COLECCIÓN

¿Cómo trabajamos la noción de conjunto de elementos en funcional y en lógico? Con Listas, exacto[1].

En Objetos, el concepto relacionado con los conjuntos es la colección.
Una lista, ¿podía tener elementos de distinto tipo?

- En Haskell no porque hay chequeo estático de tipos (repasamos: el lenguaje es el que pone la restricción, el paradigma no baja ninguna línea al respecto).
- En PROLOG sí, porque el SWI-PROLOG que vemos en PDP tiene chequeo de tipos dinámico.

En Smalltalk una variable es un puntero a un objeto pero no hay restricciones sobre el tipo de objeto al que debe apuntar la variable. Yo puedo evaluar en un Workspace:

amigo := Persona new.
amigo := 'Gerardo'.

Y es algo totalmente válido. La única restricción es que los mensajes que le envíe a amigo puedan ser entendidos por el objeto al que referencia amigo. Si bien hay tipos, no se hace ningún chequeo en tiempo de compilación (por eso Smalltalk tiene chequeo de tipos dinámico, si un mensaje no puede ser entendido por un objeto eso se rompe al evaluarse, es decir, en tiempo de ejecución).

Entonces puedo tener una colección con objetos de clases distintas: puedo poner un 2, una Pera, un Boolean. Esto funciona y no va a dar error, pero cuando quiera enviarle mensajes indistintamente a uno u otro elemento mmmmm…. algo se va a romper. Entonces sí, puedo poner cualquier elemento en una colección, pero tendría onda si los elementos son qué:
polimórficos

¡genial! O sea, en un determinado contexto para mí, que soy el que los colecciona, los trato en forma similar. Tengo triángulos y cuadrados en una misma colección y puedo pedirles a cada uno su área, los puedo tener juntitos[2].

¿Se acuerdan la clase pasada cuando vimos una tortuga y una ballena y dijimos que podían ser polimórficos en algún contexto? ¿no? Bueno, vayan al apunte de la clase pasada… Ahora sí, entonces podemos meter una tortuga y una ballena en una colección y hacer migrar a todos los animales del hemisferio sur.

Otra defi de colección: una colección permite modelar relaciones 1 a N:

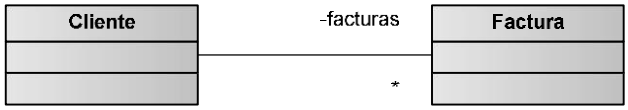
- Un cliente tiene N facturas
- Una factura tiene N renglones o detalles o líneas
- Un empleado tiene N recibos de sueldo o liquidaciones, etc.

ASOCIACIÓN

¿Cómo decimos que "un cliente tiene muchas facturas"?

[1] En realidad, los conjuntos tienen elementos sin un orden específico, mientras que las listas tienen orden en sus elementos. Salvando esta diferencia, la lista sirve como abstracción de la noción de conjunto.
[2] 2 Lo maravilloso es que ni siquiera se qué a qué clase pertenece cada objeto, me basta saber qué le quiero pedir solamente. Puedo tener mañana un pentágono, un círculo, cosas que a priori no se me ocurren…

Bueno, podemos mostrarlo en un Diagrama de Clases UML –un lenguaje estándar para diseñar que veníamos usando casi sin darnos cuenta-

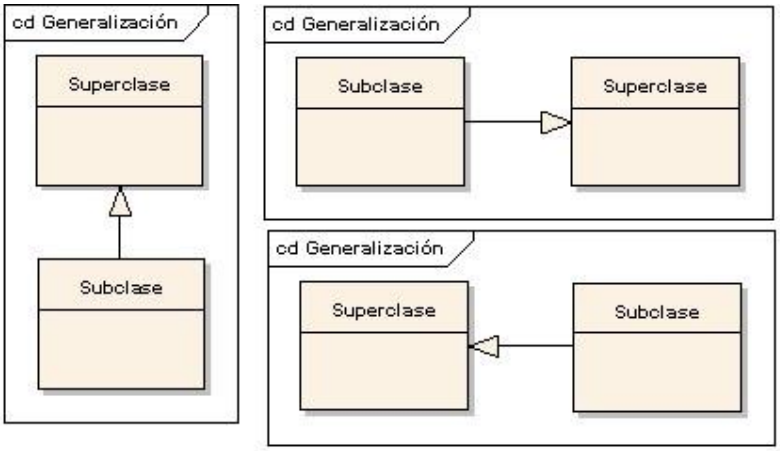


Estamos modelando la relación entre dos entidades: las clases Cliente y Factura. Un cliente **tiene** facturas.

El conector dice que es una **asociación**, esto es que hay un atributo facturas en la clase Cliente (el nombre del atributo facturas se muestra en uno de los extremos de la asociación).

El asterisco (*) sobre facturas muestra la multiplicidad: la relación es de uno (cliente) a muchos (facturas). El uno (1) sobre el extremo del cliente es opcional, podemos indicarlo o no.

Ustedes ya vieron cómo modelar en UML otra relación: la generalización, que muestra cuando una clase hereda de otra superclase. Tengan en cuenta que es importante el dibujo de las flechas:



No importa qué orientación tengan las clases, se trata siempre de una relación de herencia.

¿Qué queremos pedirle a una colección?

- Agregar un elemento.
- Sacar un elemento.
- Modificar un elemento
- Conocer su tamaño.
- Saber si tiene elementos.
- Buscar un elemento.
- Filtrar elementos que cumplan un criterio.
- Generar otra colección aplicando una transformación de sus elementos.
- Totalizar montos, cantidades o partes de elementos de una colección.

Nos metemos en un Workspace y hacemos:
`pecom := Cliente new.` ← tenemos que tener una clase Cliente
`factura := Factura new total: 300.` ← creamos una factura y le indicamos cuál es el monto total.

Método "setter": le setea al atributo total el monto que paso como parámetro.

```
total: unMonto (#Factura, es un método de instancia)
total := unMonto.
```

¿Cómo "enganchamos" la factura a pecom?
Y… estaría bueno generar un método para agregar una factura pecom
agregarFactura: factura.

Codificamos el método:

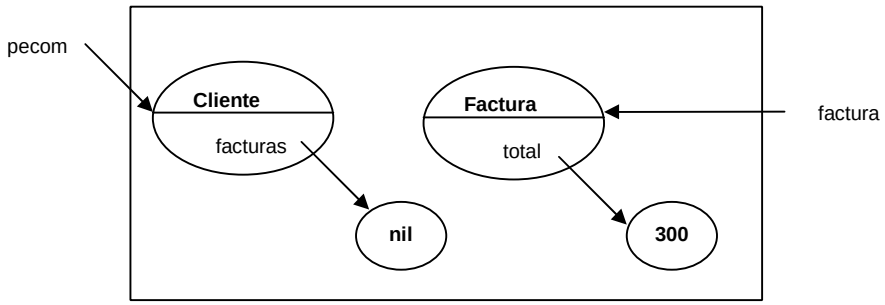
```
agregarFactura: unaFactura (#Cliente)
facturas add: unaFactura
```

Hacer hincapié en unaFactura. Lo que recibo como parámetro tiene un tipo: el tipo es una factura (al cual después me interesará pedirle el total). Por eso es preferible unaFactura que anObject, arg1 o k.

Lo probamos.

```
pecom agregarFactura: factura.
```

No anda. Me dice que no entiende el mensaje "#add:" ¿Qué pachóoooo?
Veamos las referencias:



¡Ah! Entonces, la colección de facturas no está inicializada. ¿Dónde lo inicializamos? Generemos un método:

initialize (#Cliente)
facturas := ¿? new.

¿Qué tipo de colección elegimos?

En principio vamos a usar una *OrderedCollection*, que representa una lista que respeta el orden en el que se van insertando los elementos. Más adelante vamos a ver los diferentes tipos de colección que existen. Entonces el método nos queda:

initialize (#Cliente)
facturas := OrderedCollection new.

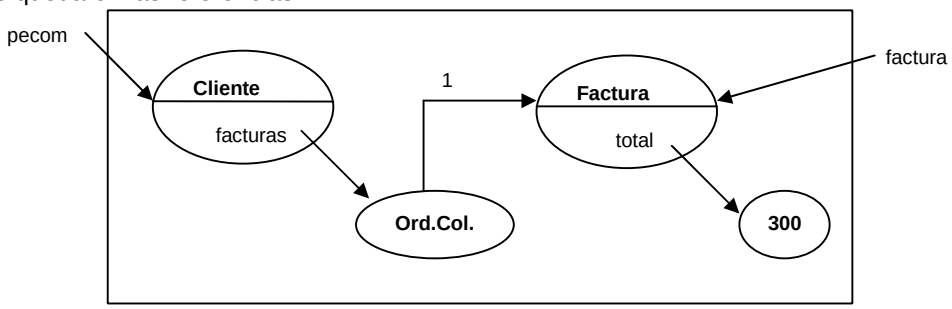
Desde el Worskpace primero hay que inicializar a pecom, pintamos el código (o nos paramos en la misma línea) y lo evaluamos:

```
pecom initialize.
```

Y ahora sí podemos hacer:

```
pecom agregarFactura: factura.
```

¿Cómo quedaron las referencias?



¿Si quiero modificar el total de la factura a 250? **Opción 1) En un Workspace** factura total: 250.

Opción 2) En un método de la clase Cliente

…

```
(facturas at: 1) total: 250
```
"También puedo hacer `facturas first total: 250`*"*

…

¿Qué objeto se modifica con la opción 1, y qué objeto se modifica con la opción 2? ¡El mismo!

Ambas referencias apuntan al mismo objeto, entonces si altero el objeto a partir de alguna de las dos referencias, es **ése** el objeto que cambia. Recordamos que en objetos tengo efecto colateral.

Agreguemos una factura más:

```
pecom agregarFactura: (Factura new total: 100).
```
← No necesito asignarlo primero a una variable.

Puedo pedirle `facturas at: 2`, y eso me devuelve la segunda factura que agregué.

Moraleja: Si la colección respeta el orden de inserción de los elementos, tiene sentido pedir "el primer elemento" de una lista o "el segundo". En la noción matemática de conjunto (donde no hay un orden establecido), no podemos determinar si un elemento está "antes" o "después" de otro.

Interfaz de Collection

¿Qué otras cosas le podemos pedir a la colección de facturas? Revisamos la lista:

Conocer su tamaño

cantidadDeFacturas (#Cliente)
^facturas size

Saber si tiene elementos

Opción 1)
tieneFacturas (#Cliente)
    ^facturas size > 0

Opción 2)
tieneFacturas (#Cliente)
^facturas notEmpty

Ahora que más o menos nos conocen, ¿qué se imaginan que vamos a elegir? Lo que sea más expresivo, está bueno no tener que traducir size > 0… ah, es si tiene elementos. Esto es tiempo que tengo que gastar cuando estoy leyendo el método.

Filtrar elementos por algún criterio

Si queremos filtrar las facturas grosas que son aquellas cuyo monto es mayor a 10.000:

En Objetos	En Funcional
facturasGrosas (#Cliente) ^facturas select: [:factura | factura total > 10000] Una segunda opción: **facturasGrosas** (#Cliente) ^facturas select: [:factura | factura esGrosa] **esGrosa** (#Factura) ^total > 10000 ¿Qué gano haciendo otro método? Que el cliente no decida cuándo una factura es grosa, eso es responsabilidad de la factura. Esto es muy útil si otros objetos también quieren saber si una factura es grosa (además del cliente).	facturasGrosas facs = filter (\factura -> total factura > 10000) facs donde total (_, monto) = monto (La factura viene representada en una tupla Cliente, monto de la factura)

¿Son parecidas, no?
select: y filter: cumplen el mismo objetivo. Filter es una función de orden superior, recibo el criterio como parámetro y lo aplico dentro de la función sin saber exactamente qué tipo de criterio es.

OBJETOS BLOQUE

select: permite recibir un criterio, con un objeto que representa una porción de código. Este objeto es muy importante, es el objeto bloque.

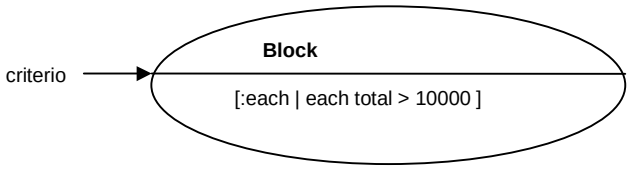
¿Qué nombre tiene ese objeto? No tiene ningún nombre en particular, representa el criterio. Yo bien podría haber hecho:

```
facturasGrosas (#Cliente)
| criterio |
criterio := [ :factura | factura total > 10000 ].
^facturas select: criterio
```

Y guardaría localmente en una variable un objeto que representa una instrucción.

Ahora supongamos que definimos la variable criterio en el mismo Workspace donde tenemos la referencia a los objetos Cliente y Factura:

criterio := [:factura | factura total > 10000]



¿Qué le puedo pedir a criterio? Evaluarlo con un parámetro (el parámetro se llama :factura) criterio value: 60.

Me da error, ahora veo que no le puedo mandar cualquier cosa.

¿Qué tipo de objeto podemos mandarle? Uno que entienda el mensaje total. La variable factura apunta a un objeto Factura que sí entiende el mensaje total.

criterio value: factura (pintamos y vemos el resultado)
False

El select:, entonces recibe un bloque que define el criterio para filtrar los elementos de la colección. ¿Puede ser cualquier bloque?
Revisemos la definición de filter: `filter :: (a -> Bool) -> [a] -> [a]` Ahá…
el criterio tiene que devolver un booleano, no cualquier cosa.
El select: toma una colección y me devuelve los que cumplen con ese criterio.

- **Objeto receptor**: una colección
- **Un solo argumento**: un bloque (el criterio: una porción de código que devuelve un objeto booleano)
- **Objeto devuelto**: otra colección con los elementos filtrados.

Ojo, no modifica la colección original (el select: no tiene efecto colateral), esto es importante y está bueno que sea así: si quiero listar los clientes morosos, no quiero que desaparezcan los clientes que están al día. Entonces trabajo con una colección de clientes en paralelo pero que tengan filtrados los clientes que no tengan deuda.

Recolectar información de una colección

Agregamos la fecha en la factura y queremos obtener el conjunto de todas las fechas en las que le facturé a un cliente.

En Objetos	**En Funcional**
fechasEnLasQueLeFacture ^facturas collect: [:factura | factura fecha]	map (\factura -> fecha factura) facturas

collect: genera una nueva colección aplicando una transformación de cada uno de los elementos. La transformación se da con un objeto bloque que le paso como parámetro.

fechasEnLasQueLeFacture
^facturas collect: [:factura | factura fecha]

¡Lo mismo que hace map!

Fíjense que en ambos la estructura es casi idéntica, sólo que

1) En Objetos el énfasis está puesto en el sustantivo (quién hace las cosas)
2) En Funcional el énfasis está puesto en la acción o el verbo (qué acción es hecha)

Definir qué es lo más importante, si el sujeto o la acción es motivo de discusión en muchos foros de programadores.

¿Tiene alguna restricción respecto a lo que tiene que devolver el bloque? No, ninguna.

Prueben en sus casas hacer

```
bloqueQueSacaLaFechaDeUnaFactura := [ :factura | factura fecha ]

bloqueQueSacaLaFechaDeUnaFactura value: factura.
```

Totalizar

Queremos saber cuánto le facturamos a un cliente:

En Objetos	**En Funcional**

totalFacturacion
^facturas
  inject: 0
  into: [:acum :factura | acum + factura total]
	totalFacturacion facturas =
	  foldl (\acum factura -> acum + total factura) 0
	facturas

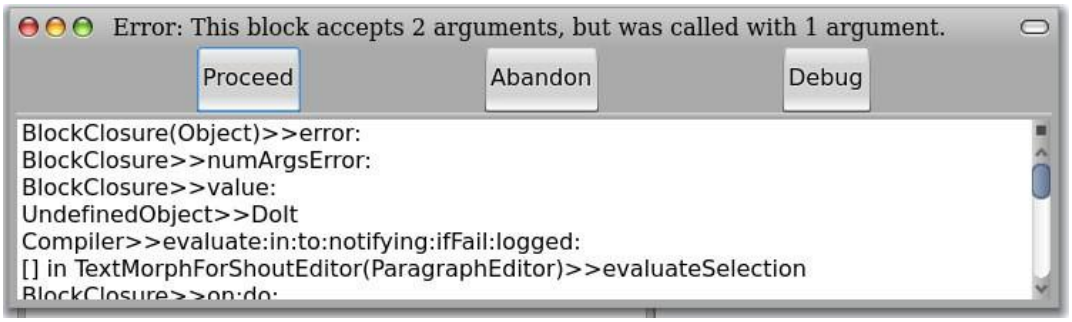
Y sí, es complicado al principio, pero cuando se acostumbren nos lo van a agradecer…

inject: into: permite acumular operaciones sucesivas sobre los elementos de una colección. Necesita:

- Un valor inicial
- Un bloque de código de acumulación que tendrá dos argumentos: el acumulador y cada elemento de la colección

Si tenemos:
suma := [:x :y | x + y]

Si hacemos:
suma value: 7 nos va a tirar:



Entonces para poder evaluar este bloque necesitamos enviarle los dos argumentos: suma value: 2 value: 9.
Que nos da 11.

En Funcional sí puedo dejar la expresión lambda por la mitad, generando una nueva función (recordar funciones parcialmente aplicadas). Esa es una diferencia, además del verbo vs. sustantivo.

Una forma freak de definir la longitud de una lista es:
lista inject: 0 into: [:acum :elemento | acum + 1]

EJERCICIOS CON CONJUNTOS includes:

Queremos saber si un elemento está en la colección.

yaLeFacture: unaFactura (#Cliente)
  ^facturas includes: unaFactura

isEmpty

Lo vimos, es para saber si un conjunto está vacío

union:

Si en Cliente tenemos dos variables de instancia: facturas y notas de débito **comprobantesDeDebito** (#Cliente)
  ^facturas union: notasDebito

¿No me conviene tenerlas juntas?

A veces sí, a veces no. Ojo, si en pocas consultas necesito que las notas de débito y las facturas sean polimórficas, entonces me conviene tenerlas separadas y sólo donde lo necesito hacer

totalFacturacion
  ^self comprobantesDeDebito inject: 0 into: ….

Otro ejemplo con inject:into:

Si una sucursal tiene clientes y un cliente tiene facturas, quizás nos interese saber las facturas de una sucursal:

```
facturas (#Sucursal)
^clients

      inject: OrderedCollection new

      into: [ :acum :cliente | acum union: cliente facturas ]
```

anySatisfy:/allSatisfy:

Es útil para determinar si algún cliente es moroso

```
clientes anySatisfy: [ :cliente | cliente esMoroso ]
```

O si todos los autos que vendo son 0 KM:

```
autos allSatisfy: [ :auto | auto es0Km ]
```

Otros mensajes interesantes:

- intersection:
- difference:

SOBRE LA DECLARATIVIDAD Y LOS PARADIGMAS

Comparando el trabajo entre colecciones y listas vemos que hay notables coincidencias:

Operación sobre un conjunto	Implementación	
	Haskell	**Smalltalk**
Filtrar elementos	filter	select:
Transformar elementos en un nuevo conjunto	map	collect:
Acumular operaciones sobre los elementos en un valor	foldl	inject: into:
Saber si algún elemento/ todos los elementos cumple/n una condición	any/all	anySatisfy:/ allSatisfy:

Y en ambos intervienen:

- La idea de generar abstracciones que representen porciones de código
- Concentrarse en qué hay que hacer y no en cómo "recorrer" los datos (de hecho no sabemos si la colección se implementa con una lista doblemente enlazada, con un array estático o dinámico, etc.)

Lo que nos gustaría decir es que aun cuando objetos es un paradigma que nació imperativo, o sea:

- con asignación destructiva
- donde hay secuencia de pasos y el orden es importante
- donde tengo mayor control sobre el algoritmo

Yo puedo decidir ser más declarativo, ¿cómo?

- cada clase que implemente Collection va a decidir cómo se acceden los elementos entonces no tengo tanto control sobre el algoritmo (estoy delegando la forma en que se recorre la colección)
- consecuencia de lo anterior: no siempre el orden es importante, no puedo determinar cuándo se va a acceder a la cuarta factura (una prueba más de que conozco menos cómo se resuelve internamente)
- disminuyen las asignaciones destructivas (piensen por ejemplo en el método **totalFacturacion** que no necesita variables locales y comparen con el mismo código hecho en C o en Pascal…)

Como me pongo en el rol de "usuario" (aún siendo programador) no me entero ni me importa cómo lo termina haciendo, yo sólo le digo qué me importa de los elementos (y en esto me ayuda mucho tener un objeto bloque que represente comportamiento).

Entonces dejamos la frase Osho del día: