing. Analía Guzman"
  instanceVariableNames: 'legajo nombre'
  classVariableNames: ''
  package: 'practicoPolimorfismo'

initialize
  legajo := 0.
  nombre := ''.

legajo
  ^ legajo

legajo: anObject
  legajo := anObject

nombre
  ^ nombre

nombre: anObject
  nombre := anObject

asString
  "Método redefinido"

  |res|
  res := String cr, 'El Empleado: ', self legajo asString, ', con nombre: ',
self nombre asString.
  ^ res.

montoIngreso
  "Método abstracto"
  ^self subclassResponsibility.
```





```
Empleado subclass: #Asalariado "Autor: Ing. Analía Guzman"  
  instanceVariableNames: 'salarioSemanal'  
  classVariableNames: ''  
  package: 'practicoPolimorfismo'  
  
initialize  
  super initialize.  
  salarioSemanal := 0.0.  
  
salarioSemanal  
  ^ salarioSemanal  
  
salarioSemanal: anObject  
  salarioSemanal := anObject  
  
asString  
  "Método redefinido"  
  
  |res|  
  res := super asString, ' tiene un salario semanal de ', self salarioSemanal  
  asString.  
  ^ res.  
  
montoIngreso  
  "Método redefinido"  
  
  ^ self salarioSemanal.
```

```
Empleado subclass: #PorComision "Autor: Ing. Analía Guzman"  
  instanceVariableNames: 'montoVenta porcentajeComision'  
  classVariableNames: ''  
  package: 'practicoPolimorfismo'  
  
initialize  
  super initialize.  
  montoVenta := 0.0  
  porcentajeComision := 0.0.  
  
montoVenta  
  ^ montoVenta  
  
montoVenta: anObject  
  montoVenta := anObject  
  
porcentajeComision  
  ^ porcentajeComision
```



```
porcentajeComision: anObject
  porcentajeComision := anObject

asString
  "Método redefinido"
  |res|
  res := super asString, 'tiene una comision de ', self porcentajeComision
  asString, ' y genero un monto de ', self montoVenta asString.
  ^ res.

montoIngreso
  "Método redefinido"
  |montoComision |

  "calcula el monto de la comision"
  montoComision := (porcentajeComision * montoVenta) asFloat /100.0.

  "retorna el monto de comision"
  ^ montoComision.
```

2. Crear objetos e invocar sus métodos en Playground:

```
|empleadoAsalariado empleadoComision|

empleadoAsalariado := Asalariado new initialize.

empleadoAsalariado legajo: 1;           "Método heredado de la clase Empleado"
  nombre: 'Juan Perez';                 "Método heredado de la clase Empleado"
  salarioSemanal: 5000.                 "Método propio de la clase Asalariado"

Transcript show: String cr, empleadoAsalariado asString. "Método polimórfico,
redefinido en la clase Asalariado "

Transcript show: String cr, 'El monto de Ingreso es: ', empleadoAsalariado
montoIngreso asString. "Método polimórfico, redefinido en la clase Asalariado "

empleadoComision := PorComision new initialize.

empleadoComision legajo: 2;           "Método heredado de la clase Empleado"
  nombre: 'Maria Juarez';             "Método heredado de la clase Empleado"
  montoVenta: 2000;                   "Método propio de la clase Comision"
  pocentajeComision: 20.              "Método propio de la clase Comision"

Transcript show: String cr, empleadoComision asString. "Método polimórfico,
redefinido en la clase Comision "

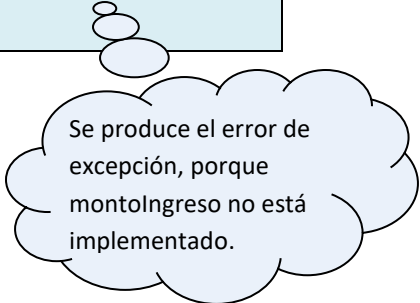
Transcript show: String cr, 'El monto de Ingreso es: ', empleadoComision
montoIngreso asString. "Método polimórfico, redefinido en la clase Comision"
```



**Atención:** Lo que no debemos hacer en Playground

Como la clase empleado tiene un método abstracto, si bien la misma se puede instanciar en Smalltalk, no se podrá invocar al método montoIngreso, ya que producirá una excepción:

```
|empleado|  
  
empleado := Empleado new initialize.  
  
empleado legajo: 1; nombre: 'Juan Perez';  
  
Transcript show: String cr, empleado asString.  
  
Transcript show: String cr, 'El monto de Ingreso es: ', empleado montoIngreso  
asString.
```



Se produce el error de excepción, porque montoIngreso no está implementado.

### 3.8 BIBLIOTECA DE CLASES DE SMALLTALK

En Smalltalk, todas las clases deben estar contenidas en una sola jerarquía de herencia donde la clase raíz de dicha jerarquía es la clase Object. La ventaja de este enfoque es que la funcionalidad proporcionada en la clase Object es heredada por todos los objetos.

Al ser un lenguaje que usa asignación dinámica de tipos, los objetos se caracterizan por los mensajes que entienden. Si dos objetos entienden el mismo conjunto de mensajes y reaccionan de la misma manera, para todo propósito práctico son indistintos, independientemente de las relaciones de sus clases respectivas. Por este motivo, es útil hacer que todos los objetos hereden gran parte de su comportamiento de una clase base común.

En un lenguaje así, toda clase definida por el usuario debe ser una subclase de alguna clase ya existente.

#### 3.8.1 CLASE OBJECT

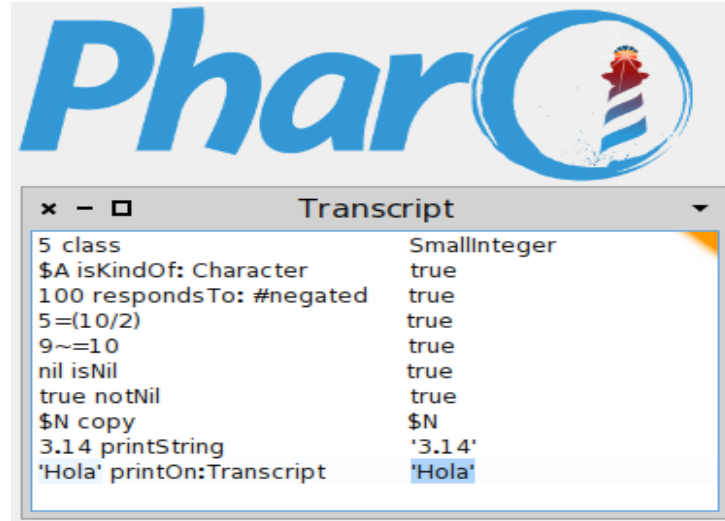
Todas las clases de Smalltalk son subclases (directa o indirectamente) de la clase Object. Es la única clase que no tiene superclase y define el comportamiento que es común a todos los objetos. [Cuenca].



La siguiente es una lista de los principales mensajes que responden todos los objetos:

<b>Mensaje</b>		<b>Ejemplos</b>
<b>Class</b>	Devuelve la clase del receptor	5 class
<b>isKindOf: UnaClase</b>	Devuelve <b>true</b> si el receptor es instancias de la clase indicada o de alguna de sus subclases	\$A isKindOf: Character
<b>respondsTo: unSelector</b>	Responde true si el receptor puede responder un mensaje con el selector que se indica	100 respondsTo:#negated
<b>= otroObjeto</b>	Devuelve true si el receptor es igual a otroObjeto	3 = (6/2)
<b>~= otroObjeto</b>	Devuelve true si el receptor es distinto a otroObjeto	5 ~= 10
<b>isNil</b>	Responde true si el receptor es el objeto indefinido	nil isNil
<b>notNil</b>	Responde true si el receptor no es el objeto indefinido	true notNil
<b>Copy</b>	Retorna un duplicado del receptor	\$A copy
<b>printString</b>	Retorna un string descriptivo del receptor	3.14 printString
<b>printOn: Transcript</b>	Muestra el receptor en la ventana Transcript	'hola' printOn: Transcript

Resultados obtenidos de la ejecución de los ejemplos:



### 3.8.2 OBJETO INDEFINIDO

Es la representación de la nada. El objeto indefinido se llama nil y es instancia de la clase **UndefinedObject**.

Todas las variables (globales o locales) tienen el valor nil hasta que se les asigna algún otro objeto.

nil isNil	cuyo resultado es: true
nil notNil	cuyo resultado es: false
25 isNil	cuyo resultado es: false
25 notNil	cuyo resultado es: true

### 3.8.3 MAGNITUDES

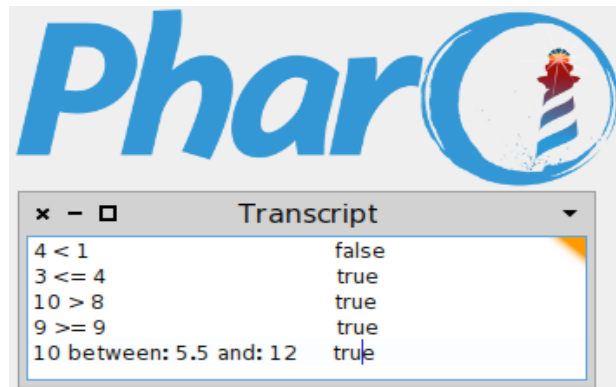
La librería de Smalltalk ofrece objetos muy diferentes, tales como números, fechas y horas; todos esos objetos (totalmente disímiles en su significado y utilización) tienen, sin embargo, algo en común: pueden ser ordenados de mayor a menor y compararse entre sí. Todos ellos son magnitudes. [Cuenca].

La clase Magnitude es una superclase abstracta que define lo que tienen en común tanto los números como las fechas y las horas; dicho de otra forma, define qué comportamientos debe tener un objeto para ser considerado como una magnitud lineal.

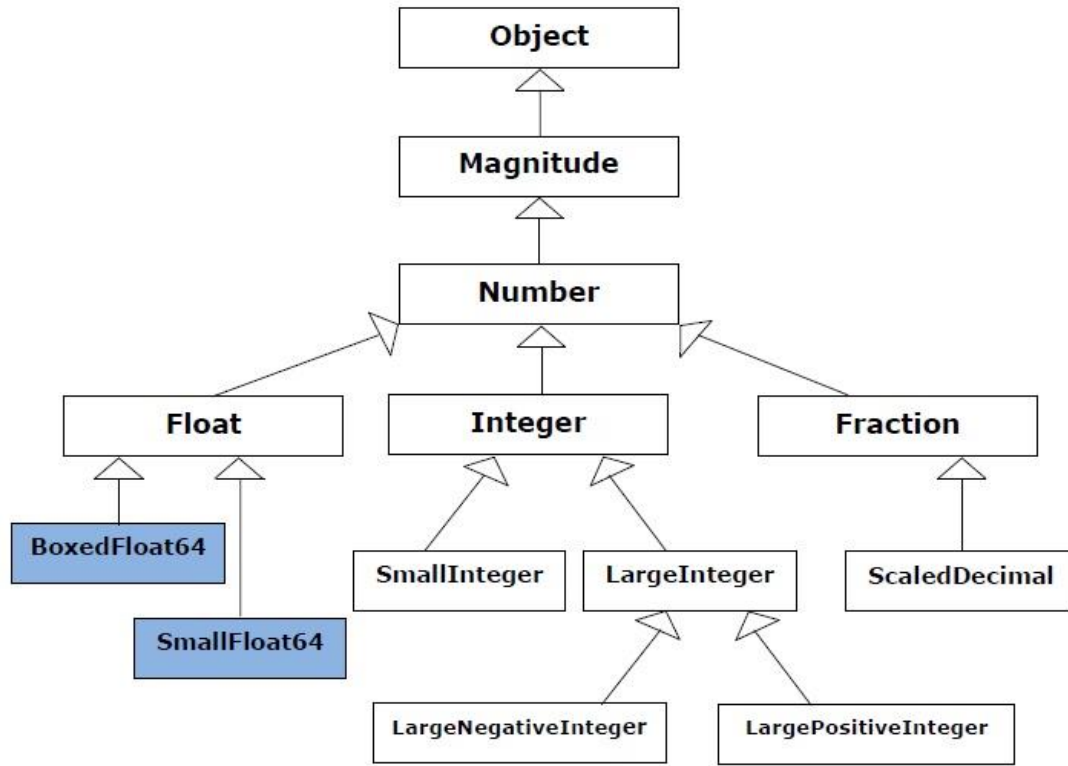


<b>Mensaje</b>		<b>Ejemplos</b>
<code>&lt; otraMagnitud</code>	Comparación por menor	<code>5 &lt; 10</code>
<code>&lt;= otraMagnitud</code>	Comparación por menor o igual	<code>3 &lt;= 3</code>
<code>&gt; otraMagnitud</code>	Comparación por mayor	<code>4 &gt; 10</code>
<code>&gt;= otraMagnitud</code>	Comparación por mayor o igual	<code>5 &gt;= 0</code>
<b>between:</b> <code>limiteInferior and: limiteSuperior</code>	Retorna <b>true</b> si el receptor se encuentra comprendido entre los límites indicados	<code>5 between: 3.14 and: 10</code>

Resultados obtenidos de la ejecución de los ejemplos:



Las diferentes subclases de **Magnitude** agregan el comportamiento necesario para especializar esta definición abstracta de magnitud hasta obtener magnitudes concretas tales como números, fracciones, fechas y horas.



### 3.8.3.1 NÚMEROS

Smalltalk cuenta con varias clases de números, todos ellos instancias de subclases de **Number**, esta es una clase abstracta que agrupa las características y comportamientos de los números en general.

Los números no son valores de datos primitivos sino objetos en sí mismos, los mismos se implementan de manera eficiente en la máquina virtual, y la jerarquía de **Number** es tan perfectamente accesible y extensible como cualquier otra jerarquía de clases.

Como vimos anteriormente, la clase abstracta de esta jerarquía es **Magnitud**, que representa todo tipo de clases de apoyo a los métodos de comparación. La clase **Number** añade métodos aritméticos y otros métodos principalmente abstractos.

**Float** y **Fraction** representan, respectivamente, números de punto flotante y valores fraccionarios. En Pharo 5.0 se añadieron las subclases flotantes **BoxedFloat64** y **SmallFloat64**, que representan **Float** en ciertas arquitecturas. Por ejemplo, **BoxedFloat64** sólo está disponible para sistemas de 64 bits.



La clase **Integer** es también abstracta, distinguiendo las subclases **SmallInteger** y **LargeInteger**. A su vez, **LargeInteger** tiene las subclases: **LargePositiveInteger** y **LargeNegativeInteger**.

Los números enteros se consideran **LargePositiveInteger** cuando son mayores o iguales a **1073741824**.

Los números enteros se consideran **SmallInteger** cuando son menores o iguales que **1073741823** y mayores o iguales que **1073741824**.

Los números enteros se consideran **LargeNegativeInteger** cuando son menores o iguales que **(-1073741825)**.

La siguiente es una lista de los principales mensajes que comprenden todos los objetos numéricos:

<b>Mensaje</b>		<b>Ejemplos</b>
<b>+</b> unNumero	Suma	5 + 10 Rta: 15
<b>-</b> unNumero	Resta	2 - 7 Rta: -5
<b>*</b> unNumero	Multiplicación	2.5 * 2 Rta: 5
<b>/</b> unNumero	División	10 / 3 Rta: 3.33
<b>//</b> unNumero	División entera	10 // 3 Rta: 3
<b>\</b> unNumero	Resto de la división entera	10 \ 3 Rta: 1
<b>Abs</b>	Valor absoluto	-10.5 abs Rta: 10.5
<b>Negated</b>	Devuelve el receptor cambiado de signo	1 negated Rta: -1
<b>Squared</b>	Devuelve el cuadrado del receptor	5 squared Rta: 25
<b>raisedTo:</b> unNumero	Devuelve el receptor elevado a la potencia indicada por <i>unNumero</i>	2 raisedTo: 3 Rta: 8
<b>Sqrt</b>	Devuelve la raíz cuadrada del receptor	16 sqrt Rta: 4
<b>Negative</b>	Retorna <b>true</b> si el receptor es negativo	-5 negative Rta: true
<b>Positive</b>	Retorna <b>true</b> si el receptor es positivo	10 positive Rta: true





<b>Even</b>	Retorna <b>true</b> si el receptor es par	4 even Rta: true
<b>Odd</b>	Retorna <b>true</b> si el receptor es impar	3 odd Rta: true
<b>Rounded</b>	Retorna el receptor como un entero	3.1415927 rounded Rta: 3 3.5 rounded <b>Rta: 4</b>

### 3.8.3.1.1 ENTEROS

La clase que representa números enteros es **Integer**; sin embargo números como 4 ó 10000 no son instancia de Integer, sino de alguna de sus subclases. Esto se debe a que atendiendo a razones de eficiencia en el uso de la memoria y dependiendo del valor absoluto del número en cuestión, es conveniente almacenarlo usando diferentes representaciones internas: para números pequeños basta con usar 16 bits, pero para números más grandes serán necesarios 32 bits. Pero estas diferencias de representación no hacen variar el comportamiento que exhiben los objetos numéricos enteros, que se encuentra definido en su totalidad en la clase abstracta Integer; sus subclases concretas solamente especializan la forma en que sus instancias se almacenan en memoria. [Cuenca].

Las instancias de **Integer**, además de todos los mensajes anteriores, son capaces de contestar los siguientes mensajes:

<b>Mensaje</b>		<b>Ejemplos</b>
<b>asCharacter</b>	Devuelve el carácter cuyo código ASCII es igual al valor del receptor	65 asCharacter Rta: \$A
<b>Factorial</b>	Devuelve el factorial del receptor	5 factorial Rta: 120
<b>radix: unEntero</b>	Devuelve en un string la representación del receptor en la base de numeración indicada por <i>unEntero</i>	25 radix: 15 Rta: '1A'

### 3.8.3.1.2 FRACCIONES

Cuando el resultado de una división entre enteros no da como resultado otro número entero, el resultado será una instancia de la clase **Fraction**.

Las fracciones también son números, por lo que responden a todo el protocolo descrito más arriba y además cuenta con los siguientes mensajes principales:



<b>Mensaje</b>		<b>Ejemplos</b>
<b>numerator</b>	Devuelve el numerador del receptor	(2/5) numerator Rta: 2
<b>denominator</b>	Devuelve el denominador del receptor	(2/5) denominator Rta: 5
<b>asFloat</b>	Devuelve el receptor como un número de punto flotante	(1/3) asFloat Rta: 0.3333333333333333

### 3.8.3.2 CARACTERES

Las instancias de la clase Character representan caracteres del código ASCII; hay por lo tanto 256 instancias de ésta clase. [Cuenca].

En adición a los heredados de Magnitude, los principales mensajes que comprenden las instancias de Character son los siguientes:

<b>Mensaje</b>		<b>Ejemplos</b>
<b>isAlphaNumeric</b>	Devuelve true si el receptor es un carácter alfanumérico	\$v isAlphaNumeric Rta: true
<b>isDigit</b>	Retorna true si el receptor es un dígito del 0 al 9	\$5 isDigit Rta: true
<b>isLowercase</b>	Retorna true si el receptor es un carácter en minúscula	\$a isLowercase Rta: true
<b>asUppercase</b>	Retorna el receptor en mayúsculas	\$a asUppercase Rta: \$A

### 3.8.3.3 FECHAS Y HORAS

Las clases Date y Time permiten obtener objetos que representan respectivamente fechas y horas. [Cuenca].

#### 3.8.3.3.1 CLASE DATE

La fecha de hoy puede obtenerse con la siguiente expresión:

<p><b>Date today</b></p> <p>Retorna la nueva fecha vigente, por ejemplo: 21 July 2014</p>
---

En general, una fecha cualquiera puede obtenerse con el siguiente mensaje:



**Date newDay:** nroDía **month:** nombreMes **year:** nroAño

Por ejemplo:

**Date newDay: 10 month: #july year: 1982**

Nos devuelve un objeto representando la fecha: 10 de Julio de 1982

Para evitar escribir el nombre del mes en el keyword **month** ; es posible utilizar el servicio de traducción ofrecido por la clase Date:

**Date newDay: 10 month: (Date nameOfMonth: 2) year: 1982**

Los principales mensajes ofrecidos por las instancias de Date son los siguientes (en los ejemplos se asumirá que la variable **unaFecha** se refiere al objeto creado con la expresión anterior, es decir: 10/7/1982):

<b>Mensaje</b>		<b>Ejemplos</b>
<b>dayOfMonth</b>	Retorna el día del mes	unaFecha dayOfMonth Rta: 10
<b>monthIndex</b>	Devuelve el denominador del receptor	unaFecha monthIndex Rta: 7
<b>Year</b>	Devuelve el receptor como un número de punto flotante	unaFecha year Rta: 1982
<b>addDays:</b> unNumero	Retorna otra fecha que resulta de sumar <i>unNumero</i> días al receptor	unaFecha addDays: 30 Rta: 9 August 1982
<b>subtractDays:</b> unNumero	Retorna otra fecha que resulta de restar <i>unNumero</i> días al receptor	unaFecha subtractDays: 10 Rta: 30 June 1982
<b>subtractDate:</b> otraFecha	Retorna la cantidad de días que separan al receptor de <i>otraFecha</i>	"supongamos fecha2 = 24 Enero 1982" unaFecha subtractDate: unaFecha2 Rta: 167
<b>Mmddyyy</b>	Retorna un String conteniendo una representación imprimible del receptor	unaFecha mmddyyyy Rta: '7/10/1982'

Ejemplos de ejecución en la ventana Transcript o Playground [Cuenca]:



**Ejemplo 1:**

```
| unaFecha |  
unaFecha := Date newDay:10 month:#July year: 1982.  
unaFecha dayOfMonth  
Rta: 10
```

**Ejemplo 2:**

```
| unaFecha unaFecha2 |  
unaFecha := Date newDay:10 month:#July year: 1982.  
unaFecha2 := Date newDay:24 month:#January year: 1982.  
unaFecha subtractDate: unaFecha2  
Rta: 167
```

### 3.8.3.3.2 CLASE TIME

Las instancias de Time representan un instante particular del día, expresado en segundo desde medianoche.

Para obtener la hora actual, se utiliza la expresión:

```
Time now
```

Para crear un objeto que represente un instante de tiempo en especial, se utiliza el mensaje:

```
Time fromSeconds: cantSegundosDesdeLasCeroHoras
```

Las instancias de Time responden a los siguientes mensajes (los ejemplos asumen que la variable *unTiempo* ha sido inicializada con la siguiente expresión: **Time fromSeconds: ( 60\*60\*6) + (30\*60)** ) Equivalente a las **6:30 a.m** ).



<b>Mensaje</b>		<b>Ejemplos</b>
<b>Hours</b>	Retorna la hora	unTiempo hours Rta: 6
<b>Minutes</b>	Retorna los minutos	unTiempo minutes Rta: 30
<b>Seconds</b>	Retorna los segundos	unTiempo seconds Rta: 0
<b>addTime: otroTime</b>	Retorna otro objeto Time que resulta de sumar <i>otroTime</i> al receptor	"supongamos que unTiempo2 es 1 hora 30 min 50 seg" unTiempo addTime: unTiempo2 Rta: 8:00:50 am
<b>subtractTime: otroTime</b>	Retorna otro objeto Time que resulta de restar <i>otroTime</i> al receptor	"supongamos que unTiempo2 es 1 hora 30 min 50 seg" unTiempo addTime: unTiempo2 Rta: 4:59:10 am

**Ejemplo 1:**

```
|unTiempo|  
unTiempo:= Time fromSeconds: ( (60*60*6) + (30*60)).  
unTiempo minutes  
Rta: 30
```

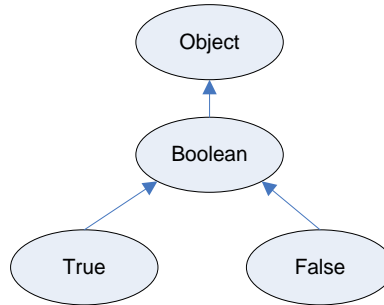
**Ejemplo 2:**

```
"supongamos que unTiempo2 es 1 hora 30 min 50 seg"  
|unTiempo unTiempo2|  
unTiempo:= Time fromSeconds: ( (60*60*6) + (30*60)).  
unTiempo2:= Time fromSeconds: ((60*60) + (30*60) + (50)).  
unTiempo addTime: unTiempo2  
Rta: 8:00:50 am
```

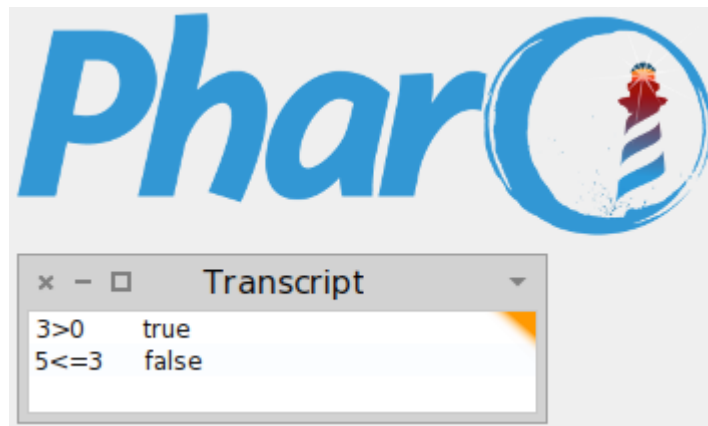


### 3.8.4 BOOLEANOS

Los objetos booleanos representan valores de verdad y falsedad. Existen solo 2 objetos booleanos: true y false, instancias respectivamente de las clases **True** y **False**, subclases a su vez de la clase abstracta Boolean.



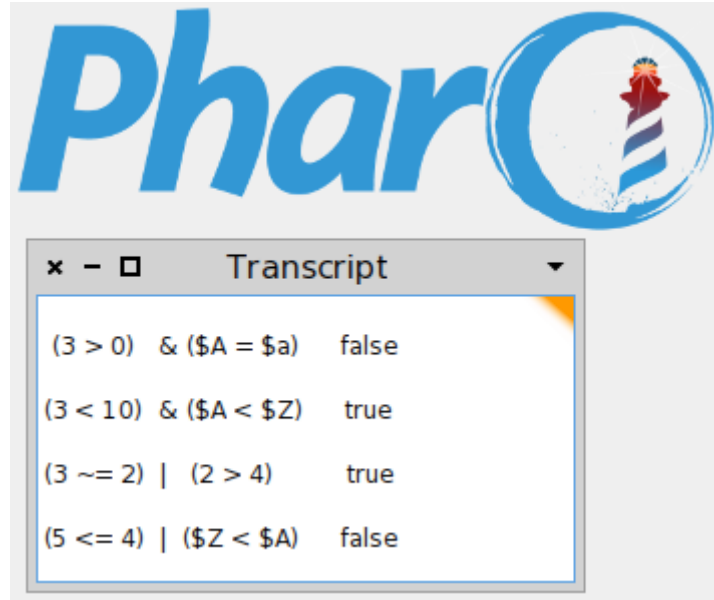
Los objetos booleanos se obtienen principalmente a través de la comparación de magnitudes.



Los objetos booleanos definen mensajes para permitir la construcción de expresiones lógicas:

& AND

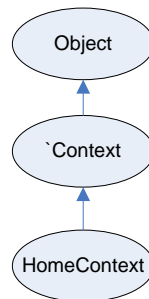
| OR



### 3.8.5 BLOQUES

Los bloques son objetos que representan secuencia de expresiones Smalltalk (código). [Cuenca].

Los bloques más altos son objetos instancias de la clase HomeContext y los bloques anidados son instancias de la clase Context. No es posible crear bloques mediante el envío del mensaje new a Context o HomeContext. [Lalonde97].



Un bloque está formado por una secuencia de sentencias separadas por punto y encerrada entre corchetes.

Ejemplo:

```
| a b c |
[ a:= 5. b:= 7. c:= a negated + b factorial ].
```

La secuencia de instrucciones contenida en el bloque se ejecuta cuando el bloque recibe el mensaje **value**. El resultado de la evaluación del bloque es igual al valor de la última sentencia del bloque.



Si evaluamos el bloque del ejemplo anterior, obtenemos:

```
| a b c |  
[   a:= 5.  
    b:= 7.  
    c:= a negated + b factorial  
] value  
Resultado: 5035
```

Los bloques constituyen objetos como cualquier otro, por lo cual pueden ser asignados a variables:

```
| a b c unBloque |  
unBloque := [ a:= 5.  
              b:= 7.  
              c:= a negated + b factorial  
            ].  
unBloque value      Resultado: 5035
```

Un bloque sin instrucciones se denomina bloque nulo y al ser evaluado devuelve nil.

```
| bloqueNulo |  
bloqueNulo := [ ].  
bloqueNulo value      Resultado: nil
```

Un bloque puede recibir ninguno, uno o dos parámetros. En cada uno de esos casos la sintaxis es la siguiente:

```
[ ... sentencias ... ].      “ningún parámetro”  
[ : x | ... sentencias ... ].  “un parámetro”  
[ :x :y | ... sentencias ... ]. “dos parámetros”
```

Los mensajes para evaluar bloques de uno y dos parámetros son, respectivamente:

```
value: parámetro  
value: parámetro value: parámetro2
```

Ejemplos:

```
“bloque de un parámetro”  
| unBloque |
```





```
unBloque := [ :x | x*2 ].
```

*Paso de mensajes al bloque con un parámetro:*

```
unBloque value: 2 Resultado: 4
```

```
unBloque value: 3 Resultado: 6
```

“bloque de dos parámetros”

```
| unBloque |
```

```
unBloque := [ :x :y | x + y ].
```

*Paso de mensajes al bloque con dos parámetros:*

```
unBloque value: 2 value: 3 Resultado: 5
```

```
unBloque value: 5 value: -8 Resultado: -3
```

El principal uso de los bloques es en la construcción de estructuras de control. Además, pueden utilizarse para construir objetos que encapsulen algoritmos.

### 3.8.5.1 ESTRUCTURAS DE CONTROL

En Smalltalk, las estructuras de control se construyen a partir de mensajes enviados a objetos booleanos, bloques y números, y de la cooperación entre ellos.

#### 3.8.5.1.1 ESTRUCTURAS CONDICIONALES

Se forman enviando mensajes a objetos booleanos y evaluando bloques si las condiciones se cumplen.

Ejemplos:

```
( a>b ) ifTrue: [ c:= a + b ]. Ejecuta el bloque si la condición es verdadera  
( a=b ) ifFalse: [ c:= a + b]. Ejecuta el bloque si la condición es falsa.  
( a>=b ) Ejecuta el bloque [ c:= a + b ] si la condición  
ifTrue: [ c:= a + b ]. es verdadera; en caso contrario, ejecuta  
ifFalse: [ c:= a - b ]. el otro bloque.
```

Ejemplo en Playground:



```
"Ejemplo de bloque"  
|a b c|  
a:=10.  
b:=5.  
(a>=b)  
  ifTrue: [ c:=a+b ]  
  ifFalse: [c:= a-b]. 15
```

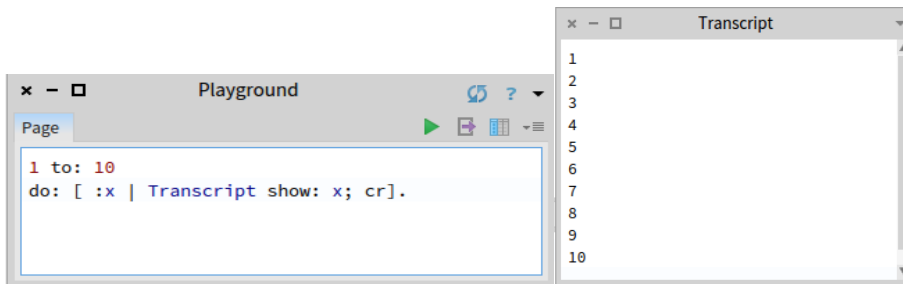
### 3.8.5.1.2 ESTRUCTURAS REPETITIVAS DETERMINADAS

Se forman a partir de mensajes enviados a números entero y bloques de un parámetro.

#### Ejemplo 1:

```
1 to: 10           Itera de 1 hasta 10, imprimiendo los sucesivos números  
do: [ :x | Transcript show: x; cr].      en la ventana Transcript.
```

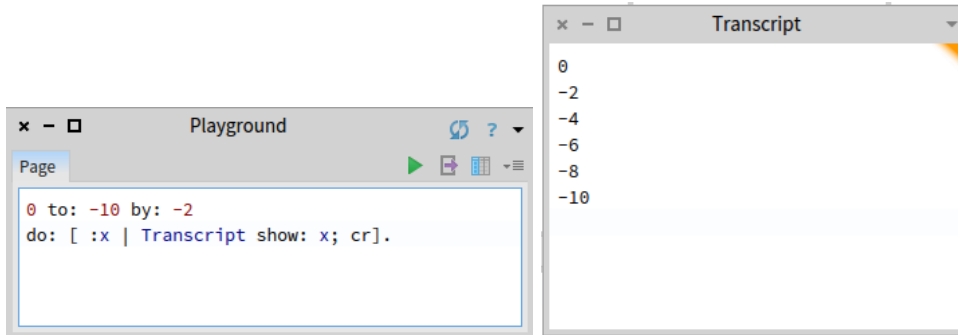
Resultado visualizado en la ventana Transcript:



#### Ejemplo 2:

```
0 to: -10           Itera de 0 a -10, de a dos valores por vez, e imprime  
by: -2              esos valores en la ventana Transcript.  
do: [ :x | Transcript show: x; cr.]
```

Resultado visualizado en la ventana Transcript:

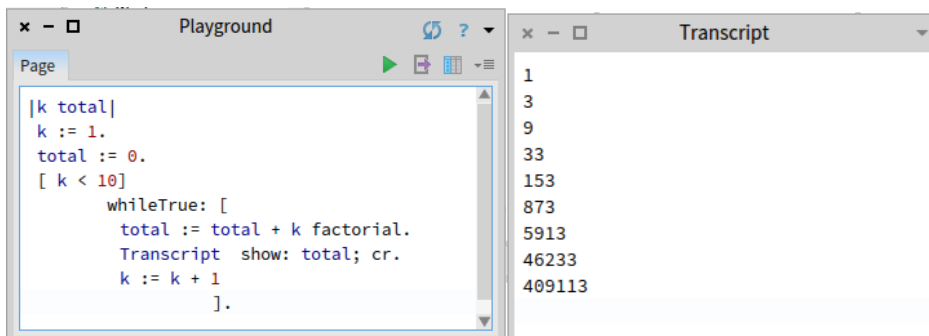


### 3.8.5.1.3 ESTRUCTURAS REPETITIVAS INDETERMINADAS (O CONDICIONALES)

Se forman a partir de mensajes a bloques que evalúan en valores booleanos.

```
|k total|  
k:= 1.  
total:= 0.           Repite el segundo bloque mientras la condición  
                    k<10 sea verdadera.  
[ k<10]  
    whileTrue: [  
        total:= total + k factorial.  
        Transcript show: total; cr.  
        k := k +1  
    ]
```

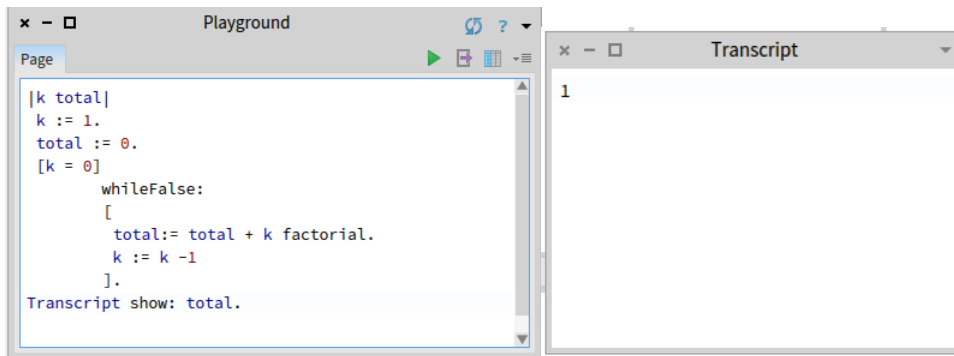
Resultado visualizado en la ventana Transcript:





```
|k total|
k:= 1.
total:= 0.      “Repite el segundo bloque mientras la condición k=0 sea falsa.”
[k=0]
    whileFalse: [ total:= total + k factorial.
                  k := k -1].
Transcript show: total.
```

Resultado visualizado en la ventana Transcript:



#### 3.8.5.1.4 RESOLUCIÓN DE CASO DE ESTUDIO

Se requiere desarrollar un programa orientado a objetos en Smalltalk, que permita modelar los artículos que tiene a la venta un comercio. De cada artículo se conoce su:

- Código, es un valor numérico que identifica a cada artículo.
- Descripción, correspondiente al nombre del artículo.
- Costo, que representa el costo de compra en pesos del artículo.
- Tipo, los valores que puede asumir son: 1, 2, ó 3.
- Stock, que representa la cantidad de unidades disponibles en stock del artículo.

Además, cada artículo debe exhibir el siguiente comportamiento:

- a) precioVenta, que calcule el precio unitario de venta el cual se calcula sumándole al costo un porcentaje de ganancia que depende directamente del tipo de artículo. Si es tipo 1 el porcentaje de ganancia es el



10%; si es tipo 2, el porcentaje de ganancia es del 20%; si es tipo 3, el porcentaje de ganancia es del 25%.

- b) += unaCantidad, que permita incrementar el stock actual del artículo en una cierta cantidad, siempre y cuando esta cantidad fuese mayor o igual a cero.
- a) -= unaCantidad, que permita descontar del stock actual del artículo una cierta cantidad, siempre y cuando el stock sea mayor o igual a esta cantidad, y esta cantidad fuese positiva o cero.
- c) calcularImporteTotalStock, que calcule el importe total de venta que se obtendría de todo el stock de artículos.
- d) asString, que devuelva todos los datos del artículo.

```
Object subclass: #Articulo "Autor: Ing. Nicolas Colaccioppo"
  instanceVariableNames: 'codigo descripcion tipo costo stock'
  classVariableNames: ''
  package: 'EjPrimeraSemana-GestiónArtículos'

initialize
  "inicializa los atributos de un artículo con valores por defecto"
  self codigo: 0.
  self descripcion: 'no especificado'.
  self costo:0.
  self tipo: 1.
  self stock: 0.

codigo
  "devuelve el valor de código"
  ^ codigo

codigo: unCodigo
  "asigna unCodigo al atributo código del artículo"
  codigo:=unCodigo.

costo
  "Devuelve el valor de costo"
  ^ costo

costo: unCosto
  "asigna unCosto al atributo costo del artículo siempre y cuando unCosto
  sea Number y sea positivo o 0"
  unCosto isNumber
  ifTrue:
  [
    (unCosto>=0.0)
    ifTrue:
    [
```



```
        costo:=unCosto.  
        ^true.  
    ].  
].  
^false.  
  
descripcion  
"Devuelve el valor de descripcion"  
^ descripcion  
  
descripcion: unaDescripcion  
"asigna unaDescripcion al atributo descripción del artículo"  
descripcion:=unaDescripcion.  
  
stock  
"Devuelve el valor de stock"  
^ stock  
  
stock: unStock  
"asigna unStock al atributo stock del artículo siempre y cuando unStock  
sea Number y sea positivo o 0"  
unStock isNumber  
ifTrue:  
[  
    (unStock>=0)  
    ifTrue:  
    [  
        stock:=unStock.  
        ^true.  
    ].  
].  
^false.  
  
tipo  
"Devuelve el valor de tipo"  
^ tipo  
  
tipo: unTipo  
"asigna unTipo al atributo tipo del artículo siempre y cuando unTipo sea  
Integer y esté comprendido entre 1 y 3"  
unTipo isInteger  
ifTrue:  
[  
    ((unTipo>=1) & (unTipo<=3))  
    ifTrue:  
    [  
        tipo:=unTipo.  
        ^true.  
    ].  
].  
^false.
```



```
asString
    "devuelve todos los datos del artículo"
    ^String cr, 'Código: ',self codigo asString, String cr, 'Descripción:
',self descripcion asString, String cr, 'Tipo: ',self tipo asString, String
cr, 'Stock: ',self stock asString.

+=unaCantidad
    "agrega unaCantidad al stock de artículos"
    unaCantidad isNumber
    ifTrue:
        [
            unaCantidad>=0
            ifTrue:
                [
                    self stock: self stock + unaCantidad.
                ].
        ].

-=unaCantidad
    "descuenta unaCantidad del stock de artículos"
    unaCantidad isNumber
    ifTrue:
        [
            self stock >= unaCantidad
            ifTrue:
                [
                    self stock: self stock - unaCantidad.
                ].
        ].

importeTotalEnStock
    "devuelve el importe total de todos los artículos en stock"
    ^self stock * self importeVenta.

importeVenta
    "devuelve el importe de venta unitario en base al costo y tipo de
artículo"
    |porcGanancia|
    porcGanancia:=10. "Por defecto asumimos que es tipo 1"
    (self tipo = 2)
    ifTrue: [
        porcGanancia:=20.
    ]
    ;
    ifFalse: [
        (self tipo = 3)
        ifTrue: [
            porcGanancia:=25.
        ].
    ].
    ^self costo + (self costo * (porcGanancia/100) asFloat).
```



En Playground:

```
|art|
art:=Articulo new initialize.
Transcript show: art asString; cr.

art codigo:1; descripcion:'Mouse'; tipo:4; costo: 100; stock:100.
Transcript show: art asString; cr.

Transcript show: 'Importe total en stock: $',art importeTotalEnStock; cr.

art -=30.
Transcript show: art asString; cr.
Transcript show: 'Importe total en stock: $',art importeTotalEnStock; cr.

art+=50.
Transcript show: art asString; cr.
Transcript show: 'Importe total en stock: $',art importeTotalEnStock; cr.
```

## 3.9 COLECCIONES EN SMALLTALK

### 3.9.1 INTRODUCCIÓN

Imaginemos querer modelar una agenda para una semana. Creamos nuestra clase Semana y, como sabemos que los días de la semana son 7, creamos 7 variables de instancia para cada uno de ellos en la clase. Esta situación no siempre es posible representar sólo con variables de instancias. Muchas veces vamos a necesitar modelar relaciones entre un objeto y varios (una relación 1 a N). Por ejemplo, los departamentos de un edificio, los estudiantes de una universidad, etc. En muchas oportunidades no vamos a poder conocer el número total o máximo de elementos o simplemente sólo sabremos que van a ser muchos. Para tales situaciones nos van a servir las colecciones.

Las **colecciones** permiten agrupar objetos para poder operar sobre ellos. En general podremos referirnos a esa colección de objetos bajo un mismo nombre, y poder agregar o sacar objetos, filtrarlos, recorrerlos y muchas otras cosas de manera mucho más sencilla. Las colecciones nos van a permitir modelar conjuntos de objetos, como ser: las piezas de un tablero, los empleados de una fábrica, la flota de buques de una empresa de transporte fluvial, etc. y poder realizar operaciones sobre ese grupo.

Las colecciones se pueden ver como un conjunto de objetos, y de hecho vamos a decir eso normalmente (abusando del lenguaje), pero en realidad las colecciones son conjuntos de referencias a objetos; es decir, los objetos no están adentro de la colección, sino que la colección los conoce (los referencia) de alguna forma. Es importante aclarar que, conceptualmente, la colección no tiene “adentro suyo” a sus elementos. Al agregar un objeto a la colección simplemente se agrega una referencia que parte de la colección y llega al objeto “agregado”. Es importante destacar también que los objetos que se agregan a una colección pueden estar referenciados por otros objetos que existan y que no necesariamente estén relacionados con la colección.

Los objetos agregados pueden ser incluso otras colecciones, dado que las colecciones son también objetos.





Pero, ¿y si las colecciones son un conjunto de objetos (o referencias más correctamente), cómo las representamos? Sabemos que en Smalltalk sólo tenemos objetos y mensajes, por lo cual las colecciones van a ser representadas como...**objetos!!!**

Resumiendo, **un objeto colección es un objeto que representa un conjunto de referencias a otros objetos**, con el cual se van a poder realizar distintas operaciones sobre los elementos que ella referencia.

### ¿Qué operaciones podemos hacer con una colección?

Para poder responder más fácilmente y poder comprenderlo, vamos a elegir un ejemplo: una colección de CDs.

Con la colección de CDs podemos:

- Mirar los CDs (recorrer la colección)
- Organizarlos por artista (ordenar la colección)
- Agregar nuevos CDs (agregar elementos a la colección)
- Regalar un CD (quitar elementos de la colección)
- Quedarme con los CDs de rock nacional (hacer un filtro o selección de los elementos según un criterio)
- Saber si tengo un CD de Kill Bill (preguntarle a una colección si incluye o no un determinado objeto como elemento)



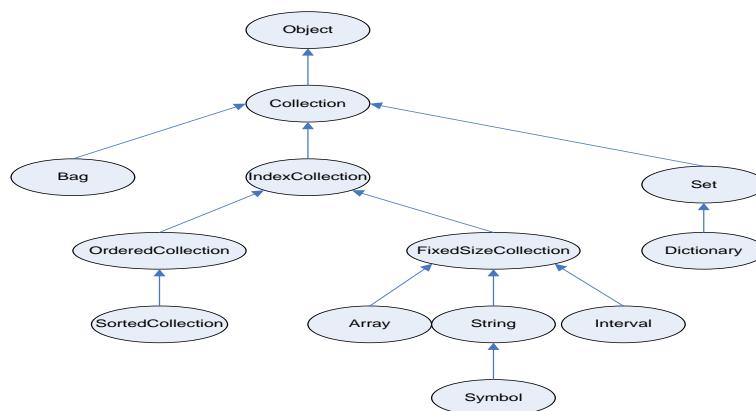
### ¿Y cómo se modela este comportamiento con objetos de Smalltalk?

La respuesta debería ser obvia: con métodos definidos para las colecciones. Es decir, podemos crear nuestra instancia de colección que representa a nuestra colección de CDs y a ese objeto enviarle mensajes para agregarle un nuevo objeto CD, para que nos filtre los CDs de rock nacional, etc.

En Smalltalk no estamos limitados en cuanto al tipo de objeto que podemos agregar. Por ejemplo, podemos agregar a nuestra colecciónCDs un número, un String, una fecha... lo que sea. Se dice que las colecciones en Smalltalk son heterogéneas dado que permiten que cualquier objeto de cualquier clase pueda ser agregado a ellas junto con otros objetos de otras clases.

Las colecciones de Smalltalk son polimórficas y presentan un protocolo unificado, por lo que todas las diferentes formas de colecciones provistas por el entorno responden a un mismo conjunto de mensajes básicos, lo que facilita su aprendizaje y el cambio de un tipo de colección a otra. [Cuenca].

El siguiente gráfico muestra la jerarquía de colecciones de Smalltalk/V [Cuenca]:





Existen en esta jerarquía clases abstractas y concretas.

CLASES ABSTRACTAS [Cuenca]:

- **Collection**: define el comportamiento común a todas las posibles formas de representar colecciones.
- **IndexedCollection**: representa una forma más especializada de colección en donde es posible acceder a cada elemento de la colección por medio de subíndices numéricos.
- **FixedSizeCollection**: especializa `IndexCollection` para representar colecciones subindicables con una cantidad fija de elementos. Sus subclases concretas permiten manejar arreglos (**Array**), cadenas de caracteres (**String**) e intervalos numéricos (**Interval**).

CLASES CONCRETAS [Cuenca]:

- **Set** y **Bag** son colecciones desordenadas, las instancias de `Bag` admiten duplicados, mientras que las de `Set` no.
- **OrderedCollection** es una colección indexada en la que se mantiene el orden en que los objetos son insertados. Tiene una subclase, **SortedCollection**, en la cual es posible especificar que los elementos de la colección se ordenen siguiendo un criterio específico (por ejemplo, de menor a mayor).
- **Dictionary** es una colección que mantiene asociaciones entre objetos llamados claves (keys) y valores (values).

### 3.9.2 SET<sup>1</sup>

La primera clase que vamos a usar es una colección que se llama `Set`. Un `Set` representa un conjunto entendido como los conjuntos matemáticos que aprendimos en la escuela.

El orden de los objetos agregados en una colección `Set` no corresponde con el orden de inserción de los mismos y tampoco tiene un criterio de orden detectable. Ésta es una de las características que distinguen a un `Set` entre otros tipos de colecciones que existen en `Smalltalk`: los elementos no están ordenados. La otra característica de un `Set` es que no repite sus elementos, es decir que si agregamos dos veces un mismo objeto, no aparecerá dos veces en él.

Para crear una colección `Set` y asignarla a una variable que represente a nuestra colección de CDs, escribimos:

```
coleccionCDs := Set new.
```

Supongamos que tenemos definida la clase de los CDs. Creamos instancias de CDs y los agregamos a nuestro conjunto `coleccionCDs`, mandándole el mensaje `add:`. Definimos además un cuarto CD que no agregamos.

---

<sup>1</sup> Basado en el apunte "Colecciones en `Smalltalk`" de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires (cátedra de Paradigmas de Programación), de Victoria Pocladova, Carlos Lombardi, Leonardo Volinier y Jorge Silva.



```
cd1 := CD new.  
cd1 titulo: 'Abbey Road'.  
cd1 autor: 'The Beatles'.  
cd1 origen: 'U.K.'.  
  
cd2 := CD new.  
cd2 titulo: 'Tribalistas'.  
cd2 autor: 'Tribalistas'.  
cd2 origen: 'Brasil'.  
  
cd3 := CD new.  
cd3 titulo: 'Peluson of milk'.  
cd3 autor: 'Spinetta'.  
cd3 origen: 'Argentina'.  
  
cd4 := CD new.  
cd4 titulo: 'Piano Bar'.  
cd4 autor: 'Charly García'.  
cd4 origen: 'Argentina'.  
  
coleccionCDs add: cd1.  
coleccionCDs add: cd2.  
coleccionCDs add: cd3.
```

A este conjunto se le pueden enviar algunos mensajes:

```
coleccionCDs size.           => "devuelve 3"  
coleccionCDs isEmpty.       => "devuelve false ya que la colección tiene  
elementos"  
coleccionCDs includes: cd2. => "devuelve true"  
coleccionCDs includes: cd4. => "devuelve false"
```

### 3.9.3 BAG<sup>2</sup>

Esta colección es de las denominadas "sin orden" y es la representación de una bolsa. Una bolsa de objetos. El Bag es la colección más adecuada para representar por ejemplo un carrito o bolsa de supermercado. Imagínense que puedo poner tres latas de tomates, dos botellas de agua y siete peras.

---

<sup>2</sup> Basado en el apunte "Colecciones en Smalltalk" de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires (cátedra de Paradigmas de Programación), de Victoria Pocladova, Carlos Lombardi, Leonardo Volinier y Jorge Silva.



Imagínense que tengo una lata de tomates. Lo vamos a representar así:

```
unaLata := LataDeTomates new.  
unaLata conTomates: 'perita'.  
unaLata deLaMarca: 'La Marca que a uds les guste'  
unaLata conPrecio: 1.00
```

Bien, tenemos el objeto `unaLata` y queremos agregar a mi carrito de compras 3 latas de tomates. Entonces hago:

```
miCarrito add: unaLata.  
miCarrito add: unaLata.  
miCarrito add: unaLata.
```

En mi carrito debería haber 3 latas de tomates. Pero, como vimos, si representamos a mi carrito de compras con un `Bag` sí voy a tener 3 objetos dentro de mi carrito.

El `Set` está concebido para la representación de conjuntos. Una propiedad distintiva sobre conjuntos es que los elementos que pertenecen al conjunto son únicos en él. Es decir, en los conjuntos no vamos a tener dos veces el mismo elemento (no admite repetidos). Salvo por esta propiedad, el comportamiento es el mismo que el del `Bag`.

### 3.9.4 ORDEREDCOLLECTION

Es una colección indexada en la que se mantiene el orden en que los objetos son insertados o agregados a la colección [Cuenca]. Al contrario de un `Set` o un `Bag`, en donde los elementos están "todos tirados".

Operaciones que se pueden llevar a cabo con las colecciones indexadas:

<b>Mensaje</b>	
<b>at:</b> indice <b>put:</b> unObjeto	Coloca <i>unObjeto</i> en la posición <i>indice</i> de la colección ( <i>indice</i> debe ser un número entero)
<b>at:</b> indice	Retorna el objeto ubicado en la posición <i>indice</i> de la colección
<b>indexOf:</b> unObjeto	Retorna el índice de la posición que <i>unObjeto</i> ocupa en la colección ( uno (1) es el índice del primer elemento). Si <i>unObjeto</i> no figura en la colección, devuelve cero (0)
<b>indexOf:</b> unObjeto	Retorna el índice de la posición que <i>unObjeto</i> ocupa en la colección ( uno



<b>ifAbsent:</b> unBloque	(1) es el índice del primer elemento). Si <i>unObjeto</i> no figura en la colección, evalúa <i>unBloque</i>
<b>First</b>	Retorna el primer elemento de la colección
<b>Last</b>	Devuelve el último elemento de la colección
<b>copyFrom:</b> inicio <b>to:</b> fin	Retorna una colección conteniendo los elementos del receptor comprendidos entre <i>inicio</i> y <i>fin</i> .
<b>replaceFrom:</b> inicio <b>to:</b> fin <b>with:</b> otraColeccion	Reemplaza los elementos del receptor comprendidos entre <i>inicio</i> y <i>fin</i> por los objetos contenidos en <i>otraColeccion</i>
<b>replaceFrom:</b> inicio <b>to:</b> fin <b>with:</b> otraColeccion <b>startingAt:</b> otroIndice	Reemplaza los elementos del receptor comprendidos entre <i>inicio</i> y <i>fin</i> por los objetos contenidos en <i>otraColeccion</i> , extrayéndolos a partir de la posición <i>otroIndice</i>
<b>replaceFrom:</b> inicio <b>to:</b> fin <b>withObject:</b> unObjeto	Reemplaza los elementos del receptor comprendidos entre <i>inicio</i> y <i>fin</i> por <i>unObjeto</i>

Si creamos mi colección así<sup>3</sup>:

```
miCol := OrderedCollection new.  
miCol add: 'esto'.  
miCol add: 'es'.  
miCol add: 'un'.  
miCol add: 'ejemplo'.
```

Después podemos pedirle varias cosas:

```
miCol first el primero 'esto'  
miCol last el último 'ejemplo'  
miCol at: 3 el tercero 'un'  
miCol at: 4 el cuarto 'ejemplo'  
miCol size cantidad de elementos 4
```

Y podemos seguir agregándole elementos, ante lo cual la colección "se estira", pues tiene un tamaño variable (a diferencia de un array que veremos más adelante).

```
miCol first el primero 'esto'
```

<sup>3</sup> Basado en el apunte "Colecciones en Smalltalk" de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires (cátedra de Paradigmas de Programación), de Victoria Pocladova, Carlos Lombardi, Leonardo Volinier y Jorge Silva.



```
miCol add: 'agrego'.  
miCol add: 'cosas'.  
miCol size cantidad de elementos ahora 6  
miCol last el último ahora 'cosas'
```

Aquí presentamos otro ejemplo:

```
| unString fragmento |  
unString := 'Hola, buenos días'.  
fragmento := unString copyFrom: 7 to: (unString size).  
fragmento Resultado: 'buenos días'
```

### 3.9.5 SORTEDCOLLECTION

La clase **SortedCollection** [Cuenca] provee colecciones ordenadas según un criterio específico.

Si no definimos el criterio, la SortedCollection ordena los elementos por su “orden natural” (significa que los ordenará de menor a mayor, usando el mensaje <). Dicho de otra forma, no ordena por orden de llegada, sino por comparación entre los elementos.

Si se instancia una SortedCollection con la expresión:

```
SortedCollection new
```

Se obtiene una colección que ordena sus elementos de menor a mayor. Esto implica que los objetos contenidos en una SortedCollection que utiliza el criterio de ordenamiento por defecto deben poder responder al mensaje <=.

#### Ejemplo 1:

```
| unaSortedCollection |  
unaSortedCollection := SortedCollection new.  
unaSortedCollection add: 1.      => ( 1 )  
unaSortedCollection add: 5.      => ( 1 5 )  
unaSortedCollection add: 2.      => ( 1 2 5 )  
unaSortedCollection add: 4.      => ( 1 2 4 5 )  
unaSortedCollection add: -8.     => ( -8 1 2 4 5 )
```

Ejecución del ejemplo:



The screenshot shows a REPL window with two panes. The left pane, titled 'Playground', contains the following code:

```
| unaSortedCollection |  
unaSortedCollection := SortedCollection new.  
unaSortedCollection add: 1.  
unaSortedCollection add: 5.  
unaSortedCollection add: 2.  
unaSortedCollection add: 4.  
unaSortedCollection add: -8.  
Transcript show: unaSortedCollection asString.
```

The right pane, titled 'Transcript', shows the output: 'a SortedCollection(-8 1 2 4 5)'. The cursor is positioned at the end of the output string.

Si queremos ordenar los elementos de la colección con un criterio en particular, necesitamos pasárselo a la colección ya creada. La forma de hacerlo, es pasarle lo que denominamos “sortBlock”, que es un bloque (objeto de la clase BlockClosure).

Para especificar un algoritmo de ordenamiento diferente, la colección debe instanciarse con la siguiente expresión:

```
SortedCollection sortBlock: unBloqueBinario
```

En donde unBloqueBinario es un bloque de dos parámetros cuya misión es determinar si el primero de ellos debe ubicarse antes que el segundo.

#### Ejemplo: crear una colección que ordene de mayor a menor:

```
| unBloque unaSortedCollection |  
unBloque := [ :x :y | x >= y ].  
unaSortedCollection := SortedCollection sortBlock: unBloque.  
unaSortedCollection := SortedCollection new.  
unaSortedCollection add: 1.      => ( 1 )  
unaSortedCollection add: 5.      => ( 5 1 )  
unaSortedCollection add: 2.      => ( 5 2 1 )  
unaSortedCollection add: 4.      => ( 5 4 2 1 )  
unaSortedCollection add: -8.     => ( 5 4 2 1 -8 )
```

#### Ejecución del ejemplo:



```
Playground
| unBloque unaSortedCollection |
unBloque := [ :x :y | x >= y ].
unaSortedCollection := SortedCollection sortBlock:
unBloque.
unaSortedCollection add: 1.
unaSortedCollection add: 5.
unaSortedCollection add: 2.
unaSortedCollection add: 4.
unaSortedCollection add: -8.
Transcript show: unaSortedCollection asString.
```

```
Transcript
a SortedCollection(5 4 2 1 -8)
```

Una SortedCollection puede reordenarse en cualquier momento enviándole el mensaje:

```
unaSortedCollection sortBlock: otroBloqueBinario
```

#### Acerca del sortblock<sup>4</sup>

El **sortBlock** es un bloque que necesita 2 parámetros, que son los objetos a comparar. El código del bloque es un código que debe devolver true o false. Para ordenar los objetos dentro de la colección, se evalúa el código y si el objeto retornado es true, el primer parámetro va antes que el segundo. Si retorna false, el segundo parámetro se ubica antes que el primero.

### 3.9.6 ARRAY

Array es una colección donde su característica distintiva es que es de tamaño fijo. Por ejemplo, para instanciar un Array, hacemos `Array new: 6`, donde 6 es la cantidad de elementos que contendrá. En casos especiales donde los elementos son literales booleanos, strings o números, se pueden crear los arreglos de la siguiente forma:

**# (objeto1 objeto2 objeto3 . . . objetoN)**

que representa una instancia de Array conteniendo los objetos que se indican.

Los **arreglos literales** son especialmente útiles para crear colecciones inicializadas.

```
Ejemplo 1: #(1 2 3 4 5) class "Resultado: "
```

<sup>4</sup> Basado en el apunte "Colecciones en Smalltalk" de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires (cátedra de Paradigmas de Programación), de Victoria Pocladova, Carlos Lombardi, Leonardo Volinier y Jorge Silva.





```
#(1 2 3 4 5) class "Resultado: " Array
```

**Ejemplo 2:** `#('hola' true 5 1/3 $D) class "Resultado: "`

```
#('hola' true 5 1/3 $D) class "Resultado: " Array
```

Los Arrays no implementan el mensaje `add`; justamente porque no se pueden modificar su tamaño. La forma de agregarles elementos es a través del mensaje `at:put:`, como por ejemplo:

```
miVector := Array new: 2.  
miVector at: 1 put: unaLata.
```

### 3.9.7 DE UN TIPO A OTRO

Si bien los arreglos literales son instancia de `Array`, es posible obtener a partir de ellos una instancia de la clase deseada usando los mensajes de conversión; por ejemplo: `[Cuenca]`

```
#(1 2 3 4 5) asOrderedCollection
```

```
#(1 2 3 4 5) asOrderedCollection an OrderedCollection(12345)
```



```
#{1 2 3 4 5) asset
```

```
#{1 2 3 4 5) asBag
```

```
#{1 2 3 4 5) asSortedCollection
```

Todas las colecciones<sup>5</sup> entienden una serie de mensajes que permiten obtener distintos tipos de colecciones con los mismos elementos que la colección receptora. Estos mensajes son de la forma “as{ColeccionQueQuiero}”.

Por ejemplo, si tuviese una colección de la clase Bag, y quiero sacarle los repetidos, sé que el Set no tiene repetidos, entonces tomo mi colección como un Set. Hacemos lo siguiente:

```
sinRepetidos := miCarrito asSet.
```

Si tuviese un array, y lo quiero convertir en una colección de tamaño variable, podría hacer:

```
coleccionVariable := miVector asOrderedCollection.
```

Si quisiera ordenar mi carrito de compras del producto más caro al más barato, haría algo como:

```
ordenadosPorPrecio := miCarrito asSortedCollection:  
    [:unProd :otroProd | unProd precio > otroProd  
    precio].
```

El mensaje asSortedCollection: recibe un parámetro que, obviamente, es un sortBlock.

También está el mensaje asSortedCollection (o sea sin el dos-puntos, o sea que no requiere parámetro) que, como dijimos antes, ordenará los elementos por el “orden natural”, dado por el mensaje <. En este caso, debemos tener en cuenta que todos los objetos que agreguemos a esta colección deberán entender el mensaje <, pues si así no fuera, tendríamos un error de mensaje no entendido.

### 3.9.8 OPERACIONES SOBRE COLECCIONES

#### 3.9.8.1 CREACIÓN DE INSTANCIAS

Para crear colecciones se debe enviar alguno de los siguientes mensajes a la clase que se debe instanciar:

---

<sup>5</sup> Basado en el apunte “Colecciones en Smalltalk” de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires (cátedra de Paradigmas de Programación), de Victoria Pocladova, Carlos Lombardi, Leonardo Volinier y Jorge Silva.

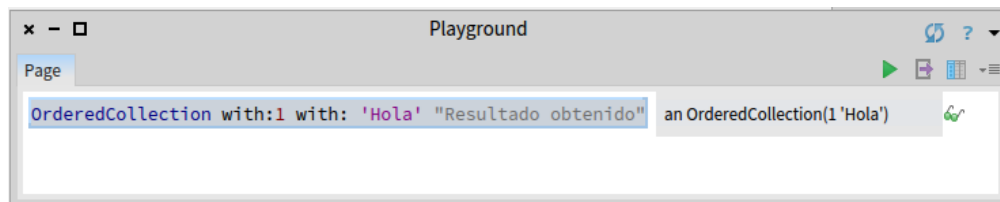


<b>Mensaje</b>		<b>Ejemplos</b>
<b>new</b>	Instancia una colección, sin especificar el tamaño inicial	OrderedCollection new Dictionary new
<b>new: tamañoInicial</b>	Instancia una colección, con un cierto tamaño inicial	Array new: 20
<b>with: unObjeto</b>	Instancia una colección conteniendo inicialmente <i>unObjeto</i>	OrderedCollection with: 'Hola'
<b>with:objeto1 with:objeto2</b>	Instancia una colección conteniendo inicialmente los dos objetos indicados	Array with:1 with:2
<b>with:objeto1 with:objeto2 with:objeto3</b>	Instancia una colección conteniendo inicialmente los tres objetos indicados	Set with:1 with:true with:'Hola'
<b>with:objeto1 with:objeto2 with:objeto3 with:objeto4</b>	Instancia una colección conteniendo inicialmente los cuatro objetos indicados	SortedCollection with:1 with:6 with:4.5 with:1/3

Nota:

- Los mensajes **new** y **new: tamaño inicial** son ilegales en intervalos.
- **with:...** no son aplicables a intervalos y diccionarios.

Ejemplo:



### 3.9.8.2 CONSULTA

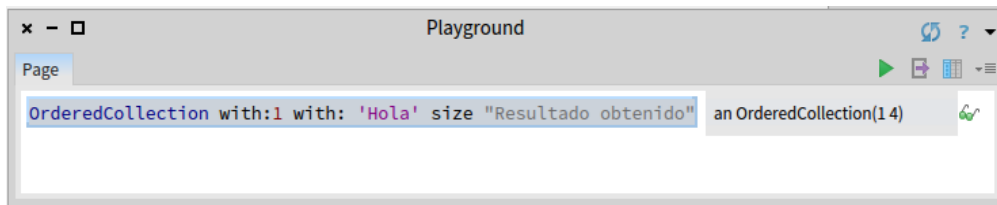
<b>Mensaje</b>	
<b>Size</b>	Devuelve la cantidad de elementos contenidos en la colección
<b>isEmpty</b>	Retorna <b>true</b> si la colección está vacía
<b>notEmpty</b>	Retorna <b>true</b> la colección contiene algún objeto
<b>includes: unObjeto</b>	Retorna <b>true</b> si la colección incluye a <i>unObjeto</i>
<b>occurrencesOf: unObjeto</b>	Retorna la cantidad de elementos de la colección que son iguales a <i>unObjeto</i>



### Ejemplo 1:

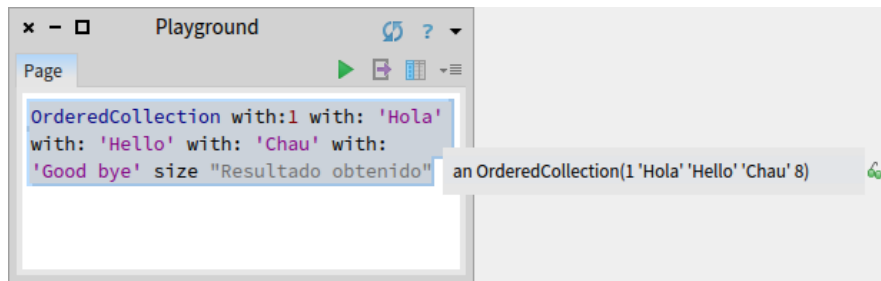
```
| unaColeccion |  
unaColeccion := OrderedCollection new.  
unaColeccion isEmpty      Respuesta: true  
unaColeccion size         Respuesta: 0  
unaColeccion := #( 2 'Hola').  
unaColeccion size         Respuesta: 2  
unaColeccion includes: 2  Respuesta: true  
unaColeccion includes: 5  Respuesta: false  
unaColeccion := Array with:1 with:5 with:2 with:5.  
unaColeccion occurrencesOf: 5  Respuesta: 2
```

### Ejemplo 2:



De la última cadena de caracteres agregada a la colección, el mensaje `size` devuelve su cantidad de caracteres.

### Ejemplo 3:



### Ejemplo 4 (utilizando variables):



```
| coleccion |  
coleccion := OrderedCollection with:1 with: 'Hola' with: 'Hello'.  
Transcript show: coleccion size.
```

3

Utilizando variables, el mensaje **size** devuelve la cantidad de elementos que posee la colección.

### 3.9.8.3 AÑADIR Y REMOVER OBJETOS

<b>Mensaje</b>	
<b>add:</b> unObjeto	Agrega <i>unObjeto</i> a la colección
<b>addAll:</b> otraColeccion	Agrega <i>otraColeccion</i> a la colección receptora
<b>remove:</b> unObjeto	Elimina el primer <i>unObjeto</i> que encuentra en la colección. Si <i>unObjeto</i> no está incluido en la colección, genera un error
<b>remove:</b> unObjeto <b>ifAbsent :</b> unBloque	Elimina <i>unObjeto</i> de la colección. Si <i>unObjeto</i> no está incluido en la colección, ejecuta <i>unBloque</i>
<b>removeAll:</b> otraColeccion	Elimina de la colección receptora los objetos incluidos en <i>otraColeccion</i>

Nota: Estos mensajes son ilegales en instancias de `FixedSizeCollection`.

#### Ejemplo 1:

```
| unaColeccion otraColeccion |  
unaColeccion := OrderedCollection new.  
unaColeccion add: 10; add: 4.  
otraColeccion := #('Hola' 10).  
unaColeccion addAll: otraColeccion.  
Transcript show: 'Elementos de la colección ', unaColeccion asString.
```



```
Transcript
Elementos de la colección an OrderedCollection(10 4 'Hola' 10)
```

### Ejemplo 2:

```
Playground
Page
| unaColeccion otraColeccion |
unaColeccion := OrderedCollection new.
unaColeccion add: 10; add: 4.
otraColeccion := #('Hola' 10).
unaColeccion addAll: otraColeccion.
unaColeccion remove: 10.
Transcript show: 'Elementos de la colección ', unaColeccion asString.
```

```
Transcript
Elementos de la colección an OrderedCollection(4 'Hola' 10)
```

### Ejemplo 3:

```
Playground
Page
| unaColeccion otraColeccion |
unaColeccion := OrderedCollection new.
unaColeccion add: 10; add: 4.
otraColeccion := #('Hola' 10).
unaColeccion addAll: otraColeccion.
unaColeccion remove: 'Good bye' ifAbsent: [Transcript show: 'Elemento
inexistente'].
Transcript show: String cr, 'Elementos de la colección ', unaColeccion
asString.
```



```
Transcript
Elemento inexistente
Elementos de la colección an OrderedCollection(10 4 'Hola' 10)
```

### 3.9.8.4 RECORRIDO

<b>Mensaje</b>	
<b>do:</b> unBloque	Evalúa <i>unBloque</i> para cada elemento de la colección, pasándole cada elemento de la colección como parámetro a <i>unBloque</i>
<b>select:</b> unBloque	Evalúa <i>unBloque</i> para cada elemento de la colección. Retorna una nueva colección conteniendo todos aquellos elementos que hicieron que <i>unBloque</i> evaluara en <b>true</b>
<b>reject:</b> unBloque	Evalúa <i>unBloque</i> para cada elemento de la colección. Retorna una nueva colección conteniendo todos aquellos elementos que hicieron que <i>unBloque</i> evaluara en <b>false</b>
<b>collect:</b> unBloque	Evalúa <i>unBloque</i> para cada elemento de la colección. Retorna una nueva colección conteniendo el resultado de las sucesivas evaluaciones del bloque
<b>detect:</b> unBloque	Evalúa <i>unBloque</i> para cada elemento de la colección. Retorna el primer elemento que haga que <i>unBloque</i> evalúe en <b>true</b> . Si ningún elemento cumple la condición, provoca un error
<b>detect:</b> <i>unBloque</i> <b>ifNone:</b> <i>bloqueExcepcion</i>	Evalúa <i>unBloque</i> para cada elemento de la colección. Retorna el primer elemento que haga que <i>unBloque</i> evalúe en <b>true</b> . Si ningún elemento cumple la condición, <i>bloqueExcepcion</i> es evaluado
<b>inject:</b> objetoInicial <b>into:</b> unBloqueBinario	Evalúa <i>unBloqueBinario</i> para cada elemento de la colección. Este bloque tiene dos parámetros: el segundo es el elemento de la colección receptora del mensaje; el primero es el resultado de la evaluación de <i>unBloqueBinario</i> en la intervención anterior. La primera vez es objetoInicial. Retorna el valor <i>unBloqueBinario</i> en la última iteración.



**Ejemplo 1:** “Contar cuantas veces figura la letra A en la frase: Hola, que tal?”

```
Playground
Page
| frase cuenta |
cuenta := 0.
frase := 'Hoy es un bello día'.
frase do: [ :letra | (letra asUppercase = $0)
            ifTrue: [ cuenta := cuenta + 1. ]
          ].
Transcript show: 'La cantidad de veces que se contó "', $0 asString, "
u "', $0 asString, " en la frase '", frase , ', fueron: ', cuenta
asString , ' veces'.
```

```
Transcript
La cantidad de veces que se contó "0" u "o" en la frase "Hoy es un bello día", fueron: 2 veces
```

**Ejemplo 2:** “Encontrar los números pares del arreglo (1 2 3 4 5 6 7 8 9 10)”

```
Playground
Page
| unArreglo pares |
unArreglo := #(0 1 2 3 4 5 6 7 8 9 ).
pares := unArreglo select: [ :nro | nro even ].
Transcript show: 'El contenido de la colección de pares es: ', pares
asString.
```

```
Transcript
El contenido de la colección de pares es: #(0 2 4 6 8)
```





### 3.9.9 DICTIONARY

Los diccionarios<sup>6</sup> (Dictionary) son un tipo especial de colecciones, ya que sus elementos son pares de claves y valores. Los objetos que referencia esta colección son instancias de una clase particular: Association. En una Association se tienen dos objetos, uno que juega el rol de clave y el otro el valor. En particular el objeto nil no puede ser la clave de una asociación de un diccionario.

Con los diccionarios se pueden representar obviamente diccionarios de palabras o sinónimos, pero también se pueden representar cosas interesantes como una agenda, donde cada elemento del diccionario es una letra, y el valor asociado son todos los contactos que empiezan con esa letra.



Otra forma de ver a un diccionario es como una lista, donde el acceso a los elementos no se hace por índice entero sino por cualquier otro objeto. De hecho, el mensaje para agregar nuevos elementos al diccionario es at: put:, pero en este caso el primer parámetro puede ser cualquier objeto. Veámoslo gráficamente con un ejemplo de código:

```
dic := Dictionary new.

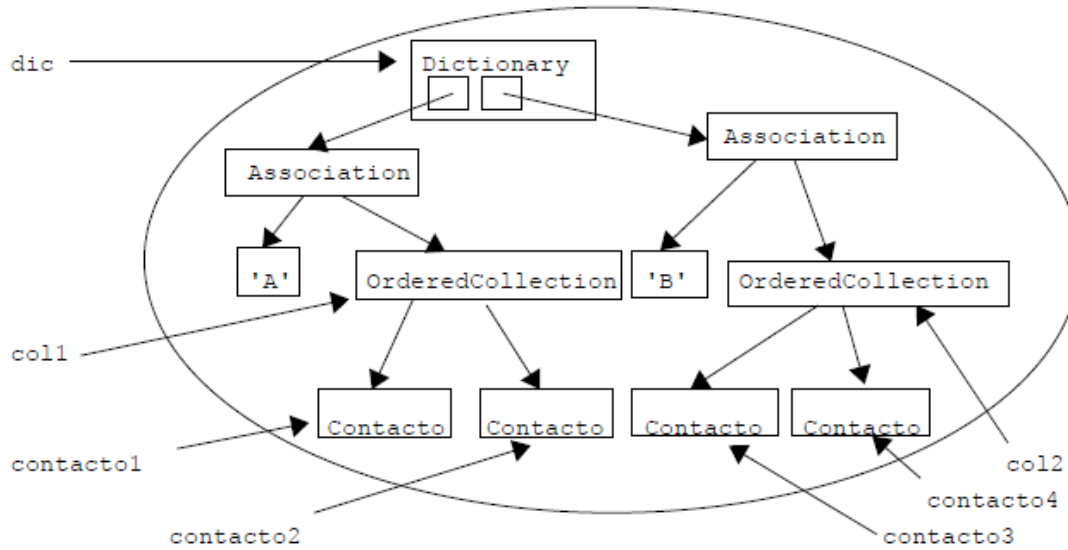
col1 := OrderedCollection new.
contacto1 := Contacto new: 'Armendariz Garcia' nombre: 'Joaquín'
nro:'8122967'.
contacto2 := Contacto new: 'Andrade Cantu' nombre: 'Rolando' nro:'913-3122'.
col1 add: contacto1.
col1 add: contacto2.

dic at: 'A' put: col1.

col2 := OrderedCollection new.
contacto3 := Contacto new: 'Blanco Castillo' nombre: 'Juan' nro:'.
contacto4 := Contacto new: 'Barrera Bravo' nombre: 'Felipe' nro:'8120298'.
col2 add: contacto3.
col2 add: contacto4.

dic at: 'B' put: col2.
```

<sup>6</sup> Basado en el apunte "Colecciones en Smalltalk" de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires (cátedra de Paradigmas de Programación), de Victoria Pocladova, Carlos Lombardi, Leonardo Volinier y Jorge Silva.



Observar que el diccionario crea una instancia de Association y que cada instancia referencia a la clave y valor que se agregaron al diccionario.

Ejemplo: una tabla de traducción que asocie palabras en castellano con su equivalente inglés, tal como la siguiente:

Clave	Valor
Auto	Car
Mesa	Table
Perro	Dog
Gato	Cat
Azul	Blue

puede implementarse por medio de una instancia de Dictionary:



```
Playground
Page
| unDiccionario |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
at: #Mesa put: #Table;
at: #Perro put: #Dog;
at: #Gato put: #Cat;
at: #Azul put: #Blue.
Transcript show: 'Elementos de la colección: ', String cr, unDiccionario asString.

Transcript
Elementos de la colección:
a Dictionary(#Auto->#Car #Azul->#Blue #Gato->#Cat #Mesa->#Table #Perro->#Dog )
```

A diferencia del resto de las colecciones, los diccionarios definen métodos que permiten realizar operaciones sobre las asociaciones, sobre los objetos que son clave y sobre los objetos que son valor.

### 3.9.9.1 INSERCIÓN Y REMOCIÓN DE ELEMENTOS

Los mensajes para agregar y eliminar elementos son los siguientes:

<i>Mensaje</i>	
<b>at:</b> unaClave <b>put:</b> unValor	Inserta <i>unValor</i> en el diccionario y lo asocia a <i>unaClave</i>
<b>removeKey:</b> unaClave	Elimina el valor asociado a <i>unaClave</i> . Si <i>unaClave</i> no figura en el diccionario, produce un error
<b>removeKey:</b> unaClave <b>ifAbsent:</b> unBloque	Elimina el valor asociado a <i>unaClave</i> . Si <i>unaClave</i> no figura en el diccionario, ejecuta <i>unBloque</i>

#### Ejemplo 1:

```
Playground
Page
| unDiccionario |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
at: #Verdadero put: #True;
at: #Perro put: #Dog;
at: #'5' put: #Cinco;
at: #Arreglo put: #(1 2 3 4 5 ).
Transcript show: 'Elementos de la colección: ', String cr, unDiccionario asString.
```



```
Transcript
Elementos de la colección:
a Dictionary{#5'->#Cinco #Arreglo->#(1 2 3 4 5) #Auto->#Car #Perro->#Dog #Verdadero->#True }
```

### Ejemplo 2:

```
Playground
Page
| unDiccionario |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
at: #Verdadero put: #True;
at: #Perro put: #Dog;
at: #'5' put: #Cinco;
at: #Arreglo put: #(1 2 3 4 5).
unDiccionario removeKey: '5' ifAbsent: [ Transcript show: 'El elemento con clave "5", no se encuentra en la colección' ].
Transcript show: 'Elementos de la colección: ', String cr, unDiccionario asString.
```

```
Transcript
Elementos de la colección:
a Dictionary{#Arreglo->#(1 2 3 4 5) #Auto->#Car #Perro->#Dog #Verdadero->#True }
```

### Ejemplo 3:

```
Playground
Page
| unDiccionario |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
at: #Verdadero put: #True;
at: #Perro put: #Dog;
at: #Arreglo put: #(1 2 3 4 5).
unDiccionario removeKey: '5' ifAbsent: [ Transcript show: 'El elemento con clave "5", no se encuentra en la colección' ].
Transcript show: String cr, 'Elementos de la colección: ', String cr, unDiccionario asString.
```

```
Transcript
El elemento con clave "5", no se encuentra en la colección
Elementos de la colección:
a Dictionary{#Arreglo->#(1 2 3 4 5) #Auto->#Car #Perro->#Dog #Verdadero->#True }
```



### 3.9.9.2 RECUPERACIÓN DE ELEMENTOS

Los mensajes básicos para la recuperación de los elementos de un Dictionary son los siguientes:

<i>Mensaje</i>	
<b>at:</b> unaClave	Devuelve el valor asociado a <i>unaClave</i> . Si la clase no figura en el diccionario, genera un error
<b>at:</b> unaClave <b>ifAbsent:</b> unBloque	Devuelve el valor asociado a <i>unaClave</i> . Si la clase no figura en el diccionario, se ejecuta <i>unBloque</i>
<b>keyAtValue:</b> unObjeto	Busca si entre los valores figura <i>unObjeto</i> y si lo encuentra, devuelve la clave de la que está asociado. En caso contrario, devuelve <b>nil</b>
<b>keyAtValue:</b> unObjeto <b>ifAbsent:</b> unBloque	Busca si entre los valores figura <i>unObjeto</i> y si lo encuentra, devuelve la clave de la que está asociado. En caso contrario, ejecuta <i>unBloque</i>
<b>Keys</b>	Retorna una colección conteniendo todas las claves del diccionario
<b>Values</b>	Retorna una colección conteniendo todos los valores del diccionario

Por ejemplo:

```
Playground
Page
| unDiccionario |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
              at: #'5' put: #Cinco;
              at: #Perro put: #Dog;
              at: #Arreglo put: #(1 2 3 4 5).
Transcript show: 'Elementos de la colección: ', String cr, unDiccionario asString.
```

```
Transcript
Elementos de la colección:
a Dictionary[ #'5'->#Cinco #Arreglo->#(1 2 3 4 5) #Auto->#Car #Perro->#Dog ]
```

**Ejemplo 1:** unDiccionario at: '5' => #cinco'



```
Playground  
Page  
| unDiccionario |  
unDiccionario := Dictionary new.  
unDiccionario at: #Auto put: #Car;  
                at: #'5' put: #Cinco;  
                at: #Perro put: #Dog;  
                at: #Arreglo put: #(1 2 3 4 5 ).  
Transcript show: (unDiccionario at: #'5').  
  
Transcript  
Cinco
```

**Ejemplo 2:** unDiccionario at: 'Hola' => ERROR!!

```
Playground  
Page  
| unDiccionario |  
unDiccionario := Dictionary new.  
unDiccionario at: #Auto put: #Car;  
                at: #'5' put: #Cinco;  
                at: #Perro put: #Dog;  
                at: #Arreglo put: #(1 2 3 4 5 ).  
Transcript show: (unDiccionario at: 'Nuevo').  
  
KeyNotFound: key 'Nuevo' not found in Dictionary  
Proceed Abandon Debug Report  
Dictionary errorKeyNotFound:  
Dictionary at: [ self errorKeyNotFound: key ]  
Dictionary at:ifAbsent:  
Dictionary at:  
UndefinedObject Dolt  
OpalCompiler evaluate  
RubSmalltalkEditor evaluate:andDo:
```

**Ejemplo 3:** unDiccionario keyAtValue: true => #verdadero



The screenshot shows a Playground window with the following code:

```
| unDiccionario |  
unDiccionario := Dictionary new.  
unDiccionario at: #Auto put: #Car;  
                at: #'5' put: #Cinco;  
                at: #Perro put: #Dog;  
                at: #Arreglo put: #(1 2 3 4 5 ).  
Transcript show: (unDiccionario keyAtValue: #Cinco).
```

The Transcript window below shows the output: 5

**Ejemplo 4:** unDiccionario keys => (5 #Auto #Perro #Arreglo)

The screenshot shows a Playground window with the following code:

```
| unDiccionario |  
unDiccionario := Dictionary new.  
unDiccionario at: #Auto put: #Car;  
                at: #'5' put: #Cinco;  
                at: #Perro put: #Dog;  
                at: #Arreglo put: #(1 2 3 4 5 ).  
Transcript show: unDiccionario keys. |
```

The Transcript window below shows the output: #(#Arreglo #Perro #Auto #'5')

**Ejemplo 5:** unDiccionario values => (#Cinco #Car #Dog #(1 2 3 4 5))



The screenshot shows a 'Playground' window with the following code:

```

| unDiccionario |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
               at: #'5' put: #Cinco;
               at: #Perro put: #Dog;
               at: #Arreglo put: #(1 2 3 4 5).
Transcript show: unDiccionario values.
  
```

Below the code is a 'Transcript' window showing the output:

```

#(0#(1 2 3 4 5) #Dog #Car #Cinco)
  
```

Dentro de un diccionario, los pares clave-valor se almacenan en instancias de la clase **Association**. [Cuenca]. Las asociaciones pueden recuperarse con los siguientes mensajes:

<i>Mensaje</i>	
<b>associationAt:</b> unaClave	Devuelve la asociación que tiene como clave a <i>unaClave</i> . Si la clave no figura en el diccionario, genera un error
<b>associationAt:</b> unaClave <b>ifAbsent:</b> unBloque	Devuelve la asociación que tiene como clave a <i>unaClave</i> . Si la clave no figura en el diccionario, ejecuta <i>unBloque</i>

Una vez obtenida la asociación, pueden recuperarse la clave y el valor enviándole los mensajes **key** y **value** respectivamente.

**Ejemplo 1:**

The screenshot shows a 'Playground' window with the following code:

```

| unDiccionario unaAsociacion |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
               at: #Verdadero put: #True;
               at: #Perro put: #Dog;
               at: #'5' put: #Cinco;
               at: #Arreglo put: #(1 2 3 4 5).
unaAsociacion := unDiccionario associationAt: 'Perro'.
Transcript show: 'Asociación: ', unaAsociacion asString.
  
```





```
Transcript
Asociación: #Perro->#Dog
```

**Ejemplo 2:**

```
Playground
Page
| unDiccionario unaAsociacion |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
              at: #Verdadero put: #True;
              at: #Perro put: #Dog;
              at: #'5' put: #Cinco;
              at: #Arreglo put: #(1 2 3 4 5 ).
unaAsociacion := unDiccionario associationAt: 'Perro'.
Transcript show: 'La clave de asociación es: ', unaAsociacion key asString.
```

```
Transcript
La clave de asociación es: Perro
```

**Ejemplo 3:**

```
Playground
Page
| unDiccionario unaAsociacion |
unDiccionario := Dictionary new.
unDiccionario at: #Auto put: #Car;
              at: #Verdadero put: #True;
              at: #Perro put: #Dog;
              at: #'5' put: #Cinco;
              at: #Arreglo put: #(1 2 3 4 5 ).
unaAsociacion := (unDiccionario associationAt: 'Perro') value.
Transcript show: 'El valor de la asociación de "Perro" es: ', unaAsociacion
asString.
```

```
Transcript
El valor de la asociación de "Perro" es: Dog
```



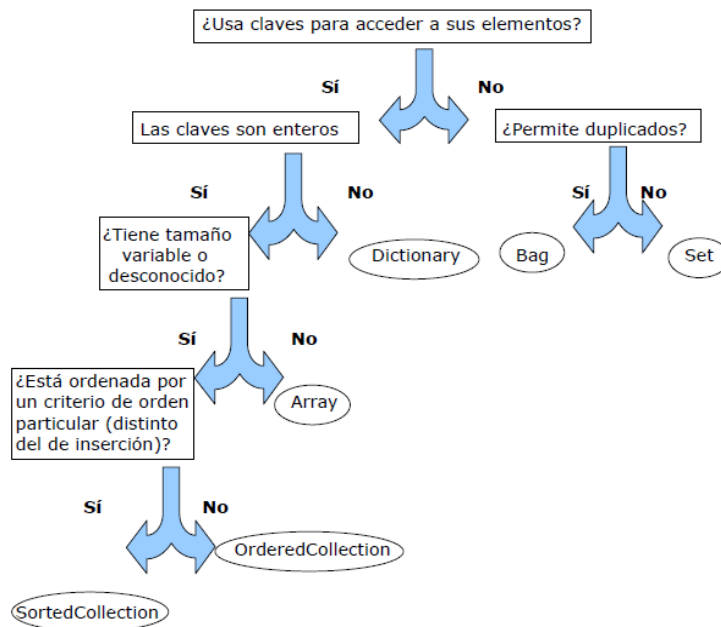
### 3.9.9.3 RECORRIDO

En adición a las operaciones de recorrido heredadas de **Collection**, **Dictionary** ofrece además los siguientes servicios:

<i>Mensaje</i>	
<b>associationsDo:</b> unBloque	Para cada asociación, evaluar <i>unBloque</i> con esa asociación como parámetro
<b>associationsSelect:</b> unBloque	Para cada asociación, evaluar <i>unBloque</i> con esa asociación como parámetro.  Retorna otra colección conteniendo las asociaciones que hicieron que <i>unBloque</i> evaluara en <b>true</b>
<b>keysDo:</b> unBloque	Para cada una de las claves del diccionario, evaluar <i>unBloque</i> con esa clave como argumento

### 3.9.10 CÓMO ELEGIR LA COLECCIÓN MÁS ADECUADA?

A continuación se muestra un algoritmo posible a seguir para seleccionar la colección más adecuada de acuerdo al problema a modelar. Sin embargo, la elección de la colección debería hacerse de manera intuitiva, considerando las características de cada una de las opciones disponibles.





## 4. ANEXO I: EJEMPLO

### EJEMPLO: LA GOLONDRINA

A continuación realizaremos un ejemplo incremental, el cuál será desarrollado paso a paso de manera de reforzar lo visto hasta el momento.

La idea Original del presente ejemplo está tomada de Apunte: Introducción a la orientación a objetos de la Asignatura Paradigmas de Programación UTN – FRBA Autores: Carlos Lombardi – Nicolás Passerini

Este ejemplo ha sido ajustado a la teoría explicitada en el presente apunte y ajustado a la implementación en Samalltalk Pharo 5.0.

#### **Ejemplo: Doña Pepita**

Pongamos como objeto a Pepita, una golondrina tijerita; y analicemos la conducta de distintos observadores.

A un ornitólogo le puede interesar p.ej. a qué velocidad vuela, qué come, cómo digiere, cuánto pesa, cuáles son sus patrones de consumo de energía, cuáles son sus ritos de apareamiento; las interacciones (lo que va a hacer el ornitólogo con la golondrina) van a ser pesarla, medirla, analizar su sangre, ponerle comida y esperar que coma, llevar un registro con los momentos en que empieza y para de volar, medir su velocidad, ponerle un anillo de reconocimiento.

A un fotógrafo de la naturaleza le pueden interesar de Pepita cosas como los colores, el brillo, a qué horas aparece, también los ritos de apareamiento; las interacciones van a ser esperarla, sacarle fotos, atraerla.

Un zorro con hambre claramente va a tener otra perspectiva acerca de Pepita: le van a interesar más que nada el peso y la velocidad; y sus interacciones van a ser ... bastante primitivas, perseguirla, cazarla y comérsela.

Resumiendo: a distintos observadores les van a interesar distintas características y formas de interacción con los mismos objetos.

Entonces, de todos los objetos con los que podríamos interactuar, nos quedamos con algunos; y de todas las formas de interactuar con cada objeto, nos quedamos con algunas.

Estas operaciones de recorte forman el modelo que el observador se hace de lo que quiere observar.

Armar modelos y trabajar en base a ellos es la forma que encontramos las personas para poder pensar acerca del mundo complejo en el que vivimos (lo que hablábamos al principio).

¿En qué consiste, dónde está, el modelo que el ornitólogo se hace de Pepita?

En el montón de notas, fotos, cuadros, etc., que tomó a partir de sus observaciones; y en lo que quedó en su cabeza que le permite pensar acerca de Pepita.

El modelo es la representación que el ornitólogo se hizo de Pepita; no es Pepita ni está en ella.

Yo, que soy una persona, digo

“Pepita vuela moviendo sus alas como buena golondrina que es”



A Pepita no le hacen falta los conceptos de “volar”, “ala” ni “golondrina” para hacer eso que yo llamo “volar”.

Resumiendo: el modelo está separado del objeto, queda del lado del observador. El observador genera una representación de los objetos con los que interactúa.

Y acá es donde entra el software. Los objetos de la orientación a objetos son las representaciones computacionales de los entes con los que vamos a interactuar.

Si hago una aplicación para que el ornitólogo registre la información que va obteniendo y genere reportes y otros resultados, sí voy a tener en esa aplicación un objeto que representa a Pepita. Si un usuario quiere saber cuánto voló Pepita entre el 21 y el 25 de febrero, su velocidad standard de vuelo, su peso, cuál fue el vuelo más largo que hizo, etc.; se lo va a pedir al objeto que representa a Pepita.

Cuando el ornitólogo quiera registrar un vuelo de Pepita, el resultado de un análisis, el código de su anillo identificatorio; es probable que lo haga pidiéndole a ese objeto que registre la información.

El código que voy a escribir es el que va a implementar el objeto que representa a Pepita, de forma tal de poder registrar todo lo que resulta de las interacciones del ornitólogo, y poder responder correctamente a las consultas que le puedan interesar al mismo ornitólogo o a otro usuario.

Concentrémonos ahora en el primer pedido: cuánto voló Pepita entre el 21 y el 25 de febrero. Supongamos que la respuesta es 32 km.

Para poder formular la pregunta, el usuario tiene que poder representar de alguna forma las dos fechas y/o el período entre ambas. Sí, adivinaste: va a haber objetos que representen las fechas, y objetos que representen períodos de tiempo.

La respuesta también va a estar dada por un objeto, que representa la magnitud “32 km”, y que está a su vez compuesto por dos, uno que representa al número 32 y el otro a la unidad de medida kilómetro.

Si le pido el código del anillo identificatorio, me va a responder un código, p.ej. “AZQ48”. Si ya tenés claro que hay un objeto que representa al código, y también hay un objeto que representa a cada letra, le estás tomando la onda.

Resumiendo: además de los objetos que representan los entes que tienen sentido para los usuarios (en este caso Pepita), aparecen muchos objetos más que representan conceptos más básicos (fechas, distancias, números, códigos, texto, unidades de medida. Cualquier ente que quiera representar, lo tengo que representar mediante un objeto; la orientación a objetos no me da otras formas.

A partir de ahora, en vez de decir “el objeto que representa a X” vamos a decir “el objeto X”. P.ej. vamos a hablar del objeto Pepita, los objetos fecha, los objetos número, etc.

Los objetos con los cuales decida trabajar van a conformar el modelo y los vamos a elegir en base al criterio: Que tenga sentido interactuar con ellos de alguna forma.

Por lo expuesto hasta el momento, claramente va a existir un objeto Pepita; es aquel con el que va a interactuar el ornitólogo en las formas que describimos más arriba. A una fecha le puedo pedir el mes, qué día de la semana es, si eso no anterior a otra fecha, y otras cosas; entonces es probable que aparezcan objetos fecha. Lo mismo con una distancia (le puedo pedir que se sume a otra distancia, que me diga a cuántos metros equivale), un número (le puedo pedir que se multiplique con otro número), un texto (le puedo pedir la cantidad de letras, si aparece o no cierta letra).

A las interacciones vistas como una unidad las vamos a llamar comportamiento; es la palabra que se usa en la literatura sobre orientación a objetos.



Si le quiero indicar al objeto Pepita que Pepita (la golondrina de verdad) comió 30 gramos de alpiste (lo que el objeto Pepita necesita saber porque eso aumenta su energía), el “30” (o el “30 gramos”) va a ser un parámetro. Si no, el objeto Pepita debería entender un mensaje “comiste 30 gramos de alpiste”, otro distinto “comiste 50 gramos de alpiste”, y así hasta el infinito

Puede haber un resultado, que es la respuesta que obtiene el emisor. P.ej. si le pido al objeto Pepita la energía, un resultado posible es “48 joules”.

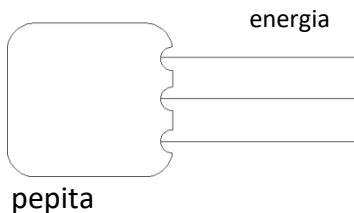
Está claro que tanto los parámetros como el resultado son objetos, en los ejemplos de recién el parámetro es el objeto 30 (o “30 gramos”) y el resultado es el objeto 48 (o “48 joules”).

Hasta acá queda claro que si el ornitólogo quiere saber cuánta energía tiene Pepita en un determinado momento, lo que tiene que hacer es enviarle el mensaje energía al objeto pepita. Bajemos un poco a detalle.

El ornitólogo trabaja en un entorno Smalltalk. En el entorno en el que está trabajando, el ornitólogo tuvo que haber configurado al objeto Pepita.

Para poder interactuar directamente con un objeto, se le pone un nombre dentro del entorno en el que estoy trabajando; p.ej. pepita. En Smalltalk los nombres de los objetos empiezan en minúscula, salvo excepciones que veremos más adelante.

La configuración va a incluir pasarle a pepita las observaciones relevantes; si lo que le interesa es ir viendo la evolución de la energía, le va a pasar los registros de los eventos que la afectan, p.ej. las veces que comió (que aumentan su energía) y las veces que voló (que la disminuyen).



Después de esto, sólo queda enviarle a pepita el mensaje energía. En código, esto se hace así:

```
pepita energia
```

En Smalltalk podemos (y lo vamos a hacer): abrir un entorno de trabajo, configurar ahí a pepita, después escribir lo que dice arriba, pintarlo y pedirle “mostrame el resultado de esto”, y anda.

La sintaxis de Smalltalk para enviarle un mensaje a un objeto es:

### objeto mensaje

Donde “objeto” es el nombre que le di al objeto en el entorno en donde estoy escribiendo el código. Así se entiende el ejemplo de arriba; pepita es el (nombre del) objeto, energía es el (selector del) mensaje.

Y va exactamente así, sin puntos ni paréntesis ni nada.

Volviendo: cuando el ornitólogo envía el mensaje pepita energia ¿qué pasa? Obviamente, pasa que se ejecuta código.



¿Qué código? Ahí vamos. Para que el ornitólogo pueda interactuar con pepita, alguien la tuvo que programar; donde “programar” quiere decir: indicar qué mensajes va a entender pepita, y para cada uno escribir el código que se va a ejecutar cuando pepita lo reciba.

O sea: fue un programador el que decidió que pepita va a entender el mensaje energía, y escribió el código asociado a ese mensaje. Ese código es el que se va a ejecutar cuando el ornitólogo ejecute pepita energía.

La sección de código que se asocia a un mensaje se llama método. Un método tiene un nombre, que es el del mensaje correspondiente; y un cuerpo, que es el código que se ejecuta.

Un detalle: en la jerga de objetos, se usa mucho el término “evaluar” en vez de “ejecutar”.

Lo que no vimos es dónde se escriben los métodos. Para eso falta un rato.

Antes de pasar a otro tema; arriba dijimos “el ornitólogo tuvo que haber configurado al objeto Pepita”. ¿Cómo hizo? Obviamente enviándole los mensajes que le dicen a pepita lo que le pasó a Pepita. Dicho en general, los que le indican a la representación computacional lo que le pasó al objeto representado.

Insistimos, la única forma que tengo de interactuar con un objeto es enviándole mensajes. P.ej. para indicarle que Pepita comió 30 gramos de alpiste una forma posible es

pepita comioAlpiste: 30

Los nombres de mensaje con un parámetro terminan con dos puntos, eso indica que atrás viene un parámetro; los dos puntos forman parte del nombre. La sintaxis de envío es

### **objeto mensaje parametro**

Acá estamos suponiendo que pepita entiende el mensaje comioAlpiste:, y que el parámetro es un número que expresa la cantidad en gramos.

En el apunte vamos a ir introduciendo más elementos de la sintaxis de Smalltalk.

Vamos a programar una versión muy simplificada de pepita. Los que queremos que pepita haga es:

- lo único que quiero es poder saber en cada momento cuánta energía tiene pepita,
- los únicos eventos relevantes son comer alpiste y volar,
- el alpiste le brinda 4 joules por cada gramo que come,
- en cada vuelo consume 10 joules de “costo fijo” (es lo que le cuesta arrancar), más un joule por cada minuto que vuela,

Para hacerla fácil vamos a representar los joules, gramos y minutos con números, p.ej. la cantidad “30 joules” la vamos a representar con el número 30.

Los mensajes que va a entender pepita (Protocolo) son:

- energía, que devuelve la energía expresada en joules
- comioAlpiste:, que registra la ingestión de alpiste; el parámetro es la cantidad expresada en gramos

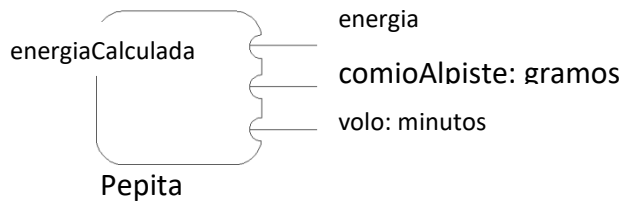


- volo: que registra un vuelo; el parámetro es la duración expresada en minutos (no vamos a tener en cuenta los errores, como que la energía quede negativa después de volar)

Para implementar este comportamiento, hagamos que pepita recalcule la energía después de cada acción; cuando le preguntan la energía, devuelve la que tiene calculada.

¿Dónde guardo la energía calculada? Ahí vamos. A cada objeto le puedo asociar un conjunto de variables (colaboradores internos), cuyo valor vive entre envíos de mensajes al mismo objeto.

Nuestro Modelo de Objetos de pepita , queda:



Volviendo, con el planteo que hicimos alcanza con que pepita tenga una sola variable (un solo colaborador interno), llamémosla energíaCalculada.

El código para pepita es como sigue, indentando para marcar las distintas partes

#### **Pepita**

##### variables

```
energiaCalculada
```

##### métodos

```
comioAlpiste: gramos
  "Le indican a pepita que comió alpiste; el parámetro es la cantidad
  expresada en gramos"
  energiaCalculada := energiaCalculada + (gramos * 4)
volo: minutos
  "Le indican a pepita que voló; el parámetro es la duración del vuelo
  expresada en minutos"
  energiaCalculada := energiaCalculada - (10 + minutos)
energia
  "Devuelve la energía del receptor medida en joules"
  ^energiaCalculada
reset
  "Vuelve pepita a un estado inicial"
  energiaCalculada := 0
```



La sintaxis es la de Smalltalk; resaltamos algunos elementos

- la asignación es con :=
- los comentarios van entre comillas dobles
- para indicar el resultado de un método se pone ^resultado (el circunflejo es análogo al “return” en otros lenguajes).

Agregamos reset al comportamiento de pepita para poder hacer muchas pruebas empezando siempre de 0.

Una observación interesante: el selector para pedirle la energía a pepita es energía, no dameTuEnergía, getEnergía u otras variantes. Es la convención Smalltalk, los selectores que piden info tienen el nombre de lo que piden sin ningún agregado; en principio conviene adherir a las convenciones del lenguaje/entorno que usamos. Además, se escribe menos y el código queda más legible.

#### Lo implementamos ahora en Smalltalk Pharo

Ahora usemos a pepita. En Smalltalk Pharo, esto se puede hacer creando un entorno en el cual se interactúa con el objeto (obviamente, enviándole mensajes). Ese entorno es una ventana de texto; el texto se escribe, se pinta y se evalúa; si hay un resultado, se muestra en la misma ventana. Los envíos de mensajes se separan con un punto decimal, como si cada uno fuera una oración.

Supongamos que la simulación que le interesa al ornitólogo es esta secuencia: comer 50 gramos de alpiste, volar 5 minutos, comer otros 30 gramos, volar 10 minutos.

```
|pepita|
pepita := Pepita new.
pepita reset.
pepita comioAlpiste: 50.
pepita volo: 5.
pepita comioAlpiste: 30.
pepita volo: 10.
```

Si bien hasta acá no obtenemos ningún resultado interesante, la evolución quedó en el estado interno de pepita, a la que ahora le podemos pedir la energía pepita.

```
pepita energia.
```

Si queremos visualizar los resultados, usaremos la ventana Transcript y evaluamos en Playground las líneas anteriores. Ahora sí me va a dar un resultado interesante, que es 285 como esperábamos.





```
Playground
Page
|pepita|
pepita := Pepita new.
pepita reset.
pepita comioAlpiste: 50.
pepita volo: 5.
pepita comioAlpiste: 30.
pepita volo: 10.
Transcript show: pepita energia asString.

Transcript
285
```

Veamos ahora el concepto de encapsulamiento

Otro programador podría haber implementado pepita con un estado de dos variables: una para la energía ingerida, y otra distinta para la energía consumida. Quedaría así:

**Pepita**

variables

```
energiaIngerida
energiaConsumida
```

métodos

```
comioAlpiste: gramos
  "Le indican a pepita que comió alpiste; el parámetro es la cantidad expresada en gramos"
  energiaIngerida := energiaIngerida + (gramos * 4)
volo: minutos
  "Le indican a pepita que voló; el parámetro es la duración del vuelo expresada en minutos"
  energiaConsumida := energiaConsumida - (10 + minutos)
energia
  "Devuelve la energía del receptor medida en joules"
  ^energiaIngerida - energiaConsumida
reset
  "Vuelve pepita a un estado inicial"
  energiaIngerida := 0
  energiaConsumida := 0
```



Esta implementación es bastante distinta a la anterior.

Ahora viene el ornitólogo y quiere hacer las mismas pruebas de la sección anterior, usando la nueva implementación de pepita.

Repasamos la parte de uso de pepita, ¿qué impacto tuvo el cambio de implementación en la forma de usar a pepita? Claramente ninguno, todos los mensajes que el ornitólogo le enviaba a la “vieja pepita” se los puede enviar a la “nueva pepita”, obteniendo los mismos resultados.

Digámoslo de nuevo en términos más generales.

Cambiamos bastante la implementación de un objeto, y el usuario de ese objeto no se enteró.

¿Cómo logramos esto? Porque no cambiamos el protocolo del objeto, o sea el conjunto de mensajes que entiende; y lo único que ve el usuario del objeto son los mensajes que entiende, no ve cómo el objeto lo implementa “adentro suyo” (métodos y estado interno).

Trabajando en Smalltalk, el ornitólogo no tiene forma de conocer la forma del estado interno de pepita, ni sus valores; y tampoco puede acceder a detalles sobre cuál es la implementación que está detrás de los mensajes que le envía. Lo único que conoce es el comportamiento que pepita exhibe; es lo único que necesita conocer para poder interactuar con el objeto.

Esta idea de que un observador no ve todos los aspectos de un objeto sino solamente aquellos que le sirven para interactuar con él se llama encapsulamiento; la idea del nombre es que los aspectos internos del objeto se encapsulan de forma tal que los observadores no pueden verlos.

La idea de encapsulamiento también puede observarse en la relación que nosotros tenemos con el mundo que nos rodea: para interactuar con una videocasetera me alcanza con entender el control remoto, no necesito (¡afortunadamente!) saber p.ej. por dónde pasan las correas que llevan el movimiento del motor a los cabezales; para darle de comer a una golondrina no necesito la composición química de sus jugos gástricos, y miles de etc.

Al encapsular un objeto, estoy al mismo tiempo acotando y explicitando (haciendo explícitas) las formas posibles de interacción; sólo se puede interactuar con un objeto mediante el comportamiento que exhibe. Esto nos da varias ventajas que pasamos a comentar.

Como las formas de interacción son acotadas y las maneja quien programa el objeto, se hace más sencillo probar si un objeto se comporta correctamente.

### Veamos ahora el concepto de Polimorfismo

Volvamos a la implementación de la viaja pepita, en donde teníamos un solo colaborador interno.

Además de pepita va a aparecer un picaflor llamado pepe. Al ornitólogo le interesa hacer con pepe lo mismo que con pepita: estudiar la evolución de su cantidad de energía a medida que come y que vuela. Obviamente, pepe tiene distintos patrones de consumo de energía:

- el alpiste le brinda 16 joules por gramo que come, pero cada vez que come hace un esfuerzo fijo que le demanda 45 joules.
- en cada vuelo consume 30 joules, que es lo que le cuesta arrancar y parar. Es tan liviano, que el costo de mantenerse en vuelo y de moverse es despreciable.



¿Qué mensajes queremos que entienda pepe? Una respuesta posible es:

Voy a hacer que pepe entienda los mismos mensajes que pepita, porque las formas de interacción del ornitólogo con los dos van a ser las mismas. El ornitólogo le va a enviar los mismos mensajes a los dos, lo que puede cambiar son las respuestas de cada uno.

Esta respuesta, esta forma de pensarlo, está perfecta. Si tu intuición te dijo esto (o algo así), vas por un camino muy bueno. Hay dos aspectos que destacamos:

- al pensar en un objeto, lo pienso desde las formas en que el observador va a querer interactuar con él. Dicho de otra forma, al configurar un objeto lo pienso desde los usuarios que va a tener ese objeto.
- si los patrones de interacción con dos objetos son los mismos, lo lógico es que tengan un vocabulario en común (Protocolo – Conjunto de mensajes). Yendo un poco más allá en esta idea, si hago esto le va a ser más fácil al observador hablar con uno o con el otro indistintamente.

Veamos cómo sería la codificación de pepe

## Pepe

### Variables

```
energiaCalculada
```

### métodos

```
comioAlpiste: gramos
```

```
"Le indican a pepe que comió alpiste; el parámetro es la cantidad expresada en gramos"
```

```
energiaCalculada := energiaCalculada + (gramos * 16) - 45
```

```
voló: minutos
```

```
"Le indican a pepe que voló; el parámetro es la duración del vuelo expresada en minutos"
```

```
energiaCalculada := energiaCalculada - 30
```

```
energia
```

```
"Devuelve la energía del receptor medida en joules"
```

```
^energiaCalculada
```

```
reset
```

```
"Vuelve pepe a un estado inicial"
```

```
energiaCalculada := 0.
```

Si quiero ejecutar la misma simulación con pepe en lugar de con pepita ¿qué tengo que cambiar en el código? Solamente el objeto al que le envío los mensajes. Al observador le es felizmente indistinto hablar con uno o con el otro, no necesita ninguna adaptación.



Ahora demos un paso más. El ornitólogo tiene preparados distintas simulaciones, que evalúan lo que pasaría con la energía de un ave en distintos escenarios; tiene datos de varias aves y quiere aplicarles los distintos escenarios a cada una y ver con cuánta energía quedan.

A esta altura está clarísimo que vamos a representar cada ave con la que quiere trabajar el ornitólogo mediante un objeto, y que todos estos objetos van a entender el mismo protocolo, los mensajes reset, comioAlpiste:, volo: y energia.

Observemos que además de trabajar con muchas aves, el ornitólogo quiere tener a mano muchas simulaciones. Entonces, de repente tiene interés de representar simulaciones así como representamos aves. Una vez decidido esto, debería quedarnos claro que van a aparecer objetos simulación, porque (otra vez) cualquier ente que quiera representar, lo voy a representar mediante un objeto.

¿Cómo va a interactuar el ornitólogo con un objeto simulación? Pasándole el (objeto) ave con el que quiere que trabaje, y después diciéndole sencillamente algo así como ... “dale masa”.

Ergo, los objetos simulación van a entender dos mensajes, a los que vamos a llamar trabajaCon: y daleMasa.

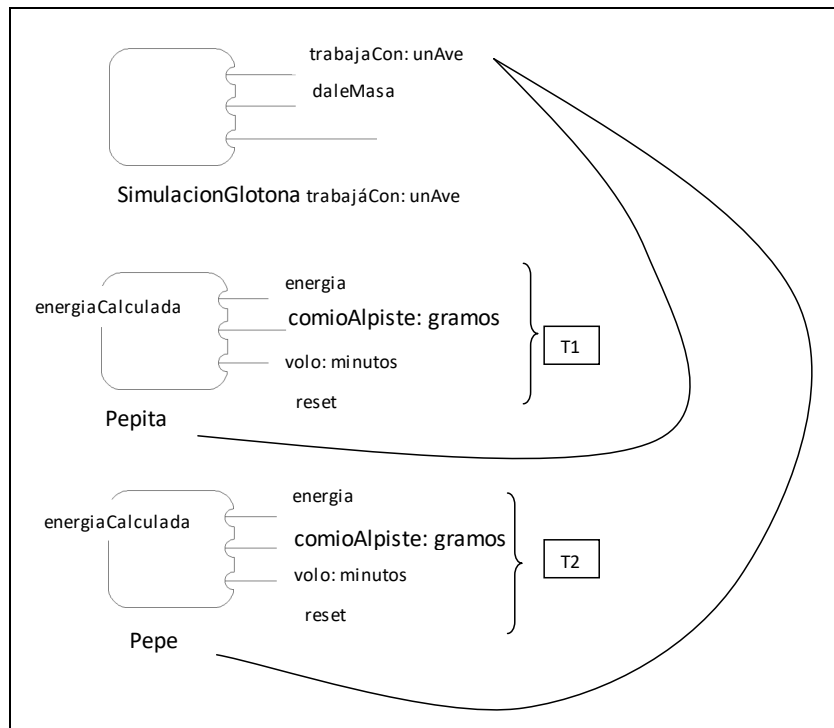
Empecemos armando una simulación en el cual los pájaros comen mucho, al que vamos a llamar simulaciónGlotona.

Pensemos un poco en cómo implementarlo.

En el método daleMasa vamos a indicarle al ave con la que estamos trabajando la secuencia de eventos propios de la simulación, y a devolver la energía que tiene el ave después de estos eventos.

La simulación necesita acordarse del objeto ave (colaborador externo) que le pasaron para trabajar, esto lo va a hacer en el método trabajaCon:.

El nuevo modelo de objetos lo podemos representar como:





Los TIPOS T1 y T2 que se corresponden con los protocolos de ambos Objetos; Pepita y Pepe son Polimorfos, lo que posibilita que en el ámbito del colaborador ave, pueden ser intercambiados.

El código de Simulación Glotona

### Simulación Glotona

#### variables

```
Ave
```

#### métodos

```
trabajaCon: unAve  
"Le ordenan trabajar con un ave determinada"  
ave := unAve  
  
daleMasa  
"Le ordenan ejecutar la simulacion"  
ave reset.  
ave comioAlpiste: 50.  
ave volo: 5.  
ave comioAlpiste: 30.  
ave volo: 10.  
^ave energia.
```

Ahora al ornitólogo le resulta mucho más fácil evaluar la simulación para un ave, p.ej. pepita. El código en la ventana de interacción queda así:

```
simulaciónGlotona trabajaCon: pepita.  
simulaciónGlotona daleMasa.
```

Para usarlo con pepe, el cambio es trivial:

```
simulaciónGlotona trabajaCon: pepe.  
simulaciónGlotona daleMasa.
```

Nota antes de seguir: observamos que el parámetro que recibe trabajáCon: es un ave. Queremos destacar esto para mostrar que cualquier objeto puede ser colaborador externo, no solamente los números u otros objetos "simples".

Volviendo, ¿tuvimos que cambiar el código de simulaciónGlotona para que trabaje con pepe o con pepita?



No, para la simulación trabajar con pepe o con pepita es exactamente lo mismo. De hecho, la idea es que para la simulación sea lo mismo trabajar con cualquier ave que yo quiera representar, eso tiene mucho interés porque puedo usar la misma simulación para todas las aves.

Obviamente para que esto pase, cada ave que represente tiene que tener un mismo protocolo, o lo que es lo mismo que entender los mensajes reset, comióAlpiste:, voló: y energía; porque si no en el método daleMasa le voy a estar enviando al ave un mensaje que no entiende.

De hecho, la simulación podrá interactuar con cualquier objeto que entienda estos cuatro mensajes, el que sea un ave o no a la simulación no le importa nada.

Lo que termina pasando es que la simulación puede interactuar con muchos objetos distintos, de los cuales lo único que conoce es cómo de interactuar con ellos, y sin tener que hacer ninguna modificación en la simulación para que trabaje con uno o con el otro.

A esta característica de poder trabajar con objetos distintos en forma transparente la llamamos polimorfismo; decimos que pepita y pepe son objetos polimórficos.

Ahora, ¿a quién le sirve que pepita y pepe sean polimórficos? ¿A pepita? ¿A pepe? ¡Claro que no! Le sirve al usuario de pepita y de pepe, que en este caso es es otro objeto, simulaciónGlotona.

#### Lo implementamos ahora en Smalltalk Pharo

```
pepita := Pepita new.  
pepita reset.  
pepe := Pepe new.  
pepe reset.  
simulacion := SimulacionGlotona new.  
simulacion trabajaCon: pepita.  
simulacion daleMasa.
```

Para usarlo con pepe:

```
simulacion trabajaCon: pepe.  
simulacion daleMasa.
```



## 5. BIBLIOGRAFÍA

- *Descubra Smalltalk*. Autor: Wilf LaLonde. Editorial: ADDISON-WESLEY / DIAZ DE SANTOS. Año: 1997.
  - *Inside Smalltalk*. Vol. II: Lalonde y J. Pugh; Prentice Hall International; 1991.
  - *Introducción a la Programación Orientada a Objetos*. Autor: Timothy Budd. Editorial: ADDISON-WESLEY IBEROAMERICANA. Año: 1994.
  - *Apunte Taller de Orientación a Objetos: Programación Orientada a Objetos en Smalltalk/V del Laboratorio de Sistemas-Universidad Tecnológica Nacional-Facultad Regional Córdoba*. Autor: Ing. Fernando Cuenca.
  - *Lenguajes de Programación*. Autor: Abdiel Cáceres Gonzalez. Instituto Tecnológico de Monterrey-Campus Ciudad de México. Año: 2004.
  - *Smalltalk-80 The Language and Its Implementation*. Autores: Adele Goldberg and David Robson. Editorial: Año: 1983.
  - *Notas de clase de Tópicos II, curso de la Maestría en Ingeniería de Software UNLP, dictada en 2002 por Máximo Prieto*. Documentos de Juan Carlos Vazquez.
  - *Apunte: Introducción a la orientación a objetos de la Asignatura Paradigmas de Programación UTN – FRBA* Autores: Carlos Lombardi – Nicolás Passerini.
  - *Pharo por Ejemplo*. Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet With Damien Cassou and Marcus Denker. Version of 2013-05-12.
  - *Pharo por Ejemplo*. Andrew P. Black Stéphane Ducasse, y otros. Versión: mayo de 2013.
  - Sitio [www.pharo.org](http://www.pharo.org).
  - *Pharo by Example 5*. A.P. Black, S. Ducasse, y otros. Versión: abril de 2017.
-