Ejercicio práctico para utilizar los conceptos de la POO

Conceptos que se van a tratar:

- Clase
- Objeto / Instancia
- Metodo
- Mensaje
- Encapsulamiento
- Igualdad / Identidad
- Variables de instancia y de clase
- Pseudovariables: self, super, true, false, nil
- Herencia
- Abstracción
- Ensamble

Herramientas:

- Browser de clases
- Workspace
- Inspector
- Transcript
- Depurador
- Pacakage manager

Lenguaje

- Métodos de acceso (Accessors)
- Manejo inicial de cadenas y números
- Representación en string de los objetos (#printString)
- Utilización de fechas
- Manejo de colecciones

El objetivo de este apunte es comenzar con una clase Persona, a la que se le asignarán un conjunto de atributos iniciales y, posteriormente, se irá agregando comportamiento progresivamente incrementando el nivel de complejidad. A medida que se avanza, también se irán incorporando nuevas clases y aumentando el nivel de complejidad. En este proceso se irán incorporando los conceptos de la programación orientada a objetos.

Parte 1: Clase Persona

En este ejercicio crearemos una clase Persona con atributos que se describen a continuación y le daremos el siguiente comportamiento:

a. Capacidad para asignar y retornar valores de las variables de instancia (Métodos de acceso)
b. Saludar, imprime un mensaje de saludo personalizado, utilizando el nombre de la persona.
c. Calcular la edad de la persona.
d. Calcular la edad que tendrá la persona en un año determinado.
e. Comparar la edad de dos personas.
f. Comportamiento para que la persona pueda tener un estado civil, asignado entre tres valores posibles.

Definición de atributos y primer comportamiento

a. Capacidad para asignar y retornar valores de las variables de instancia (Métodos de acceso)

Queremos modelar una clase "Persona" que tenga los siguientes atributos:

- nombre
- edad
- género

Además, la clase deberá tener un método "saludar" que imprima un mensaje de saludo personalizado, utilizando el nombre de la persona.

Ver conceptos: Clase, Objeto, método, mensaje
Aula virtual: Sección I -> Conceptos básicos del Paradigma de OO
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=19983&redirect=1

Aula virtual: Sección I -> Programación orientada a objetos - POO
https://frro.cvg.utn.edu.ar/mod/resource/view.php?id=20555&redirect=1

Para crear una clase utilizaremos el browser de clases y para probar el código utilizaremos el workspace.

Herramienta: Browser de Clases
Es una de las herramientas más importantes para el desarrollo en Smalltalk. Se utiliza para

explorar y trabajar con las clases y métodos de un sistema. En esta herramienta es donde se crean las clases, se definen atributos y comportamientos (métodos) a las mismas. Proporciona una vista jerárquica de las clases y subclases, lo que permite navegar y buscar fácilmente entre ellas.

Definir la clase "Persona" con los atributos señalados:

Browser de Clases

```
Object subclass: #Persona
    instanceVariableNames: 'nombre edad genero'
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
```

Uso del Browser de Clases en Dolphin
Aula virtual: Sección II -> Crear clases y métodos
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=20609&redirect=1

Y ya podemos empezar a crear objetos a partir de la clase creada. Para ello vamos a utilizar el workspace.

Herramienta: Workspace
Es una herramienta para ejecutar código fácilmente. Nos permite probar rápidamente el código que estamos desarrollando o bien probar código existente en la biblioteca de clases.

Workspace

```
persona := Persona new.
```

La variable “persona”, la cual es una variable temporal del workspace, va a contener un valor mientras la ventana del workspace permanezca abierta. Las variables en smalltalk siempre **apuntan** a un valor (objeto), y, cuando el workspace se cierre, se eliminarán las variables utilizadas, eliminando de esta manera las referencia al objeto al que apuntaban, en este caso al objeto persona. Si ese objeto persona no tiene otras referencias, va a ser eliminado automáticamente por el sistema (Garbage Collector) para liberar la memoria, si el objeto tiene referencias desde otros lados, continuará en memoria y no será removido.

Ver: Variables de instancia
Aula virtual: Sección II -> Variables en Smalltalk
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=20610&redirect=1

El objeto persona por ahora tiene tres atributos, pero todavía no le definimos ningún comportamiento. No hemos creado ningún mensaje para poder enviarle. Los mensajes que se pueden enviar a este objeto son los que están definidos en la clase Object (ya que la clase Persona hereda de ella). Un mensaje que se le puede enviar es el #printString, este mensaje devuelve la representación en string del objeto. Por defecto imprime a qué clase pertenece el objeto, de la forma de “es un + nombre de clase”, y lo hace en inglés resultando una salida de este tipo: “a / an” + “nombre de la clase”. En nuestro caso si enviamos el mensaje #printString al objeto persona obtendremos lo siguiente:

Workspace

```
persona := Persona new.

persona printString. -> a Persona
```

Si quisiéramos enviar el mensaje #saludar en este momento nos va a dar un error que dice: “Persona does not understand #saludar”. Esto significa que le enviamos al objeto un mensaje que no entiende (lease, un mensaje que no está implementado en su propia clase, ni en ninguna de sus superclases).

Por otro lado podemos ir haciendo uso de la herramienta “Inspector”, esta herramienta nos ayuda a ver el estado interno del objeto (esto son los atributos que tiene y sus valores). Allí podemos ver que nos muestra las tres variables definidas en la clase y que tienen un valor “nil”.

Herramienta: Inspector
El inspector es una herramienta que nos permite interactuar con cualquier objeto del sistema. Nos permite ver el estado interno de las variables, modificarlo y enviar mensajes a los objetos.

Ahora vamos a asignar un nombre a la persona desde el workspace.

Smalltalk provee un mecanismo de **encapsulamiento** donde a los objetos la única manera de asignar un valor es por medio de crear un comportamiento en la clase para tal fin. De esta manera, las únicas habilitadas para manipular de forma directa a las variables de instancia son la clase que lo define y subclases que puedan heredar el comportamiento de la clase. Desde otro lado no se pueden manipular de forma directa las variables, por lo tanto, desde el workspace es imposible asignar el nombre a la persona manipulando directamente la variable. Para ello tenemos que hacer un método que permita la asignación del nombre.

Browser de Clases - Clase Persona - método de instancia

```
nombre: unString
nombre := unString
```

Creamos un método #nombre: que se compone del mismo nombre de la variable más un dos puntos ":". Este es un método de palabra clave, donde el "unString" es el parámetro que se le pasa al método.

Vamos a asignar el nombre "Ana" a la persona que creamos en el workspace:

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: 'Ana'.
```

Al evaluar "persona nombre: 'Ana'", estamos asignando el nombre al objeto persona. Si miramos ahora el inspector vamos a ver que el atributo nombre ahora apunta al string 'Ana'.

Ahora vamos a consultar el nombre de la persona desde el workspace. Por el encapsulamiento no podemos acceder directamente a la variable a verla, por lo que vamos a necesitar un método que la retorne. Volvemos al browser de clases:

```
Browser de Clases - Clase Persona - método de instancia
nombre
^ nombre
```

Aquí creamos en la clase persona un método que tiene el mismo nombre que la variable de instancia. No hay que confundir entre variable y método. Generalmente se utiliza el mismo nombre ya que es muy representativo para dar acceso a la variable.

Volviendo al workspace, podemos ahora consultar el nombre de la persona:

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: 'Ana'.
persona nombre -> 'Ana'
```

De esta manera creamos dos métodos en la clase Persona, uno para asignar el valor y otro para que lo devuelva. Esto es lo que se llaman **Métodos de acceso** o **Accessors**, y se realizan para todas las variables que queremos se le pueda asignar un valor desde fuera de la propia clase.

Hay diferentes formas de implementar los métodos de acceso tales como:

- Agregar un prefijo “i” (input) para los métodos de asignación y un prefijo “o” para los que devuelven el valor. Aplicado para nuestro caso sería: ***iNombre: unNombre*** y ***oNombre***.
- Agregar un prefijo “asigna” para los métodos de asignación y un prefijo “dev” para los que devuelven el valor. Aplicándolo en nuestro caso sería: ***asignaNombre: unNombre*** y ***devNombre***.
- Agregar el prefijo “set” (asignar en inglés) para los métodos de asignación y el prefijo “get” (obtener) para los que devuelven. En nuestro ejemplo quedaría: ***setNombre: unNombre*** y ***getNombre***.
- No agregar ningún prefijo, utilizar el nombre de la variable como nombre del método. Es el caso mostrado en este ejemplo y los nombres de los métodos son: ***nombre: unNombre*** y ***nombre***.

Notar que lo que cambia son los nombres de los métodos pero la forma de implementarlos es para todos los casos iguales al ejemplo mostrado.

Este par de métodos es lo que se conoce como **Métodos de acceso,** más comúnmente conocido por su traducción a inglés como **Accessors**. Los métodos de acceso se utilizan para asignar y retornar el valor de una variable de instancia, asegurando el encapsulamiento de los datos y de esta manera controlar el acceso a los atributos desde la propia clase que define los atributos evitando modificaciones indeseadas o indebidas.

Ahora creamos los métodos de acceso para las otras variables:

Browser de Clases - Clase Persona - método de instancia

```
edad: unNumero
edad := unNumero

edad
^ edad

genero: unString
genero := unString

genero
^ genero
```

Volvemos al workspace y asignamos la edad y el género a la persona.

Workspace

```
persona := Persona new.

persona printString. -> a Persona

persona nombre: ‘Ana’.
persona nombre -> ‘Ana’

persona edad: 25.
persona genero: ‘Femenino’.
```

Si vemos al inspector vamos a ver actualizado el objeto con los datos asignados.

Antes de continuar con los comportamientos, crearemos un paquete llamado “Ejercicio POO”, para guardar nuestra clase en él y compartirla, distribuirla, etc.

Herramienta: Package Manager
Es una herramienta de gestión de paquetes que permite a los desarrolladores organizar, instalar, actualizar y distribuir código en forma de archivo de texto. Un paquete es una colección de clases, métodos y otros elementos que se agrupan lógicamente.

Uso del Package Manager en Dolphin
Aula virtual: Sección II -> Trabajando con paquetes
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=18243&redirect=1

b. Saludar, imprime un mensaje de saludo personalizado, utilizando el nombre de la persona.

Continuamos con la creación del método saludar:

Browser de Clases - Clase Persona - método de instancia *saludar*

```
Transcript show: 'Hola, mi nombre es ', nombre, '!'.
```

Volvemos al workspace y vemos como la persona da su saludo. El saludo va a ser impreso en la ventana del Transcript.

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: 'Ana'.
persona nombre -> 'Ana'

persona edad: 25.
persona genero: 'Femenino'.

persona saludar -> 'Hola, mi nombre es Ana!' (ver en
ventana de Transcript)
```

Herramienta: Transcript
Es un objeto que se utiliza para loguear mensajes del sistema. Es un estilo de "consola". Nosotros lo utilizaremos además para imprimir mensajes para el usuario, reportes, etc.

Ver: Uso de transcript en Smalltalk
Aula virtual: Sección II -> Operaciones de Entrada/Salida en Smalltalk
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=20607&redirect=1

Para ejecutar el método saludar podemos hacer uso del depurador y ver cómo se ejecuta paso a paso. Este procedimiento se puede hacer con cualquier método para ver su comportamiento en detalle. Para ello hay que marcar "persona saludar" y en el menú contextual poner "Debug it". Esto levantará la ventana de depuración donde se puede ver el código paso a paso.

Herramienta: Depurador
Es una herramienta para ejecutar código desde un punto específico, paso a paso y visualizando el estado interno de los objetos que interactúan. Facilita la identificación y corrección de errores.

Hasta este punto, lo que hicimos fue definir la clase "Persona" con los atributos "nombre", "edad" y "género". Además, hemos creado un método "saludar" que utiliza la clase Transcript para imprimir un mensaje personalizado.

Vamos a crear otro objeto persona en el workspace, y lo vamos a asignar a otra variable:

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: 'Ana'.
persona nombre -> 'Ana'

persona edad: 25.
persona genero: 'Femenino'.

persona saludar -> 'Hola, mi nombre es Ana!' (ver en
ventana de Transcript)

p1 = Persona new.
p1 nombre: 'Juan'.
p1 edad: 18.
p1 genero: 'Masculino'.

p1 saludar -> 'Hola, mi nombre es Juan!'
```

Al ejecutar este código, se debería imprimir en la consola el siguiente mensaje: "Hola, mi nombre es Juan!".

Desarrollo de comportamientos de la clase "Persona"

Ahora iremos agregando los diferentes comportamientos definidos a la clase Persona.

c. Verificar si la persona es mayor de edad.

```
Browser de Clases - Clase Persona - método de instancia
esMayorDeEdad
^ edad >= 18
```

Este método devuelve un booleano que indica si la persona es mayor de edad o no, basándose en su edad actual.

d. Calcular la edad de la persona.

Hasta ahora venimos trabajando la edad como un atributo asignado y no calculado. Para calcular la edad de una persona es necesario saber su fecha de nacimiento y compararla con la fecha actual.
Vamos a agregar la variable “fechaNacimiento” en la clase Persona.

```
Browser de Clases
Object subclass: #Persona
    instanceVariableNames: 'nombre edad genero
fechaNacimiento'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

Notar que si inspeccionamos los objetos que habíamos creado previamente, tienen los datos asignados y se agregó la variable fechaNacimiento en valor nil. Al crear la variable de instancia en la clase, automáticamente se crea para todas las instancias existentes en el sistema.

Ahora vamos a crear los métodos de acceso para asignar y retornar la fecha de nacimiento:

```
Browser de Clases - Clase Persona - método de instancia
fechaNacimiento: unaFecha
fechaNacimiento := unaFecha

fechaNacimiento
^ fechaNacimiento
```

Vamos al workspace a asignarle una fecha de nacimiento las instancias de persona que tenemos creadas:

```
Workspace
persona := Persona new.
```

```
persona printString. -> a Persona

persona nombre: 'Ana'.
persona nombre -> 'Ana'

persona edad: 25.
persona genero: 'Femenino'.

persona saludar -> 'Hola, mi nombre es Ana!' (ver en
ventana de Transcript)

p1 = Persona new.
p1 nombre: 'Juan'.
p1 edad: 18.
p1 genero: 'Masculino'.

p1 saludar -> 'Hola, mi nombre es Juan!'

persona fechaNacimiento: (Date fromString:
'18/03/1998').
p1 fechaNacimiento: (Date fromString: '25/08/2005').
```

Si miramos el inspector vamos a ver que tiene la fecha asignada en el atributo "fechaNacimiento".

Notar que no asignamos directamente el string como fecha de nacimiento, sino que creamos un objeto Date que representa la fecha. De esta manera vamos a poder trabajar la fecha como tal, lo que nos facilita comparaciones, cálculos de días, etc, en definitiva poder trabajarlo directamente como una fecha. Esto lo hacemos por medio de enviar el mensaje de clase #fromString: a la clase Date, pasando cómo parámetro la fecha escrita en formato string.

Ver: Trabajando con fechas
Aula virtual: Sección II -> Trabajando con fechas
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=23231&redirect=1

Con este dato agregado, estamos en condiciones de modificar el método #edad para que devuelva la edad de la persona basado en su fecha de nacimiento.

Browser de Clases - Clase Persona - método de instancia

```
edad
^ Date today yearsSince: fechaNacimiento.
```

Notar que el método de asignación de la edad (#edad:) y la variable edad ya no tienen sentido. Vamos a removerla, para ello antes vamos a verificar los lugares donde se manipula de manera directa la edad y lo vamos a cambiar por una llamada al método recién creado. Los lugares donde se utiliza son:

- ➔ ***Método #edad:*** Este método no tiene sentido ya que la edad no es un valor que se pueda ingresar directamente. Lo vamos a eliminar.
- ➔ ***Método #esMayorDeEdad*** Modificamos el uso de la variable de forma directa, por la llamada al método que retorna la edad.

```
Browser de Clases - Clase Persona - método de instancia
~~edad: unNumero~~
~~edad := unNumero~~

esMayorDeEdad
^ self edad >= 18
```

Hacemos uso de la pseudovariable self para referenciar el objeto persona que este llamando a este mensaje.

Y por último tenemos que borrar el atributo edad de la clase Persona, ya que no tiene más sentido tenerlo allí.

```
Browser de Clases
Object subclass: #Persona
    instanceVariableNames: 'nombre genero fechaNacimiento'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

Si visualizamos nuestras instancias en el inspector, veremos que ya no existe más la variable edad. Al eliminarla de la clase, automáticamente se elimina de todas las instancias.

Para verificar que todo sigue funcionando correctamente volvemos al workspace y consultamos la edad y si es mayor de edad para cada instancia.

```
Workspace
persona := Persona new.
```

```
persona printString. -> a Persona

persona nombre: ‘Ana’.
persona nombre -> ‘Ana’

persona edad: 25.
persona genero: ‘Femenino’.

persona saludar -> ‘Hola, mi nombre es Ana!’ (ver en
ventana de Transcript)

p1 = Persona new.
p1 nombre: ‘Juan’.
p1 edad: 18.
p1 genero: ‘Masculino’.

p1 saludar -> ‘Hola, mi nombre es Juan!’

persona fechaNacimiento: (Date fromString:
‘18/03/1998’).
p1 fechaNacimiento: (Date fromString: ‘25/08/2005’).

persona edad -> ‘25’
p1 edad -> ‘18’

persona esMayorDeEdad -> true
p1 esMayorDeEdad -> true
```

e. Calcular la edad que tendrá la persona en un año determinado.

Para esto, definiremos el siguiente método:

Browser de Clases - Clase Persona - método de instancia
edadEn: anio

```
    ^ anio - fechaNacimiento year
```

Este método toma como argumento un año y devuelve la edad que tendrá la persona en ese año.

Volvemos al workspace a probarlo:

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: 'Ana'.
persona nombre -> 'Ana'

persona edad: 25.
persona genero: 'Femenino'.

persona saludar -> 'Hola, mi nombre es Ana!' (ver en
ventana de Transcript)

p1 = Persona new.
p1 nombre: 'Juan'.
p1 edad: 18.
p1 genero: 'Masculino'.

p1 saludar -> 'Hola, mi nombre es Juan!'

persona fechaNacimiento: (Date fromString:
'18/03/1998').
p1 fechaNacimiento: (Date fromString: '25/08/2005').

persona edad -> '25'
p1 edad -> '18'

persona esMayorDeEdad -> true
p1 esMayorDeEdad -> true
```

```
persona edadEn: 2047 -> 49
p1 edadEn: 2047 -> 42
```

f. Ahora vamos a implementar un método para comparar la edad de dos personas.

Para esto, definiremos el siguiente método:

```
Browser de Clases - Clase Persona - método de instancia
mayorQue: otraPersona
^ self edad > otraPersona edad
```

En este caso comparamos por la edad de las personas, es decir si nacieron en el mismo año, va a retornar que ninguna de las dos persona es mayor a la otra. Para tener mejor precisión, podemos comparar por la fecha de nacimiento:

```
Browser de Clases - Clase Persona - método de instancia
mayorQue: otraPersona
^ fechaNacimiento > otraPersona fechaNacimiento
```

Notar que aquí usamos directamente la variable fechaNacimiento para obtener la fecha de nacimiento del objeto receptor, y utilizamos el método #fechaNacimiento en el segundo objeto a quien comparamos.

Vamos al workspace a probarlo.

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: 'Ana'.
persona nombre -> 'Ana'

persona edad: 25.
persona genero: 'Femenino'.
```

```
persona saludar -> 'Hola, mi nombre es Ana!' (ver en
ventana de Transcript)

p1 = Persona new.
p1 nombre: 'Juan'.
p1 edad: 18.
p1 genero: 'Masculino'.

p1 saludar -> 'Hola, mi nombre es Juan!'

persona fechaNacimiento: (Date fromString:
'18/03/1998').
p1 fechaNacimiento: (Date fromString: '25/08/2005').

persona edad -> '25'
p1 edad -> '18'

persona esMayorDeEdad -> true
p1 esMayorDeEdad -> true

persona edadEn: 2047 -> 49
p1 edadEn: 2047 -> 42

persona mayorQue: p1 -> true
p1 mayorQue: persona -> false
```

g. Agregar comportamiento para manejar el estado civil de la persona

Para ello vamos a agregar un atributo “estadoCivil” que represente el estado civil de la persona, que solamente pueda ser “Soltero/a”, “Casado/a”, “Divorciado/a”. Para hacer esto, vamos a cambiar la definición de la clase "Persona" agregando el atributo:

```
Browser de Clases
Object subclass: #Persona
```

```
    instanceVariableNames: 'nombre genero fechaNacimiento
estadoCivil'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

Ya podemos ver el inspector y verificar que se agregó la variable estadoCivil a nuestras dos instancias y ambas están en nil.

En este caso ya que solo tenemos que asignar tres estados civiles que están definidos, no vamos a crear el método accessor #estadoCivil: para asignarlo, ya que él mismo daría libertad de asignar cualquier valor. Si vamos a utilizar el método accessor #estadoCivil para que retorne el estado civil de la persona.

Para asignar el estado civil vamos a crear un método para cada estado y que le asigne el valor que deseamos.

```
Browser de Clases - Clase Persona - método de instancia
estadoCivil
^ estadoCivil

soltero
estadoCivil := 'Soltero/a'

casado
estadoCivil := 'Casado/a'

divorciado
estadoCivil := 'Divorciado/a'
```

Ahora volvemos al workspace y probamos con nuestras instancias.

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: 'Ana'.
persona nombre -> 'Ana'
```

```
persona edad: 25.
persona genero: ‘Femenino’.

persona saludar -> ‘Hola, mi nombre es Ana!’ (ver en
ventana de Transcript)

p1 := Persona new.
p1 nombre: ‘Juan’.
p1 edad: 18.
p1 genero: ‘Masculino’.

p1 saludar -> ‘Hola, mi nombre es Juan!’

persona fechaNacimiento: (Date fromString:
‘18/03/1998’).
p1 fechaNacimiento: (Date fromString: ‘25/08/2005’).

persona edad -> ‘25’
p1 edad -> ‘18’

persona esMayorDeEdad -> true
p1 esMayorDeEdad -> true

persona edadEn: 2047 -> 49
p1 edadEn: 2047 -> 42

persona mayorQue: p1 -> true
p1 mayorQue: persona -> false

persona soltero.
persona estadoCivil -> ‘Soltero/a’

p1 casado.
p1 estadoCivil -> ‘Casado/a’
```

Workspace completo

```
Workspace
persona := Persona new.

persona printString. -> a Persona

persona nombre: ‘Ana’.
persona nombre -> ‘Ana’

persona edad: 25.
persona genero: ‘Femenino’.

persona saludar -> ‘Hola, mi nombre es Ana!’ (ver en
ventana de Transcript)

p1 := Persona new.
p1 nombre: ‘Juan’.
p1 edad: 18.
p1 genero: ‘Masculino’.

p1 saludar -> ‘Hola, mi nombre es Juan!’

persona fechaNacimiento: (Date fromString:
‘18/03/1998’).
p1 fechaNacimiento: (Date fromString: ‘25/08/2005’).

persona edad -> ‘25’
p1 edad -> ‘18’

persona esMayorDeEdad -> true
p1 esMayorDeEdad -> true

persona edadEn: 2047 -> 49
p1 edadEn: 2047 -> 42

persona mayorQue: p1 -> true
```

```
p1 mayorQue: persona -> false

persona soltero.
persona estadoCivil -> 'Soltero/a'

p1 casado.
p1 estadoCivil -> 'Casado/a'
```

Ejercicio práctico: Parte 1

Crear la clase Producto que tenga los siguientes atributos:

- nombre
- descripción
- precio
- cantidad en inventario (stock)

Crear comportamientos para:

a. Crear métodos de acceso y carga datos para la clase Producto.
b. Retornar el monto total en dinero que se tiene de producto en inventario. Este método retorna un valor numérico que resulta de multiplicar el precio por la cantidad en inventario.
c. Verificar disponibilidad: se le pasa por parámetro la cantidad de productos que se requieren y verifica si hay disponibilidad de dicha cantidad de productos. El método retorna true/false en caso de tener o no disponibilidad.
d. Incrementar cantidad en inventario. Permite aumentar la cantidad del producto en el inventario. Este método tiene como parámetro la cantidad y se debe adicionar a la cantidad en inventario existente. Este método se utilizará luego para la implementación de compra de productos.
e. Disminuir la cantidad en inventario. Permite disminuir la cantidad del producto en el inventario. Este método tiene como parámetro la cantidad y se debe restar de la cantidad en inventario existente, para ello hay que verificar previamente si hay disponibilidad del producto para la cantidad que se quiere disminuir. Este método se utilizará en un futuro para la implementación de la venta de productos.
f. Verificar cantidad mínima en inventario: es un método que retorna verdadero o falso, comparando la cantidad en inventario contra la cantidad mínima en inventario. (Notar que hay que agregar una variable a la clase Producto que modele la cantidad mínima en inventario para poder realizarlo).
g. Retornar la cantidad de productos a comprar para alcanzar la cantidad mínima. Este método retorna un número entero.

Parte 2 - Herencia

A partir de la clase Persona creada en el punto anterior, modelamos un sistema para gestionar el funcionamiento de una universidad, comenzando por una descripción básica que la iremos complejizando a medida que se avanza con el ejercicio.

La universidad cuenta con docentes y alumnos. Para modelar estas entidades nos apoyaremos en la clase Persona con la que estamos trabajando.

Además de los datos definidos en la clase Persona, el docente cuenta con los siguientes atributos:

- Fecha alta
- Cargo

Y al alumno se le agregan los siguientes atributos:

- Fecha inscripción
- Carrera

Y queremos tener los siguientes comportamientos:

h. Docente
   i. **antigüedad** este método tiene que devolver la cantidad de años que el docente trabaja en la universidad.
   ii. **saludar** El docente tiene un saludo particular, donde aparte de decir su nombre también dice su cargo y la antigüedad que tiene en el mismo.

i. Alumno
   i. **diasEstudiando** devuelve la cantidad de días que transcurrieron desde la inscripción del alumno al día de hoy.
   ii. **saludar** El alumno saluda dando su nombre y la carrera a la que está inscripto.

a. Crear las clases Docente y Alumno

Definir la clase "Docente" como subclase de Persona con los atributos definidos:

```
Browser de Clases
Persona subclass: #Docente
    instanceVariableNames: 'fechaAlta cargo'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

Definir la clase Alumno también como subclase de Persona con los atributos definidos:

Browser de Clases

```
Persona subclass: #Alumno
    instanceVariableNames: 'fechaInscripcion carrera'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

Y ya podemos crear objetos alumnos y docentes a partir de la clase creada. Para ello vamos a utilizar el workspace.

Workspace

```
alumno := Alumno new.
docente := Docente new.
```

Al inspeccionar ambos objetos se verá que el objeto alumno tiene los atributos heredados de su superclase Persona, y también los atributos definidos para la clase Alumno. Lo mismo sucede con la clase docente. Las clases Alumno y Docente al heredar de la clase Persona, no solo heredan sus atributos, sino que también sus comportamientos.

Ver: Herencia
Aula virtual: Sección I -> Conceptos básicos del Paradigma de OO
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=19983&redirect=1

Aula virtual: Sección I -> Programación orientada a objetos - POO
https://frro.cvg.utn.edu.ar/mod/resource/view.php?id=20555&redirect=1

Para ver esto en acción, vamos a utilizar los métodos definidos en la clase Persona para dar un nombre, una fecha de nacimiento, un género y el estado civil al alumno y al docente.

Workspace

```
alumno := Alumno new.
docente := Docente new.

alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.

docente nombre: 'Carlos'.
```

```
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
docente casado.
```

Si se inspeccionan ambos objetos se verá que los atributos definidos en Persona están completados, pero no es así para los específicos de Alumno y Docente. Esto es porque todavía no implementamos ningún comportamiento en dichas clases, por lo tanto no podemos asignar valores a los atributos (por encapsulamiento).

b. Crear métodos de acceso para Alumno y Docente

Comenzamos con la clase Alumno y creamos los métodos para asignar la fecha de inscripción y la carrera a la que está inscripto.

```
Browser de Clases - Clase Alumno - método de instancia
fechaInscripcion: unaFecha
fechaInscripcion := unaFecha

fechaInscripcion
^ fechaInscripcion

carrera: unString
carrera := unString

carrera
^ carrera
```

Volvemos al workspace a probar los métodos desarrollados:

```
Workspace
alumno := Alumno new.
docente := Docente new.

alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.
```

```
docente nombre: 'Carlos'.
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
docente casado.

alumno fechaInscripcion:  (Date fromString:
'12/03/2021').
alumno carrera: 'Ingeniería en Sistemas de Información'.

alumno fechaInscripcion -> 'viernes, 12 de marzo de
2021'
alumno carrera -> 'Ingeniería en Sistemas de
Información'
```

Continuamos con el Docente y creamos los métodos para asignar la fecha de alta y el cargo que posee.

Browser de Clases - Clase Docente - método de instancia

```
fechaAlta: unaFecha
fechaAlta := unaFecha

fechaAlta
^ fechaAlta

cargo: unString
cargo := unString

cargo
^ cargo
```

Vamos al workspace a probar los métodos desarrollados:

Workspace

```
alumno := Alumno new.
docente := Docente new.
```

```
alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.

docente nombre: 'Carlos'.
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
docente casado.

alumno fechaInscripcion:  (Date fromString:
'12/03/2021').
alumno carrera: 'Ingeniería en Sistemas de Información'.

alumno fechaInscripcion -> 'viernes, 12 de marzo de
2021'
alumno carrera -> 'Ingeniería en Sistemas de
Información'

docente fechaAlta:  (Date fromString: '12/08/1993').
docente cargo: 'Ayudante de primera'.

docente fechaAlta -> 'jueves, 12 de agosto de 1993'
docente cargo -> 'Ayudante de primera'
```

Volvemos al Alumno y vamos a implementar la funcionalidad requerida. Por un lado es calcular la cantidad de días que el alumno está estudiando en la universidad y también la implementación del saludo del alumno.

c. Implementar comportamiento de cantidad de días estudiando

Comenzamos con la cantidad de días que está estudiando. Para ello creamos el siguiente método en la clase Alumno:

Browser de Clases - Clase Alumno - método de instancia

```
diasEstudiando
^ Date today subtractDate: fechaInscripcion
```

Analizando el método tenemos:

- Date today -> retorna la fecha del día, retorna un objeto Date.
- Método subtractDate: -> es un método que está implementado en la clase Date, que retorna la cantidad de días entre el objeto receptor y la fecha (objeto Date) que le pasamos por parámetro.
- En este caso el objeto receptor es la fecha del día (Date today) y el parámetro es la fechaInscripcion que también es un objeto Date (creado con el Date fromString: …)

Ver: Trabajando con fechas
Aula virtual: Sección II -> Trabajando con fechas
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=23231&redirect=1

Vamos al workspace a probar el método:

```
Workspace
alumno := Alumno new.
docente := Docente new.

alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.

docente nombre: 'Carlos'.
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
docente casado.

alumno fechaInscripcion:  (Date fromString:
'12/03/2021').
alumno carrera: 'Ingeniería en Sistemas de Información'.

alumno fechaInscripcion -> 'viernes, 12 de marzo de
2021'
```

```
alumno carrera -> 'Ingeniería en Sistemas de
Información'

docente fechaAlta:  (Date fromString: '12/08/1993').
docente cargo: 'Ayudante de primera'.

docente fechaAlta -> 'jueves, 12 de agosto de 1993'
docente cargo -> 'Ayudante de primera'

alumno diasEstudiando -> 764
```

d. Comportamiento saludar de un alumno

Ahora vamos a implementar el método saludar en el Alumno. El alumno debe saludar brindando su nombre y la carrera en la que está inscripto.

Browser de Clases - Clase Alumno - método de instancia
saludar

```
^ Transcript show: 'Hola, mi nombre es ', nombre, ' y estudio
la carrera ', carrera.
```

Vamos al workspace a probar el método:

Workspace

```
alumno := Alumno new.
docente := Docente new.

alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.

docente nombre: 'Carlos'.
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
docente casado.
```

```
alumno fechaInscripcion:  (Date fromString:
'12/03/2021').
alumno carrera: ‘Ingeniería en Sistemas de Información’.

alumno fechaInscripcion -> ‘viernes, 12 de marzo de
2021’
alumno carrera -> ‘Ingeniería en Sistemas de
Información’

docente fechaAlta:  (Date fromString: '12/08/1993').
docente cargo: ‘Ayudante de primera’.

docente fechaAlta -> ‘jueves, 12 de agosto de 1993’
docente cargo -> ‘Ayudante de primera’

alumno diasEstudiando -> 764
alumno saludar -> ‘Hola, mi nombre es Delfina y estudio
la carrera de Ingeniería en Sistemas de Información’
```

e. Comportamiento: antigüedad del docente

Ahora vamos a implementar los comportamientos para el docente. Consta de retornar la cantidad de años de antigüedad en el cargo y de hacer un saludo como docente.

Browser de Clases - Clase Docente - método de instancia *antiguedad*

```
^ Date today year - fechaAlta year
```

Este método le pide el año a la fecha actual y el año a la fecha de alta y luego los resta. Esto nos da la antigüedad. Al implementarlo de esta manera no tenemos en cuenta ni los meses ni los días para calcular la antigüedad, ya que solo se estarían restando los años. Para tenerlos en cuenta podemos implementar la antigüedad utilizando el método #yearsSince:

Browser de Clases - Clase Docente - método de instancia *antiguedad*

```
^ Date today yearsSince: fechaAlta
```

El método #yearsSince: está implementado en la clase Date, el objeto receptor es la fecha de hoy y se le pasa como parámetro otra fecha. Este método devuelve la cantidad de años entre las dos fechas teniendo en cuenta los días y los meses.

Vamos al workspace.

```
Workspace
alumno := Alumno new.
docente := Docente new.

alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.

docente nombre: 'Carlos'.
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
docente casado.

alumno fechaInscripcion:  (Date fromString:
'12/03/2021').
alumno carrera: 'Ingeniería en Sistemas de Información'.

alumno fechaInscripcion -> 'viernes, 12 de marzo de
2021'
alumno carrera -> 'Ingeniería en Sistemas de
Información'

docente fechaAlta:  (Date fromString: '12/08/1993').
docente cargo: 'Ayudante de primera'.

docente fechaAlta -> 'jueves, 12 de agosto de 1993'
docente cargo -> 'Ayudante de primera'

alumno diasEstudiando -> 764
```

```
alumno saludar -> 'Hola, mi nombre es Delfina y estudio
la carrera de Ingeniería en Sistemas de Información'

docente antiguedad -> 29
```

f. Comportamiento saludar del docente

Vamos a implementar el saludo del docente donde tiene que decir su nombre, su cargo y la antigüedad en el mismo.

Browser de Clases - Clase Docente - método de instancia *saludar*

```
^ Transcript show: 'Hola, mi nombre es ', nombre, ' y soy ',
cargo, ' desde hace', self antiguedad, ' años'.
```

Aquí utilizamos la pseudovariable self para poder obtener la antigüedad del docente. La variable self toma el valor del objeto receptor, en este caso es un objeto docente.

Ver: Pseudovariables
Aula virtual: Sección II -> Variables en Smallalk
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=20610&redirect=1

Vamos a probar el método en el workspace:

Workspace

```
alumno := Alumno new.
docente := Docente new.

alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.

docente nombre: 'Carlos'.
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
```

```
docente casado.

alumno fechaInscripcion:  (Date fromString:
'12/03/2021').
alumno carrera: ‘Ingeniería en Sistemas de Información’.

alumno fechaInscripcion -> ‘viernes, 12 de marzo de
2021’
alumno carrera -> ‘Ingeniería en Sistemas de
Información’

docente fechaAlta:  (Date fromString: '12/08/1993').
docente cargo: ‘Ayudante de primera’.

docente fechaAlta -> ‘jueves, 12 de agosto de 1993’
docente cargo -> ‘Ayudante de primera’

alumno diasEstudiando -> 764
alumno saludar -> ‘Hola, mi nombre es Delfina y estudio
la carrera de Ingeniería en Sistemas de Información’

docente antiguedad -> 29
docente saludar -> ‘Hola, mi nombre es Carlos y soy
Ayudante de primera desde hace 29 años’
```

Notar que el método saludar está implementado en la clase Persona, Alumno y Docente. Aquí podemos ver algunos conceptos:

- **Polimorfismo**: “Es la habilidad de dos o más objetos de poder responder a un mensaje con el mismo nombre, cada uno según su propia manera”. En este caso tenemos que los objetos persona, alumno y docente saludan cada uno de una manera diferente, pero respondiendo al mismo mensaje #saludar. Entonces decimos que el método #saludar es polimórfico.
- **Redefinición**: “Es la capacidad de una clase de modificar o redefinir un método que ya ha sido definido en su super-clase (clase padre)”. Cuando se redefine un método en una subclase, la nueva definición reemplaza la definición heredada de la super-clase. Esto significa que al llamar al método en un objeto de la subclase, se ejecutará la definición de la subclase en lugar de la definición de la super-clase. En nuestro caso la clase Alumno y la clase Docente están redefiniendo el método #saudar, por lo tanto al enviar un mensaje a un objeto alumno o a un objeto docente cada uno responderá con su propio saludo.

Ver: Herencia, polimorfismo, redefinición
Aula virtual: Sección I -> Conceptos básicos del Paradigma de OO
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=19983&redirect=1

Aula virtual: Sección I -> Programación orientada a objetos - POO
https://frro.cvg.utn.edu.ar/mod/resource/view.php?id=20555&redirect=1

Workspace completo

```
Workspace
alumno := Alumno new.
docente := Docente new.

alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.

docente nombre: 'Carlos'.
docente genero: 'Masculino'.
docente fechaNacimiento: (Date fromString:
'29/07/1956').
docente casado.

alumno fechaInscripcion:  (Date fromString:
'12/03/2021').
alumno carrera: ‘Ingeniería en Sistemas de Información’.

alumno fechaInscripcion -> ‘viernes, 12 de marzo de
2021’
alumno carrera -> ‘Ingeniería en Sistemas de
Información’

docente fechaAlta:  (Date fromString: '12/08/1993').
docente cargo: ‘Ayudante de primera’.

docente fechaAlta -> ‘jueves, 12 de agosto de 1993’
```

```
docente cargo -> ‘Ayudante de primera’

alumno diasEstudiando -> 764
alumno saludar -> ‘Hola, mi nombre es Delfina y estudio
la carrera de Ingeniería en Sistemas de Información’

docente antiguedad -> 29
docente saludar -> ‘Hola, mi nombre es Carlos y soy
Ayudante de primera desde hace 29 años’
```

Ejercicio práctico: Parte 2

Continuando con la clase Producto, modelamos un sistema para gestionar el funcionamiento de una distribuidora de alimentos.

La distribuidora vende productos alimenticios y bebidas. Además de los atributos definidos en la clase Producto, agregar los siguiente para cada una de sus subclases:

Alimento:

- peso (en gramos)
- fechaVencimiento
- diasAvisoCaducidad

Bebida:

- contenido (en mililitros)
- costoEnvaseRetornable

En el caso de que una bebida no tenga un envase retornable, el costoEnvaseRetornable será cero.

En cuanto a los comportamientos, crear los métodos de acceso y carga datos para ambas clases y además:

1. Alimento
    a. **estaVencido** devuelve true/false de acuerdo a si el producto está vencido o no, comparando la fecha actual con la fecha de vencimiento del alimento.
    b. **proximoAVencer** devuelve true/false verifica si la fecha de caducidad del producto está próxima a vencer o ha caducado, teniendo en cuenta la fecha actual y la fecha de vencimiento restando los días de aviso de caducidad. Notar que si el alimento se encuentra vencido, retornará true.
    c. **mostrar** Imprime en Transcript el nombre, el peso, precio, cantidad en inventario y si está vencido o próximo a vencerse.
2. Bebida
    a. **precioTotal** devuelve el precio de la bebida adicionando al precio el costo del envase.
    b. **mostrar** Imprime en Transcript el nombre, el contenido, el precio total y la cantidad en inventario.

Parte 3 - Ensamble

Vamos a agregar funcionalidad a nuestro sistema de la Universidad. Ahora vamos a modelar las materias cursadas por los alumnos, incluyendo la asignación de una nota obtenida por el alumno en la materia cursada.

De una materia se conoce su nombre, año de cursado, horario y nota.

El comportamiento que queremos implementar es:

- Cargar los datos de una materia.
- Inscribir a un alumno a una materia. Un alumno puede estar inscripto en muchas materias.
- Retornar la cantidad de materias en la que un alumno está inscrito.
- Asignar la nota a una materia que está cursando un alumno.
- Calcular el promedio de un alumno.

a. Creación de la clase Materia

Como primer paso creamos la clase Materia.

Browser de Clases

```
Object subclass: #Materia
    instanceVariableNames: 'nombre anioCursado horario nota'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

b. Crear métodos de acceso a la clase Materia

En segundo lugar creamos los métodos de acceso para cargar los datos de la materia:

Browser de Clases - Clase Materia - método de instancia

```
nombre: unString
nombre := unString

nombre
^ nombre

anioCursado: unEntero
```

```
anioCursado := unEntero

anioCursado
^ anioCursado

horario: unString
horario := unString

horario
^ horario

nota: unNumero
nota := unNumero

nota
^ nota
```

Podemos ir al workspace a crear una materia y asignarle los datos:

```
Workspace
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.
```

Si inspeccionamos la variable materia veremos que tiene asignado los datos que cargamos en el workspace.

c. Inscribir un alumno a una materia

Como primer paso vamos a hacer que un alumno pueda inscribirse a una materia. Para ello vamos a agregar una variable de instancia "materia" en la clase alumno.

Para ello vamos a la clase alumno y agregamos la variable:

```
Browser de Clases
Persona subclass: #Alumno
    instanceVariableNames: 'fechaInscripcion carrera materia'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

Y creamos los métodos de acceso para la variable:

```
Browser de Clases - Clase Alumno - método de instancia
materia: unaMateria
materia := unaMateria

materia
^ materia
```

Vamos al workspace a probar lo que hicimos. Tomamos el objeto alumno creado en el punto anterior y los inspeccionamos, el mismo tendrá todos los datos cargados salvo la materia que como lo agregamos recién, su valor estará en nil.

```
Workspace
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia
```

Lo que hemos implementado es la asignación de una sola materia a un alumno. De esta forma un alumno sólo podrá inscribirse a una única materia, con lo cual no cumplimos con la consigna planteada. Pero vamos a utilizar esto para dar una primera definición de ensamble y luego mejoramos la implementación.

Ensamble: "Es la relación todo-parte o es parte de, en la cual los objetos que presentan los componentes de algún conjunto, se asocian a un objeto que representa el todo".

Ver: Ensamble
Aula virtual: Sección I -> Conceptos básicos del Paradigma de OO
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=19983&redirect=1

Aula virtual: Sección I -> Programación orientada a objetos - POO
https://frro.cvg.utn.edu.ar/mod/resource/view.php?id=20555&redirect=1

En nuestro ejemplo tenemos un ensamble entre la clase Alumno y la clase Materia. Este **ensamble se materializa** cuando asignamos la materia al alumno. Hasta este momento la relación es de 1::1 ya que un alumno tiene una sola materia. Cada objeto alumno y materia tienen sus propios comportamientos. El ensamble es distinto a la herencia ya que el ensamble involucra diferentes objetos donde uno es parte del otro, en este caso la materia es parte del alumno.

En este punto podemos consultar la nota obtenida por el alumno en la materia:

```
Workspace
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8
```

Acá lo que estamos haciendo es pedirle la materia al alumno, y luego pedirle la nota a la materia.

Para que un alumno pueda inscribirse a muchas materias lo que necesitamos es guardar las materias en una colección. Utilizaremos la OrderedCollection para guardar las materias en el alumno.

Ver: Colecciones
Aula virtual: Sección II -> Colecciones

https://frro.cvg.utn.edu.ar/mod/url/view.php?id=22657&redirect=1

Volvemos a la definición de la clase Alumno y cambiamos el nombre de la variable materia (en singular) por materias (en plural).

```
Browser de Clases
Persona subclass: #Alumno
    instanceVariableNames: 'fechaInscripcion carrera materias'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

Si inspeccionamos el objeto alumno, veremos que la variable “materias” está en nil y que la variable “materia” ya no está.

En este momento también borramos los métodos #materia: y #materia que ya no nos sirven porque estaban utilizados por la vieja variable “materia” que ya no está más definida en la clase Alumno.

Inicializar una colección. Vamos a crear un método para asignar la colección a la variable “materias”. Lo haremos con un método llamado “inicializar” ya que necesitamos que la variable materias sea una colección para poder alojar muchas materias, este método lo llamaremos siempre que creemos un objeto alumno.

Creamos el método inicializa en la clase Alumno

```
Browser de Clases - Clase Alumno - método de instancia
inicializa
materias := OrderedCollection new.
```

El método inicializa lo utilizaremos para dar valores iniciales a las variables de instancia, este método lo llamaremos cada vez que se crea un objeto.

Vamos a crear un nuevo objeto alumno y a inicializarlo, por último inicializamos también el objeto alumno que ya tenemos creado en la variable “alumno”.

```
Workspace
materia := Materia new.

materia nombre: ‘Fisica’.
```

```
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.
```

Si inspeccionamos la variable "al2" veremos que todos los atributos de alumno están en nil salvo la variable "materias" que tiene una colección asignada. Si inspeccionamos la variable "alumno" que es donde tenemos el objeto alumno con el que estamos trabajando, veremos que la variable "materias" sigue en nil, esto es porque sobre ese objeto nunca se llamó al método inicializa (ya que el objeto estaba creado anteriormente).

Vamos a corregir esta situación, para ello volvemos al workspace a inicializar al objeto alumno:

```
Workspace
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.
```

```
alumno inicializa.
```

Si ahora inspeccionamos el objeto alumno veremos que tiene asignada la colección en la variable “materias”.

Ahora lo que necesitamos es un método en la clase Alumno para inscribir un alumno a una materia. Lo que hará el método es agregar un objeto Materia a la colección de materias del alumno.

Browser de Clases - Clase Alumno - método de instancia
inscribirEn: unaMateria
```
materias add: unaMateria
```

El método #inscribirEn: recibe como parámetro un objeto Materia, que luego ese objeto es agregado a la colección de materias del alumno.

Vamos al workspace a probarlo:

Workspace
```
materia := Materia new.

materia nombre: ‘Fisica’.
materia anioCursado: 1.
materia horario: ‘Lunes de 10:30 a 13:10’.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.
```

Vamos a crear otra materia y a inscribir al alumno en la nueva materia creada:

```
Workspace
materia := Materia new.

materia nombre: ‘Fisica’.
materia anioCursado: 1.
materia horario: ‘Lunes de 10:30 a 13:10’.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: ‘Algebra’.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30’.
mat2 nota: 6.

alumno inscribirEn: mat2.
```

De esta manera el alumno está inscripto en la materia “Fisica” y en “Algera”, con una nota de 8 y de 6 respectivamente.

Si se inspecciona el objeto alumno se podrá ver la colección con las dos materias a las que está inscripto el alumno.

d. Cantidad de materias que cursa un alumno

Vamos a implementar un método para retornar la cantidad de materias a las que está cursando el alumno:

Browser de Clases - Clase Alumno - método de instancia
materiasCursando

```
^ materias size
```

Para conocer la cantidad de materias que está cursando un alumno, simplemente le pedimos el tamaño de la colección “materias”.

Workspace

```
materia := Materia new.

materia nombre: ‘Fisica’.
materia anioCursado: 1.
materia horario: ‘Lunes de 10:30 a 13:10’.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: ‘Algebra’.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30’.
mat2 nota: 6.
```

```
alumno inscribirEn: mat2.

alumno materiasCursando -> 2
```

e. Promedio de un alumno

Vamos a calcular el promedio de un alumno.

Browser de Clases - Clase Alumno - método de instancia
promedio

```
|acum|
acum := 0.
materias do: [:unaMateria | acum := acum + unaMateria nota].
^ acum / materias size
```

Este método recorre las materias que tiene el alumno y acumula las notas en la variable “acum”. Luego se divide por la cantidad de materias.

Workspace

```
materia := Materia new.

materia nombre: ‘Fisica’.
materia anioCursado: 1.
materia horario: ‘Lunes de 10:30 a 13:10’.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.
```

```
alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: 'Algebra'.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30'.
mat2 nota: 6.

alumno inscribirEn: mat2.

alumno materiasCursando -> 2

alumno promedio -> 7
```

f. Asignar una nota a una materia

Ahora vamos a asignar una nota a una materia. Para ello buscaremos la materia a la cual queremos cambiar la nota y se la asignaremos. Utilizaremos un método de palabra clave con dos argumentos.

Browser de Clases - Clase Alumno - método de instancia

```
asignarNota: unEntero a: unNombreMateria
|mat|
mat := materias detect: [:unaMateria| unaMateria nombre =
unNombreMateria] ifNone: [nil].
mat isNil ifTrue: [^ MessageBox notify: 'El alumno no cursa la
materia solicitada'].
mat nota: unEntero
```

Este método busca la materia por nombre en la colección de materias. Si no la encuentra retorna un mensaje de error. Si la encuentra asigna la nota a la materia encontrada.

Volvemos al workspace y vamos a modificar la nota de "Física" al alumno, y luego volvemos a pedir el promedio:

Workspace

```
materia := Materia new.
```

```
materia nombre: ‘Fisica’.
materia anioCursado: 1.
materia horario: ‘Lunes de 10:30 a 13:10’.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: ‘Algebra’.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30’.
mat2 nota: 6.

alumno inscribirEn: mat2.

alumno materiasCursando -> 2

alumno promedio -> 7

alumno asignarNota: 4 a: ‘Fisica’.
alumno promedio -> (11/2)
```

Notar que se modificó la nota de física del alumno y ahora el promedio da 11/2. El resultado se da como una fracción. Si queremos verlo como un número flotante debemos convertir el resultado del promedio a Float. Lo haremos en el método promedio:

Browser de Clases - Clase Alumno - método de instancia *promedio*

```
|acum|
acum := 0.
materias do: [:unaMateria | acum := acum + unaMateria nota].
^ acum / materias size asFloat
```

Volvemos al workspace a ejecutar nuevamente el promedio:

Workspace

```
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: 'Algebra'.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30'.
mat2 nota: 6.

alumno inscribirEn: mat2.
```

```
alumno materiasCursando -> 2

alumno promedio -> 7

alumno asignarNota: 4 a: 'Fisica'.
alumno promedio -> (11/2)

alumno promedio -> 5.5
```

Workspace completo

```
Workspace
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: 'Algebra'.
mat2 anioCursado: 1.
```

```
mat2 horario: Miercoles de 18:30 a 21:30’.
mat2 nota: 6.

alumno inscribirEn: mat2.

alumno materiasCursando -> 2

alumno promedio -> 7

alumno asignarNota: 4 a: ‘Fisica’.
alumno promedio -> (11/2)

alumno promedio -> 5.5
```

Ejercicio práctico: Parte 3

Vamos a agregar funcionalidad a nuestro sistema de la distribuidora. Los alimentos están compuestos por ingredientes.

De cada ingrediente se conoce: nombre, cantidad (en gramos) y si está certificado en calidad.

Se dice que un alimento es de calidad superior si todos sus ingredientes están certificados.

El comportamiento que queremos implementar es:

1. Métodos de acceso y carga datos a la clase Ingrediente.
2. Agregar ingredientes al alimento.
3. Retornar la cantidad de ingredientes que tiene un alimento.
4. Calcular la cantidad de gramos totales que tienen los ingredientes del alimento. Realizarlo mediante la suma de las cantidades.
5. Retornar si el ingrediente está certificado o no en calidad (retornar true/false)

Parte 4 - Listados, filtros, ordenamientos

Vamos a agregar una funcionalidad en Alumno para:

- Mostrar todas las notas mostrando materia y nota, ordenada por materia ascendiente.
- Mostrar las notas aprobadas (mayor o igual a 6) ordenado por nota descendiente.
- Mostrar las notas reprobadas (menor que 6) ordenadas por nota descendiente.

a. Listado de todas las materias que cursa un alumno

Comenzamos por el listado de todas las materias:

Browser de Clases - Clase Alumno - método de instancia *listarTodasLasMaterias*

```
|ordenadas|
ordenadas := materias asSortedCollection: [:unaMateria
:otraMateria| unaMateria nombre < otraMateria nombre ].

Transcript show: 'Listado de todas las materias: '; cr.
ordenadas do: [:unaMateria |
      Transcript show: unaMateria nombre, ' - nota: ',
unaMateria nota printString; cr ].
```

En este método utilizamos el mensaje #asSortedCollection: que se lo enviamos a la colección de materias, este mensaje se utiliza para crear una nueva colección ordenada a partir del criterio brindado. El criterio corresponde a tomar dos elementos de la colección y aplicarle una condición, en nuestro caso se hace sobre dos materias (que es el contenido de la colección original). Notar que el #asSortedCollection: no modifica la colección original, sino que crea una nueva colección con los mismos elementos pero ordenados, notar también, que el hecho de crear una nueva colección no significa que todos los elementos son copiados, sino que cada elemento es apuntado por la nueva colección creada.

Ver: Colecciones
Aula virtual: Sección II -> Colecciones
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=22657&redirect=1

Vamos al workspace a probar el método:

Workspace

```
materia := Materia new.
```

```
materia nombre: ‘Fisica’.
materia anioCursado: 1.
materia horario: ‘Lunes de 10:30 a 13:10’.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: ‘Algebra’.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30’.
mat2 nota: 6.

alumno inscribirEn: mat2.

alumno materiasCursando -> 2

alumno promedio -> 7

alumno asignarNota: 4 a: ‘Fisica’.
alumno promedio -> (11/2)

alumno promedio -> 5.5

alumno listarTodasLasMaterias -> ‘Listado de todas las
materias:
Algebra - nota: 8
```

```
Fisica - nota: 4’ (ver en ventana de Transcript)
```

b. Listado de materias aprobadas

Continuamos con el listado de materias aprobadas:

```
Browser de Clases - Clase Alumno - método de instancia
listarMateriasAprobadas
|aprobadas ordenadas|
aprobadas := materias select: [:unaMateria | unaMateria nota
>= 6].
ordenadas := aprobadas asSortedCollection: [:unaMateria
:otraMateria| unaMateria nota > otraMateria nota ].

Transcript show: ‘Listado de materias aprobadas: ‘; cr.
ordenadas do: [:unaMateria |
      Transcript show: unaMateria nombre, ‘ - nota: ‘,
unaMateria nota printString; cr ].
```

En este método primero nos filtramos por las materias cuya nota es mayor o igual a 6. Luego continuamos con el ordenamiento y la muestra.

Vamos a probar en el workspace:

```
Workspace
materia := Materia new.

materia nombre: ‘Fisica’.
materia anioCursado: 1.
materia horario: ‘Lunes de 10:30 a 13:10’.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8
```

```
al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: 'Algebra'.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30'.
mat2 nota: 6.

alumno inscribirEn: mat2.

alumno materiasCursando -> 2

alumno promedio -> 7

alumno asignarNota: 4 a: 'Fisica'.
alumno promedio -> (11/2)

alumno promedio -> 5.5

alumno listarTodasLasMaterias -> 'Listado de todas las
materias:
Algebra - nota: 7
Fisica - nota: 4' (ver en ventana de Transcript)

alumno listarMateriasAprobadas -> 'Listado de todas las
materias:
Algebra - nota: 7' (ver en ventana de Transcript)
```

c. Listado de materias reprobadas

Por último, el listado de materias reprobadas:

Browser de Clases - Clase Alumno - método de instancia

```
listarMateriasReprobadas
|reprobadas ordenadas|
reprobadas := materias select: [:unaMateria | unaMateria nota
< 6].
ordenadas := reprobadas asSortedCollection: [:unaMateria
:otraMateria| unaMateria nota > otraMateria nota ].

Transcript show: 'Listado de materias reprobadas: '; cr.
ordenadas do: [:unaMateria |
      Transcript show: unaMateria nombre, ' - nota: ',
unaMateria nota printString; cr ].
```

Este método es muy similar al método anterior, solamente cambia la condición del filtro. Lo probamos en el workspace:

```
Workspace
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: 'Algebra'.
```

```
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30'.
mat2 nota: 6.

alumno inscribirEn: mat2.

alumno materiasCursando -> 2

alumno promedio -> 7

alumno asignarNota: 4 a: 'Fisica'.
alumno promedio -> (11/2)

alumno promedio -> 5.5

alumno listarTodasLasMaterias -> 'Listado de todas las
materias:
Algebra - nota: 7
Fisica - nota: 4' (ver en ventana de Transcript)

alumno listarMateriasAprobadas -> 'Listado de todas las
materias:
Algebra - nota: 7' (ver en ventana de Transcript)

alumno listarMateriasReprobadas -> 'Listado de todas las
materias:
Fisica - nota: 4' (ver en ventana de Transcript)
```

Notar que se repite código en los listados, esto puede ser mejorado. Lo haremos más adelante.

Workspace completo

```
Workspace
materia := Materia new.

materia nombre: 'Fisica'.
materia anioCursado: 1.
materia horario: 'Lunes de 10:30 a 13:10'.
```

```
materia nota: 8.

alumno materia -> nil
alumno materia: materia.
alumno materia -> a Materia

alumno materia nota -> 8

al2 := Alumno new.
al2 inicializa.

alumno inicializa.

alumno inscribirEn: materia.

mat2 := Materia new.
mat2 nombre: 'Algebra'.
mat2 anioCursado: 1.
mat2 horario: Miercoles de 18:30 a 21:30'.
mat2 nota: 6.

alumno inscribirEn: mat2.

alumno materiasCursando -> 2

alumno promedio -> 7

alumno asignarNota: 4 a: 'Fisica'.
alumno promedio -> (11/2)

alumno promedio -> 5.5

alumno listarTodasLasMaterias -> 'Listado de todas las
materias:
Algebra - nota: 7
Fisica - nota: 4' (ver en ventana de Transcript)
```

```
alumno listarMateriasAprobadas -> ‘Listado de todas las
materias:
Algebra - nota: 7’ (ver en ventana de Transcript)

alumno listarMateriasReprobadas -> ‘Listado de todas las
materias:
Fisica - nota: 4’ (ver en ventana de Transcript)
```

Ejercicio práctico: Parte 4

Vamos a agregar una funcionalidad en la clase Alimento para:

1. Mostrar todos los ingredientes del alimento incluyendo su nombre y cantidad.
2. Mostrar todos los ingredientes que están certificados en calidad.
3. Mostrar todos los ingredientes que no están certificados en calidad.

Parte 5: Clase Universidad

Si quisiéramos tener un listado de todos los alumnos que cursan en la universidad, o de todos los docentes que dan clases en la universidad, no podríamos hacerlo con el modelo que tenemos hasta ahora ya que no tenemos en ningún lado el conjunto de alumnos ni de docentes. Para ello vamos a necesitar una nueva clase, llamada Universidad, que es quien va a tener a todos los alumnos y todos los docentes.

Con la clase Persona (y Alumno y Docente como subclases), lo que estamos modelando son los atributos y el comportamiento de una persona individualmente (sea un alumno o un docente). Para trabajarlos como conjunto debemos modelarlo en otra clase, en este caso, el conjunto de alumnos y docentes pertenecen a la universidad, y por ese motivo los asignamos allí. Por otro lado, la clase Universidad define los atributos y comportamientos que va a tener una única universidad, si quisiéramos, por ejemplo, modelar todas las UTN del país, tendríamos que tener una clase “Rectorado” que tenga una colección de universidades.

La funcionalidad que agregaremos es:

- Inicializar las colecciones de alumnos y docentes
- Inscribir un alumno a la universidad
- Dar de alta un docente en la universidad
- Listado de todos los alumnos que concurren a la universidad, ordenado por promedio descendente.
- Listado de todos los docentes que trabajan en la universidad, ordenado por nombre ascendente.
- Crear un menú de opciones para llamar a la funcionalidad implementada.

a. Crear la clase Universidad

```
Browser de Clases
Object subclass: #Universidad
    instanceVariableNames: 'alumnos docentes'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

b. Inicializar colecciones

Lo primero que haremos es inicializar las variables alumnos y docentes como colecciones.

```
Browser de Clases - Clase Universidad - método de instancia
inicializa
alumnos := OrderedCollection new.
docentes := OrderedCollection new.
```

Ya podemos comenzar con el workspace a probar el código:

```
Workspace
universidad := Universidad new.
universidad inicializa.
```

Si se inspecciona la variable “universidad” se observará que las variables alumnos y docentes tienen asignada una colección vacía.

c. Inscribir un alumno en la universidad

Ahora vamos a inscribir un alumno a la universidad. Para ello debemos crear un objeto alumno, asignarle los datos y luego agregarlo a la colección de alumnos. Utilizaremos una forma similar a la que usamos anteriormente para crear alumnos, lo que hacíamos era:

```
alumno := Alumno new.
alumno nombre: 'Delfina'.
alumno genero: 'Femenino'.
alumno fechaNacimiento: (Date fromString: '21/11/2006').
alumno soltero.
```

Esto lo hacíamos directamente en el workspace escribiendo el nombre, género, etc, directamente. Vamos a modificar esa forma para que un usuario pueda ingresar los datos del alumno por medio de ventanas interactivas.

Para ello vamos a crear un método #cargaDatos quien seráa el encargado de cargar los datos de asignación directa a los objetos. Datos de asignación directa son aquellos que se pueden ingresar por medio de un Prompter prompt:, por ejemplo, nombre, fecha de nacimiento, etc. Un dato que no se puede asignar directamente por medio de un prompter es por ejemplo la inscripción a una materia del alumno, ya que no alcanza un prompter, lo que hay que asignar en este caso es un objeto materia que se obtiene de forma diferente.

Por otro lado, nosotros tenemos que la clase Alumno hereda de la clase Persona, y que ciertos datos están en la Persona. Por ello comenzaremos implementando el #cargaDatos primero en Persona y luego extenderemos el método en Alumno.

Browser de Clases - Clase Persona - método de instancia *cargaDatos*

```
|estado|
nombre := Prompter prompt: 'Ingrese nombre de la persona'.
genero := Prompter prompt: 'Ingrese genero'.
fechaNacimiento := Date fromString: (Prompter prompt: 'Ingrese
fecha de Nacimiento DD/MM/YYYY').
estado := (Prompter prompt: 'Ingrese estado civil (C/D/S)')
asUppercase.
estado = 'C'
      ifTrue: [ self casado ]
      ifFalse: [ estado = 'D'
            ifTrue: [ self divorciado ]
            ifFalse: [ self soltero ]].
```

Ahora vamos a la clase Alumno para asignar los datos específicos del alumno.

Browser de Clases - Clase Alumno - método de instancia *cargaDatos*

```
super cargaDatos.
legajo := Prompter prompt: 'Ingrese legajo'.
fechaInscripcion := Date today.
carrera := Prompter prompt: 'Ingrese la carrera'.
```

En este método estamos utilizando la pseudovariable “super”. Esta psudovariable se utiliza para referenciar genéricamente al objeto de la clase que se está desarrollando (en este caso un objeto Alumno), en ese sentido es igual a la pseudovariable “self”, y al llamar a "super", el método que se ejecutará será el de la superclase en lugar del método definido en la propia clase. Es decir, “super cargaDatos” ejecutado desde Alumno, llamará directamente al método #cargaDatos de Persona en lugar del de Alumno. De esta forma podemos reutilizar lo que está implementado en las superclases.

Por otro lado el método #cargaDatos es un método polimórfico, ya que tienen el mismo nombre, y también hace una redefinición, ya que está definiendo un método que está también definido en la superclase.

Notar también que este #cargaDatos no agrega materias, esto es porque la materia debe ser asignada con un objeto de la clase Materia y no directamente por medio de un prompter.

Ahora, podemos volver a la inscripción del alumno a la universidad:

Browser de Clases - Clase Universidad - método de instancia
inscribirAlumno

```
|alumno|
alumno := Alumno new.
alumno inicializa.
alumno cargaDatos.
alumnos add: alumno.
```

En este método se crea un objeto alumno, se inicializa para asignar una colección vacía para las materias. Luego se cargan los datos del alumno y por último se agrega el alumno a la colección de alumnos. De esta manera se completa la inscripción del alumno a la universidad.

Probamos lo desarrollado en el workspace:

Workspace

```
universidad := Universidad new.
universidad inicializa.

universidad inscribirAlumno.
```

Al inspeccionar “universidad” se debería ver el alumno creado en la colección de alumnos.

d. Dar de alta un docente en la universidad

Ahora vamos a dar de alta a un docente en la universidad. Comenzamos implementado el método #cargaDatos del Docente:

Browser de Clases - Clase Docente - método de instancia
cargaDatos

```
super cargaDatos.
fechaAlta := Date fromString: (Prompter prompt: 'Ingrese fecha
de alta (dd/mm/yyyy)').
cargo := Prompter prompt: 'Ingrese el cargo'.
```

Aquí también reutilizamos la carga de datos realizada por la persona, por medio de la utilización de “super cargaDatos”.

Implementamos el método de alta del docente:

Browser de Clases - Clase Universidad - método de instancia
altaDocente

```
|docente|
docente := Docente new.
docente cargaDatos.
docentes add: docente.
```

Notar que el alta del docente es muy similar a la inscripción de un alumno. Para el alta del docente primero se crea un objeto docente, luego se le cargan los datos al docente y por último se agrega el docente a la colección de docentes. En este caso no se llama al método inicializa, ya que el docente no tiene ninguna colección a inicializar.

Probamos lo desarrollado en el workspace:

Workspace

```
universidad := Universidad new.
universidad inicializa.

universidad inscribirAlumno.
universidad altaDocente.
```

e. Listado de alumnos que asisten a la universidad

Ahora vamos a crear un listado de los alumnos que asisten a la universidad.

Browser de Clases - Clase Universidad - método de instancia
listadoAlumnos

```
|alumnosOrd|
alumnosOrd := alumnos asSortedCollection: [ :unAlumno
:otroAlumno |
      unAlumno promedio > unAlumno promedio ].
```

```
Transcript show: 'Listado de todos los alumnos ordenado por
promedio ascendente'; cr.
alumnosOrd do: [:unAlumno |
      Transcript show: 'Nombre: ', unAlumno nombre, ' -
promedio: ', unAlumno promedio printString; cr ].
```

En este método primero se ordena la colección por promedio, luego se pone un título al listado, y por último se imprime el nombre y el promedio de cada alumno

Inscribir alumnos a la universidad y probar el listado.

```
Workspace
universidad := Universidad new.
universidad inicializa.

universidad inscribirAlumno.
universidad altaDocente.

universidad listadoAlumnos. -> "ver en la ventana de
Transcript el resultado"
```

f. Listado de docentes que asisten a la universidad

Continuamos con la implementación del listado de los docentes que trabajan en la universidad.

```
Browser de Clases - Clase Universidad - método de
instancia
listadoDocentes
|docentesOrd|
docentesOrd := docentes asSortedCollection: [ :unDocente
:otroDocente |
      unDocente nombre > unDocente nombre].
Transcript show: 'Listado de docentes ordenado por nombre';
cr.
docentesOrd do: [:unDocente |
      Transcript show: 'Nombre: ', unDocente nombre, ' -
cargo: ', unDocente cargo; cr ].
```

Agregar docentes a la universidad y probar el listado.

```
Workspace
universidad := Universidad new.
universidad inicializa.

universidad inscribirAlumno.
universidad altaDocente.

universidad listadoAlumnos. -> "ver en la ventana de
Transcript el resultado"

universidad listadoDocentes. -> "ver en la ventana de
Transcript el resultado"
```

g. Menú de opciones

Por último vamos a implementar un menú de opciones en la universidad para poder ejecutar las funcionalidades realizadas.

```
Browser de Clases - Clase Universidad - método de
instancia
menu
| op |
op := 5.
[ op = 0 ] whileFalse: [
      MessageBox notify: 'MENU:

1- Inscribir alumno
2- Alta docente
3- Listado de alumnos
4- Listado de docentes
0- Salir'.
      op:= (Prompter prompt:'Ingrese opción:') asNumber
asInteger.
      ( op = 1 ) ifTrue:[ self inscribirAlumno ].
      ( op = 2 ) ifTrue:[ self altaDocente].
      ( op = 3 ) ifTrue: [ self listadoAlumnos ].
      ( op = 4 ) ifTrue: [ self listadoDocentes ]
```

```
    ] .
```

El menú mostrará un mensaje dando las opciones al usuario, luego el usuario debe ingresar el número de opción y se ejecutará el método correspondiente a la selección del usuario.

Workspace completo

```
Workspace
universidad := Universidad new.
universidad inicializa.

universidad inscribirAlumno.
universidad altaDocente.

universidad listadoAlumnos. -> "ver en la ventana de
Transcript el resultado"

universidad listadoDocentes. -> "ver en la ventana de
Transcript el resultado"
```

Ejercicio práctico: Parte 5

Crear la clase Distribuidora, con el atributo “productos” y agregar los siguientes comportamientos:

1. Inicializar la colección de productos. Notar que hay una sola colección donde se agregarán indistintamente alimentos o bebidas (no hacer en dos variables separadas).
2. Crear un nuevo producto.
3. Incrementar el stock de un producto seleccionado.
4. Decrementar el stock de un producto seleccionado.
5. Listado de todos los productos ordenados por cantidad de stock ascendente.
6. Listado de todas las bebidas que tengan envases retornables, ordenadas por precio total descendente.
7. Listado de detalle de alimentos y sus ingredientes, para todos los alimentos que están próximos a vencerse (si está vencido no debe mostrarse en el listado). De cada ingrediente mostrar su nombre y cantidad.
8. Listar todos los productos que tengan un stock inferior a la cantidad mínima requerida. Mostrar el detalle del producto junto con la cantidad mínima necesaria a comprar del mismo.
9. Crear un menú de opciones para llamar a la funcionalidad implementada.

Parte 6: Variables y métodos de Clase

Los docentes se identifican en la universidad por medio de un código, que es un valor numérico incremental y se asigna automáticamente por el sistema.

Una forma de implementar este comportamiento es por medio del uso de una variable de Clase, en donde guardaremos el último código utilizado. Luego al crear un docente, tomaremos ese código y lo incrementaremos para asignarlo al docente recién creado. También deberemos dar un valor inicial al código de donde queremos que inicie a asignar. Recordar que las variables de clases siempre comienzan con mayúscula.

No solamente debe crearse la variable de clase, sino que también el docente tiene que tener un atributo “código” de instancia, para que pueda tener asignado su propio valor.

Comenzaremos por la creación de la variable para luego pasar al comportamiento.

Ver: Variables de clase
Aula virtual: Sección II -> Variables en Smallalk
https://frro.cvg.utn.edu.ar/mod/url/view.php?id=20610&redirect=1

a. Crear variable de Clase

En el browser de clases creamos la variable de clase:

```
Browser de Clases
Persona subclass: #Docente
    instanceVariableNames: 'codigo fechaAlta cargo'
    classVariableNames: 'UltimoCodigo'
    poolDictionaries: ''
    category: 'Ejercicio POO'
```

b. Comportamiento de clase para inicializar último código

Crear comportamientos de clase para:

- Dar un valor inicial a la variable UltimoCodigo
- Incrementar en una unidad el UtlimoCodigo
- Retornar el UltimoCodigo

Los métodos a crear son los siguientes, tener en cuenta de realizarlo en Smalltalk seleccionando que son métodos de clase.

Browser de Clases - Clase Docente - métodos de clase

inicializaUltimoCodigo

```
UltimoCodigo := 0
```

incrementaUltimoCodigo

```
UltimoCodigo := UltimoCodigo + 1
```

ultimoCodigo

```
^ UltimoCodigo
```

Lo probamos en el workspace, los métodos de clase se envían siempre a las clases.

Workspace

```
Docente ultimoCodigo -> nil. “Porque todavia no lo
inicializamos”
Docente inicializaUltimoCodigo.
Docente ultimoCodigo -> 0.
Docente incrementaUltimoCodigo.
Docente ultimoCodigo -> 1.
```

c. Implementación asignación automática del código

Para la asignación automática podemos modificar el método #cargaDatos del Docente para que tome el codigo desde esta variable de clase:

Browser de Clases - Clase Docente - método de instancia

cargaDatos

```
super cargaDatos.
Docente incrementaUltimoCodigo.
codigo := Docente ultimoCodigo.
fechaAlta := Date fromString: ‘Ingrese fecha alta docente
(dd/mm/yyyy)’.
cargo := Prompter prompt: ‘Ingrese cargo docente’.
```

Notar que la inicialización de UltimoCodigo no debe realizarse en el #cargaDatos del Docente, ya que si lo hiciéramos aquí cada vez que se agrega un docente se estaría inicializando el

UltimoCodigo. Esta inicialización debe realizarse una única vez desde la Universidad cuando la misma se inicializa.

Browser de Clases - Clase Universidad - método de instancia

inicializa

```
alumnos := OrderedCollection new.
docentes := OrderedCollection new.
Docente inicializaUltimoCodigo.
```

Ejercicio práctico: Parte 6

Agregar la siguiente funcionalidad, agregando las variables de instancia y de clase que sean necesarias:

1. Los productos se identifican por medio de un código, que es un valor numérico incremental y se asigna automáticamente por el sistema.
2. Los productos cuentan con un atributo (true/false) para determinar si el mismo es destacado o no, el hecho de ser destacado hace que el producto reciba un descuento que es igual para todos los productos.

Anexo: Resolución caso práctico Distribuidora

Ejercicio práctico: Parte 1

```
Browser de Clases
Object subclass: #Producto
    instanceVariableNames: 'nombre descripcion precio stock'
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
```

a. Crear métodos de acceso para la clase Producto.

```
Browser de Clases - Clase Producto - método de instancia
nombre: unString
nombre := unString

nombre
^ nombre

descripcion: unString
descripcion := unString

descripcion
^ descripcion

precio: unNumero
precio := unNumero

precio
^ precio

stock: unNumero
stock := unNumero

stock
^ stock
```

```
cargaDatos
nombre := Prompter prompt: 'Ingrese el nombre del producto'.
descripcion := Prompter prompt: 'Ingrese la descripcion del
producto'.
precio := (Prompter prompt: 'Ingrese el precio del producto')
asNumber.
stock := (Prompter prompt: 'Ingrese el stock del producto')
asNumber.
```

b. Retornar el monto total en dinero que se tiene de productos en inventario. Este método retorna un valor numérico que resulta de multiplicar el precio por la cantidad en inventario.

```
Browser de Clases - Clase Producto - método de instancia
costoInventario
^ stock * precio
```

c. Verificar disponibilidad: se le pasa por parámetro la cantidad de productos que se requieren y verifica si hay disponibilidad de dicha cantidad de productos. El método retorna true/false en caso de tener o no disponibilidad.

```
Browser de Clases - Clase Producto - método de instancia
hayDisponibilidadPara: unaCantidad
^ stock >= unaCantidad
```

d. Incrementar cantidad en inventario. Permite aumentar la cantidad del producto en el inventario. Este método tiene como parámetro la cantidad y se debe adicionar a la cantidad en inventario existente. Este método se utilizará luego para la implementación de compra de productos.

```
Browser de Clases - Clase Producto - método de instancia
incrementarStock: unaCantidad
stock := stock + unaCantidad
```

e. Disminuir la cantidad en inventario. Permite disminuir la cantidad del producto en el inventario. Este método tiene como parámetro la cantidad y se debe restar de la cantidad en inventario existente, para ello hay que verificar previamente si hay disponibilidad del producto para la cantidad que se quiere disminuir. Este método se utilizará en un futuro para la implementación de la venta de productos.

Browser de Clases - Clase Producto - método de instancia *decrementarStock: unaCantidad*

```
(self hayDisponibilidadPara: unaCantidad)
       ifFalse: [ ^ MessageBox notify: 'No hay stock
suficiente para disminuir la cantidad ingresada' ].

stock := stock - unaCantidad
```

Notar que: primero se hace la validacion de que se puede restar la cantidad, en caso de que no haya stock sufciciente se muestra un mensaje al usuario y se termina la ejecucion del método. El retorno (^) corta la ejecucion del método.

f. Verificar cantidad mínima en inventario: es un método que retorna verdadero o falso, comparando la cantidad en inventario contra la cantidad mínima en inventario. (Notar que hay que agregar una variable a la clase Producto que modele la cantidad mínima en inventario para poder realizarlo).

Browser de Clases

```
Object subclass: #Producto
    instanceVariableNames: 'nombre descripcion precio stock
cantidadMínimaStock'
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
```

Browser de Clases - Clase Producto - método de instancia *hayCantidadMínimaEnStock*

```
^ stock > cantidadMinimaStock
```

g. Retornar la cantidad de productos a comprar para alcanzar la cantidad mínima. Este método retorna un número entero.

```
Browser de Clases - Clase Producto - método de instancia
cantidadMínimaStockAComprar
^ (cantidadMinimaStock - stock) max: 0
```

Notar que: Si el stock es mayor a la cantidad mínima, la resta nos dará un valor negativo, el uso del método #max: evita retornar un número negativo en esos casos.

Ejercicio práctico: Parte 2

```
Browser de Clases
Producto subclass: #Alimento
    instanceVariableNames: 'peso fechaVencimiento
diasAvisoCaducidad'
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
```

```
Browser de Clases
Producto subclass: #Bebida
    instanceVariableNames: 'cantidad costoEnvaseRetornable'
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
```

Métodos de acceso:

```
Browser de Clases - Clase Alimento - método de instancia
peso: unNumero
peso := unNumero

peso
^ peso

fechaVencimiento: unaFecha
```

```
fechaVencimiento := unaFecha

fechaVencimiento
^ fechaVencimiento

diasAvisoCaducidad: unNumero
diasCaducidad := unNumero

diasAvisoCaducidad
^ diasCaducidad

cargaDatos
super cargaDatos.
peso := (Prompter prompt: 'Ingrese el peso del producto en
gramos') asNumber.
fechaVencimiento := Date fromString: (Prompter prompt:
'Ingrese fecha de vencimiento en formato DD/MM/AAAA').
diasAvisoCaducidad := (Prompter prompt: 'Ingrese los dias para
aviso de caducidad') asNumber.
```

Browser de Clases - Clase Bebida - método de instancia

```
cantidad: unNumero
cantidad := unNumero

cantidad
^ cantidad

costoEnvaseRetornable: unNumero
costoEnvaseRetornable := unNumero

costoEnvaseRetornable
^ costoEnvaseRetornable

cargaDatos
super cargaDatos.
cantidad := (Prompter prompt: 'Ingrese el peso del producto en
mililitros') asNumber.
costoEnvaseRetornable := (Prompter prompt: 'Ingrese el costo
del envase retornable') asNumber.
```

1. Alimento
    a. **estaVencido** devuelve true/false de acuerdo a si el producto está vencido o no, comparando la fecha actual con la fecha de vencimiento del alimento.

Browser de Clases - Clase Alimento - método de instancia *estaVencido*

```
^ Date today > fechaVencimiento
```

    b. **proximoAVencer** devuelve true/false verifica si la fecha de caducidad del producto está próxima a vencer o ha caducado, teniendo en cuenta la fecha actual y la fecha de vencimiento restando los días de aviso de caducidad. Notar que si el alimento se encuentra vencido, retornará true.

Browser de Clases - Clase Alimento - método de instancia *proximoAVencer*

```
^ Date today > (fechaVencimiento subtractDays:
diasAvisoCaducidad)
```

    c. **mostrar** Imprime en Transcript el nombre, el peso, precio, cantidad en inventario y si está vencido o próximo a vencerse.

Browser de Clases - Clase Alimento - método de instancia *mostrar*

```
^ Transcript show: 'Alimento: ', nombre, ' peso: ', peso
printString, '(gr) precio: $', precio printString, ' stock: ',
stock printString, ' proximo a vencer: ', self
estadoProximoAVencer.

estadoProximoAVencer
(self proximoAVencer)
        ifTrue: [^'si']
        ifFalse: [^'no'].
```

Notar que: En lugar de mostrar true/false, creamos el método #estadoProximoAVencer para mostrar la información de una manera más amigable al usuario.

3. Bebida
    a. **precioTotal** devuelve el precio de la bebida adicionando al precio el costo del envase.

```
Browser de Clases - Clase Bebida - método de instancia
precioTotal
^ precio + costoEnvaseRetornable
```

    b. **mostrar** Imprime en Transcript el nombre, el contenido, el precio total y la cantidad en inventario.

```
Browser de Clases - Clase Bebida - método de instancia
mostrar
^ Transcript show: Bebida: ‘, nombre, ‘ cantidad: ‘, cantidad
printString, ‘(ml) precio: $‘, self precioTotal printString, ‘
stock: ’, stock printString.
```

Ejercicio práctico: Parte 3

```
Browser de Clases
Object subclass: #Ingrediente
    instanceVariableNames: 'nombre cantidad estaCertificado'
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
```

1. Métodos de acceso y carga datos a la clase Ingrediente.

```
Browser de Clases - Clase Ingrediente - métodos de
instancia
nombre: unString
nombre := unString
```

```
nombre
^ nombre

cantidad: unNumero
cantidad := unNumero

cantidad
^ cantidad

estaCertificado: unBooleano
estaCertificado := unBooleano

estaCertificado
^ estaCertificado

cargaDatos
nombre := Prompter prompt: ‘Ingrese el nombre del
ingrediente’.
cantidad := (Prompter prompt: ‘Ingrese la cantidad del
ingrediente’) asNumber.
estaCertificado := MessageBox confirm: ‘¿El ingrediente esta
certificado?’.
```

2. Agregar ingredientes al alimento.

```
Browser de Clases
Object subclass: #Producto
    instanceVariableNames: 'nombre descripcion precio stock
cantidadMínimaStock ingredientes'
    classVariableNames: ''
    poolDictionaries: ''
    category: ''
```

```
Browser de Clases - Clase Alimento - métodos de
instancia
inicializa
ingredientes := OrderedCollection new.
```

Notar que: Se agrega el atributo “ingredientes” al alimento y el metodo inicializa para inicializar la colección de alimentos.

Browser de Clases - Clase Alimento - métodos de instancia

agregarIngredientes

```
|ing continua|
continua := true.
[ continua ] whileTrue: [
    ing := Ingrediente new.
    ing cargaDatos.
    ingredientes add: ing.
    continua := MessageBox confirm: ‘¿Desea continuar
agregando ingredientes?’.
 ].
```

cargaDatos

```
super cargaDatos.
peso := (Prompter prompt: ‘Ingrese el peso del producto en
gramos’) asNumber.
fechaVencimiento := Date fromString: (Prompter prompt:
‘Ingrese fecha de vencimiento en formato DD/MM/AAAA’).
diasAvisoCaducidad := (Prompter prompt: ‘Ingrese los dias para
aviso de caducidad’) asNumber.
self agregarIngredientes.
```

Notar que: Se actualiza el método carga datos para agregar los ingredientes en el momento de la carga de datos del alimento.

3. Retornar la cantidad de ingredientes que tiene un alimento.

Browser de Clases - Clase Alimento - métodos de instancia

cantidadIngredientes

```
^ ingredientes size
```

4. Calcular la cantidad de gramos totales que tienen los ingredientes del alimento. Realizarlo mediante la suma de las cantidades.

Browser de Clases - Clase Alimento - métodos de instancia

cantidadGramosIngredientes

```
|acum|
acum := 0.
ingredientes do: [:unIng| acum := acum + unIng cantidad ].
^ acum
```

5. Retornar si el alimento está certificado o no en calidad (retornar true/false)

Browser de Clases - Clase Alimento - métodos de instancia

estaCertificado

```
ingredientes detect: [:unIng| unIng estaCertificado ] ifNone:
[ ^ false ].
^ true
```

Ejercicio práctico: Parte 4

1. Mostrar todos los ingredientes del alimento incluyendo su nombre y cantidad.

Browser de Clases - Clase Alimento - métodos de instancia

mostrarIngredientes

```
Transcript show: 'Ingredientes: '; cr.
ingredientes do: [:unIng|
          Transcript show: unIng cantidad printString,' grs.
', unIng nombre; cr ].
Transcript show: 'Total gramos ingredientes: ', self
cantidadGramosIngredientes printString.
```

Notar que: La muestra final de la cantidad de gramos no es requerida por el enunciado pero se agrega a modo de ejemplo de cómo mostrar datos totales.

2. Mostrar todos los ingredientes que están certificados en calidad.

Browser de Clases - Clase Alimento - métodos de instancia

mostrarIngredientesCertificatos

```
|certificados|
Transcript show: ‘Ingredientes certificados: ‘; cr.
certificados := ingredientes select: [:unIng| unIng
estaCertificado ].
certificados do: [:unIng|
          Transcript show: unIng cantidad printString,‘ grs.
‘, unIng nombre; cr ].
```

3. Mostrar todos los ingredientes que no están certificados en calidad.

Browser de Clases - Clase Alimento - métodos de instancia

mostrarIngredientesNoCertificatos

```
|certificados|
Transcript show: ‘Ingredientes no certificados: ‘; cr.
noCertificados := ingredientes reject: [:unIng| unIng
estaCertificado ].
certificados do: [:unIng|
          Transcript show: unIng cantidad printString,‘ grs.
‘, unIng nombre; cr ].
```

Ejercicio práctico: Parte 5

Browser de Clases

```
Object subclass: #Distribuidoras
    instanceVariableNames: 'productos'
    classVariableNames: ''
    poolDictionaries: ''
```

```
        category: ''
```

1. Inicializar la colección de productos. Notar que hay una sola colección donde se agregarán indistintamente alimentos o bebidas (no hacer en dos variables separadas).

Browser de Clases - Clase Distribuidora - métodos de instancia

inicializa

```
productos := OrderedCollection new.
```

2. Crear un nuevo producto.

Browser de Clases - Clase Distribuidora - métodos de instancia

altaProducto

```
|tipo prod|
[ tipo asUppercase = 'A' or: [ tipo asUppercase = 'B' ]]
whileFalse: [
      tipo := Prompter prompt: 'Ingrese tipo de producto a
cargar, (A) Alimento o (B) Bebida' ].

(tipo asUppercase = 'A') ifTrue: [ prod := Alimento new ].
(tipo asUppercase = 'B') ifTrue: [ prod := Bebida new ].
prod cargaDatos.
productos add: prod.
```

3. Incrementar el stock de un producto seleccionado.

Browser de Clases - Clase Distribuidora - métodos de instancia

incrementarStock

```
|prod cant|
prod := self buscaProducto.
prod isNil ifTrue: [^ MessageBox notify: 'No se encuentra el
producto, intente nuevamente.'].
```

```
cant := (Prompter prompt: 'Ingrese la cantidad de stock a
incrementar') asNumber.
prod incrementarStock: cant.

buscaProducto
|nombre prod|
nombre := Prompter prompt: 'Ingrese el nombre del producto a
buscar'.
prod := productos detect: [:unProd | unProd nombre = nombre ]
ifNone: [nil].
^ prod
```

Notar que: La busqueda del producto la realizamos en un método separado. Esta búsqueda puede devolver el producto o si no lo devolverá nil. El producto obtenido puede ser un alimento o una bebida, en ambos casos se incrementará el stock de la misma manera. En caso de que obtengamos un nil, se mostrará el mensaje al usuario y se corta la ejecución del método (porque hay un retorno (^)).

4. Decrementar el stock de un producto seleccionado.

Browser de Clases - Clase Distribuidora - métodos de instancia

```
decrementarStock
|prod cant|
prod := self buscaProducto.
prod isNil ifTrue: [^ MessageBox notify: 'No se encuentra el
producto, intente nuevamente.'].

cant := (Prompter prompt: 'Ingrese la cantidad de stock a
decrementar') asNumber.
prod decrementarStock: cant.
```

Notar que: Los métodos para incrementar y decrementar stock son prácticamente iguales, se puede optimizar sacando el código repetido en otros métodos.

5. Listado de todos los productos ordenados por cantidad de stock ascendente.

Browser de Clases - Clase Distribuidora - métodos de instancia

listadoProductos

```
|ordenados|
ordenados := productos asSortedCollection: [:unProd :otroProd
| unProd stock < unProd stock ].

Transcript show: 'Productos ordenados por stock: '; cr.
ordenados do: [:unProd|
    unProd mostrar.
    Transcript cr.
].
```

6. Listado de todas las bebidas que tengan envases retornables, ordenadas por precio total descendente.

Browser de Clases - Clase Distribuidora - métodos de instancia

listadoBebidasRetornables

```
|bebidas|
bebidas := productos select: [:unProd | unProd esBebida and: [
unProd costoEnvaseRetornable > 0] ].
bebidas := bebidas asSortedCollection: [:unaBeb :otraBeb |
unaBeb precioTotal > otraBeb precioTotal ].

Transcript show: 'Bebidas con envases retornables: '; cr.
bebidas do: [:unaBeb|
    unaBeb mostrar.
    Transcript cr.
].
```

Browser de Clases - Clase Producto - métodos de instancia

esBebida

```
^ false
```

Browser de Clases - Clase Bebida - métodos de instancia

```
esBebida
^ true
```

Notar que: Para saber si un producto es una bebida o no, se crea en la clase Producto un método esBebida que retorna false, luego se redefine ese método en la clase Bebida para retornar true. De esta manera todas las bebidas retornarán true, mientras que lo que no sea una bebida retoranará false.

7. Listado de detalle de alimentos y sus ingredientes, para todos los alimentos que están próximos a vencerse (si está vencido no debe mostrarse en el listado). De cada ingrediente mostrar su nombre y cantidad.

Browser de Clases - Clase Distribuidora - métodos de instancia

```
listadoAlimentosProximosVencer
|alimentos|
alimentos := productos select: [:unProd |
            unProd esAlimento and: [
            unProd proximoAVencer and: [
            unProd estaVencido not ]]].

Transcript show: 'Alimentos próximos a vencer: '; cr.
alimentos do: [:unAlim|
    unAlim mostrar.
    Transcript cr.
].
```

Browser de Clases - Clase Producto - métodos de instancia

```
esAlimento
^ false
```

Browser de Clases - Clase Alimento - métodos de instancia

```
esAlimento
```

```
    ^ true
```

Notar que: Se utiliza la misma técnica utilizada para la bebida, para saber si un producto es un alimento.

8. Listar todos los productos que tengan un stock inferior a la cantidad mínima requerida. Mostrar el detalle del producto junto con la cantidad mínima necesaria a comprar del mismo.

Browser de Clases - Clase Distribuidora - métodos de instancia

listadoProductosPocoStock

```
|prods|
prods := productos select: [:unProd |
            unProd hayCantidadMínimaEnStock not ].

Transcript show: 'Productos con poco stock: '; cr.
prods do: [:unProd|
    unProd mostrar.
    Transcript show: ' Cantidad minima a comprar: ', unProd
cantidadMínimaStockAComprar printString; cr.
].
```

9. Crear un menú de opciones para llamar a la funcionalidad implementada.

Browser de Clases - Clase Distribuidora - método de instancia

menu

```
| op |
[ op = 0 ] whileFalse: [
      MessageBox notify: 'MENU:

1- Alta producto.
2- Incrementar Stock producto.
3- Decrementar Stock producto.
4- Listado productos.
5- Listado bebidas retornables.
```

```
6- Listado alimentos proximos a vencer.
7- Listado productos con poco stock.

0- Salir'.
      op:= (Prompter prompt:'Ingrese opción:') asNumber
asInteger.
      ( op = 1 ) ifTrue:[ self altaProducto ].
      ( op = 2 ) ifTrue:[ self incrementarStock].
      ( op = 3 ) ifTrue: [ self decrementarStock ].
      ( op = 4 ) ifTrue: [ self listadoProductos ].
      ( op = 5 ) ifTrue: [ self listadoBebidasRetornables ].
      ( op = 6 ) ifTrue: [ self listadoAlimentosProximosVencer
].
      ( op = 7 ) ifTrue: [ self listadoProductosPocoStock ].
 ] .
```

Ejercicio práctico: Parte 6

1. Los productos se identifican por medio de un código, que es un valor numérico incremental y se asigna automáticamente por el sistema.

Browser de Clases

```
Object subclass: #Producto
    instanceVariableNames: 'codigo nombre descripcion precio
stock cantidadMinimaStock ingredientes'
    classVariableNames: 'UltimoCodigo'
    poolDictionaries: ''
    category: ''
```

Browser de Clases - Clase Producto - métodos de clase

inicializaUltimoCodigo

```
UltimoCodigo := 0
```

incrementaUltimoCodigo

```
UltimoCodigo := UltimoCodigo + 1
```

```
ultimoCodigo
^ UltimoCodigo
```

Browser de Clases - Clase Producto - método de instancia

```
cargaDatos
Producto incrementaUltimoCodigo.
codigo := Producto ultimoCodigo.
nombre := Prompter prompt: 'Ingrese el nombre del producto'.
descripcion := Prompter prompt: 'Ingrese la descripcion del
producto'.
precio := (Prompter prompt: 'Ingrese el precio del producto')
asNumber.
stock := (Prompter prompt: 'Ingrese el stock del producto')
asNumber.
```

2. Los productos cuentan con un atributo (true/false) para determinar si el mismo es destacado o no, el hecho de ser destacado hace que el producto reciba un descuento que es igual para todos los productos.

Browser de Clases

```
Object subclass: #Producto
    instanceVariableNames: 'codigo nombre descripcion precio
stock cantidadMinimaStock ingredientes esDestacado'
    classVariableNames: 'UltimoCodigo Descuento'
    poolDictionaries: ''
    category: ''
```

Browser de Clases - Clase Producto - métodos de clase

```
inicializaDescuento
Descuento := (Prompter prompt: 'Ingrese el monto del descuento
(valor entre 0 y 100)') asNumber.

descuento
^ Descuento
```

Browser de Clases - Clase Producto - método de instancia

esDestacado: unBooleano

```
esDestacado := unBooleano
```

esDestacado

```
^ esDestacado
```

cargaDatos

```
Producto incrementaUltimoCodigo.
codigo := Producto ultimoCodigo.
nombre := Prompter prompt: 'Ingrese el nombre del producto'.
descripcion := Prompter prompt: 'Ingrese la descripcion del
producto'.
precio := (Prompter prompt: 'Ingrese el precio del producto')
asNumber.
stock := (Prompter prompt: 'Ingrese el stock del producto')
asNumber.
esDestacado := (MessageBox confirm: '¿El producto es
destacado?'.
```

destacar

```
esDestacado := true
```

quitarDestacado

```
esDestacado := false
```

precioFinal

```
self esDestacado
    ifTrue: [ ^ self precioTotal + self precioTotal *
(Descuento / 100)]
    ifFalse: [ ^ self precioTotal ].
```

Browser de Clases - Clase Alimento - métodos de instancia

precioTotal

```
^ precio
```

Notar que: Para el cálculo del precio final, se crea un método #precioTotal en la clase Alimentos para que sea polimórfico con la Bebida, que ya tenía un #precioTotal creado para tener en cuenta el precio de envase.