
Paradigmas de Programación
Clase 4
1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1

# </ Clase 4 - POO

- Colecciones. Mensajes más comunes.
- Especificación e Implementación clase compuesta.
- Desarrollo de aplicación en Dolphin.
- Iteradores de colección.
- Diccionario.
- Ejercicios para resolver.

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

# </> Repaso

Mensajes más comunes

Clase simple

Rolo

} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

</>

# Colecciones

} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

# </Colecciones. Mensajes más comunes.

Tipos:

+ OrderedCollection.
+ SortedCollection.
+ Array.
+ Dictionary.

Algunos de los mensajes más comunes que una colección puede responder son: *add*, *remove*, *size*, *at*, *at put*, *isEmpty*, *includes*, *asSet* y *ocurrencesOf*.

# </Colecciones. Mensajes más comunes.

add:

remove:

size:

at:

at: put:

isEmpty:

includes:

asSet:

ocurrencesOf:

# </Colecciones. Mensajes más comunes.

add: →Agrega un elemento al final.

remove: → Elimina un objeto (ifAbsent:[]).

size: →Devuelve el tamaño.

at: → Devuelve el elemento en una posición.

at:put:→Agrega un elemento en una posición determinada.

isEmpty: →Valida si esta vacía (devuelve un booleano).

includes: →Valida si la colección contiene determinado objeto.

asSet: → Convierte la colección a tipo set (sin repeticiones)

ocurrencesOf: →Cuenta cuantas veces se repite un objeto.

</>

# Especificación clase compuesta

} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

# </Especificación clase Farmacia

Definición:

Object subclass: #Farmacia

instanceVariableNames: ‘nombre conjRem’

classVariableNames:''

PoolDictionaries:''

Métodos de clase

crear:nom

*“Crea una instancia de la clase Farmacia y la inicializa con sus datos"*

# </Especificación clase Farmacia

Métodos de Instancia:

>>init: nom *"Inicializa una Farmacia con sus datos"*

>>verNombre *"Retorna el nombre de la Farmacia"*

>>modNombre:otroNom *"Modifica el nombre de la Farmacia"*

>>agregar: unRem*"Agrega un remedio a la Farmacia"*

>>eliminar: unRem *"Elimina un remedio de la Farmacia"*

>>recuperar: i *"retorna el i-esimo remedio"*

>>tamanio *"Retorna el total de remedios de la farmacia"*

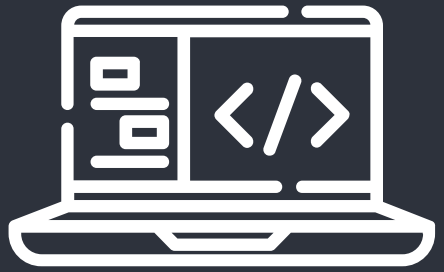
>>verTodos *"Retorna toda la colección"*

>>esVacio *"Valida si hay remedios, retorna booleano"*

>>existe: unRem *"Valida si existe el remedio en la farmacia, retorna booleano"*

</>

# Desarrollar Aplicación



} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

# </Enunciado

Crear una farmacia, darle nombre y cargarle remedios. Luego:

Realizar un incremento del 20% a los remedios con stock menor a una cantidad determinada.
Eliminar remedios del laboratorio “Bagó”.
Modificar el precio del lotrial.

</>

# Implementación clase compuesta

} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

# </Implementación clase Farmacia

Definición:

Object subclass: #Farmacia
instanceVariableNames: 'nombre conjRem'
classVariableNames:''
PoolDictionaries:''

Métodos de clase

crear:nom
*"Crea una instancia de la clase Farmacia y la inicializa con sus datos"*
^self new init:nom.

# </Implementación clase Farmacia

Métodos de Instancia:

```
>>init: nom
"Inicializa una Farmacia con sus datos"
nombre := nom.
conjRem := OrderedCollection new.

>>verNombre "Retorna el nombre de la Farmacia"
^nombre

>>modNombre:otroNom "Modifica el nombre de la Farmacia"
nombre:= otroNom
```

# </Implementación clase Farmacia

Métodos de Instancia:

```
>>agregar: unRem
"Agrega un remedio a la Farmacia"
conjRem                                                    add:unRem.

>>eliminar: unRem "Elimina un remedio de la Farmacia"
conjRem remove: unRem ifAbsent:[^nil].

>>recuperar: i "retorna el i-esimo remedio"
^conjRem at: i.

tamanio "Retorna el total de remedios de la farmacia"
^conjRem size.
```

# </Implementación clase Farmacia

Métodos de Instancia:

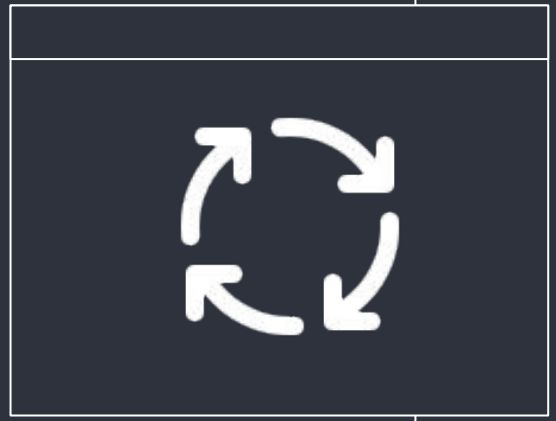
```
>>verTodos "Retorna toda la colección"
^conjRem.

>>esVacio "Valida si hay remedios, retorna booleano"
^conjRem isEmpty.

>>existe: unRem "Valida si existe el remedio en la farmacia, retorna booleano"
^conjRem includes: unRem.
```

</>

# Iteradores de colección



} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

# </Iteradores de Colección

Dada una colección col se pueden utilizar los iteradores:

col do: [: el | bloque de código] liga el parámetro de bloque “el” con cada elemento de la colección en el bloque de codigo.

```
col:= far verTodos.
col do:[:rem | MessageBox notify: rem verNombre].
```

“Se recorre toda la colección, rem toma el valor de cada objeto de la colección en cada iteración”

# </Iteradores de Colección

reject: devuelve una colección con los elementos para los que la expresión booleana devuelve false.

col:= far verTodos.
col1:= col reject:[:rem | rem verLab = 'Bayer'].
"col1 estará compuesta por objetos remedios que no sean del laboratorio Bayer"

# </Iteradores de Colección

select: devuelve una colección con los elementos para los que la expresión booleana devuelve true.

col:= far verTodos.
col1:= col select:[:rem | rem verLab = 'Bayer'].
"col1 estará compuesta por objetos remedios del laboratorio Bayer"

# </Iteradores de Colección

collect: devuelve una colección con los objetos resultantes de la evaluación de la expresión para cada elemento.

```
col:= far verTodos.
col2:= col collect:[:rem | rem verLab].
```

“col2 estará compuesta por objetos string con los nombres de los laboratorios”

# </Iteradores de Colección

detect: devuelve el primer elemento para el que la expresión booleana devuelve true.

col:= far verTodos.
r:= col detect:[:rem|rem verNombre = 'Migral'] ifNone[^nil].
"se retorana en r el objeto de la colección col que tenga el nombre Migral"

</>

# Diccionario

} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1

# </Diccionario

El diccionario es un tipo de arreglo similar a las listas, pero en lugar de tener un índice numérico, cuentan con una clave única. Son estructuras del tipo clave-valor, y tienen la siguiente forma:

```
{
  “Clave 1” : ”Valor 1”,
  “Clave 2” : “Valor 2”,
  ...
  “Clave 3” : “Valor N”
}
```

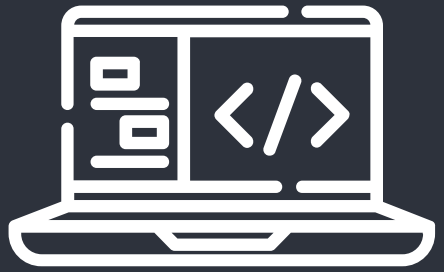
# </Diccionario

dic:= Dictionary new.
col:= far verTodos.
colLab:= col collect:[:el | el verLab]. “tomo los nombres de todos los laboratorios”
sinRep:= colLab asSet. “saco los repetidos”

sinRep do: [: el| dic at: el put: ( colLab occurrencesOf: el)].
“recorro el diccionario por claves e imprimo Laboratorio y cantidad.”
dic keysDo:[ :cla| Transcript show: ‘Laboratorio: ', cla, ‘Cantidad de remedios:’ , dic at:cla. Transcript cr].

</>

# Resolver ejercicios TP3



} /> [

1 0 1 1  0 1 1  0 1  1 0 1 1 0 0 1  1 0  1 1 0 1 1  0 1 1  0 1  1 1 0 1 1 0  1 1 0 1 1 1  1 1 0 1