

02

DEFINICIÓN DE SITUACIONES Y MANEJO DE CASOS



1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Modelar las características de un **objeto**, utilizando nuevos tipos simples de datos y la técnica de definir constantes y enumeraciones para representar los valores posibles de un **atributo**.
- Utilizar expresiones como medio para identificar una situación posible en el estado de un **objeto** y para indicar la manera de modificar dicho estado.
- Utilizar las instrucciones condicionales simples y compuestas como parte del cuerpo de un **método**, para poder considerar distintos casos en la solución de un problema.
- Identificar de manera informal los métodos de una **clase**, utilizando para esto la técnica de agrupar los métodos por tipo de **responsabilidad** que tienen: construir, modificar o calcular.

2. Motivación

En el nivel anterior se introdujo la noción de un programa como la solución a un problema planteado por un cliente. Para construir dicho programa, se presentaron y utilizaron los elementos conceptuales, tecnológicos y metodológicos necesarios para enfrentar problemas triviales. A medida que los problemas comienzan a ser más complejos, es preciso ir extendiendo dichos elementos. En este nivel vamos a introducir nuevos elementos en tres direcciones:

1. Nuevas maneras de modelar una característica.
2. La posibilidad de considerar casos alternativos en el cuerpo de un [método](#).
3. Algunas técnicas para identificar los métodos de una [clase](#).

En los siguientes párrafos se muestra la necesidad de estas extensiones dentro del proceso de desarrollo de programas.

¿Por qué necesitamos nuevas maneras de modelar una característica? Aunque con los tipos de datos para manejar enteros, reales y cadenas de caracteres se puede cubrir un amplio espectro de casos, en este nivel veremos nuevos tipos de datos y nuevas técnicas para representar las características de las clases. También aprovecharemos para profundizar en los tipos de datos estudiados en el nivel anterior.

¿Por qué es necesario poder considerar casos en el cuerpo de un [método](#)? Con las instrucciones que se presentaron en el nivel anterior, sólo es posible asignar un valor a un [atributo](#), pedir un servicio a un [objeto](#) con el cual se tiene una [asociación](#), o retornar un resultado. Por ejemplo, si en el caso del empleado del nivel 1 existiera una norma de la empresa por la que se diera una bonificación en el salario a aquellos empleados que llevan más de 10 años trabajando con ellos, sería imposible incluirla en el programa. No habría manera de verificar si el empleado cumple con esa [condición](#) para sumarle la bonificación al salario. Allí habría dos casos distintos, cada uno con un [algoritmo](#) diferente para calcular el salario.

¿Por qué necesitamos técnicas para clasificar los métodos de una [clase](#)? Uno de los puntos críticos de la programación orientada por objetos es lo que se denomina la [asignación](#) de responsabilidades. Dado que la solución del problema se divide entre muchos algoritmos repartidos por todas las clases (que pueden ser centenares), es importante tener clara la manera de definir quién debe hacer qué. En el nivel 4 nos concentraremos en discutir en detalle este punto; por el momento, vamos a sentar las bases para poder avanzar en esa dirección.

Además de los nuevos elementos antes mencionados, en este nivel trataremos de reforzar y completar algunas de las habilidades generadas en el lector en el nivel anterior. La programación, más que una actividad basada en el conocimiento de enormes cantidades de conceptos y definiciones, es una actividad de habilidades, utilizables en múltiples contextos. Por eso, en la estructura de este libro, se le da mucha importancia a las tareas, cuyo objetivo es trabajar en la manera de usar los conceptos que se van viendo.

3. El Primer Caso de Estudio

En este caso, tenemos un programa que permite manejar el inventario de una pequeña tienda, conocer cuánto dinero hay en caja y tener un control de estadísticas de venta.

La tienda maneja cuatro productos, para cada uno de los cuales se debe manejar la siguiente información:

1. Nombre. No pueden haber dos productos con el mismo nombre.
2. Tipo (puede ser un producto de papelería, de supermercado o de droguería).
3. Cantidad actual del producto en la tienda (número de unidades disponibles para la venta que hay en la bodega).
4. Cantidad mínima para abastecimiento (número de productos por debajo del cual se puede hacer un nuevo pedido al proveedor).
5. El precio base de venta por unidad.

Para calcular el precio final de cada producto, se deben sumar los impuestos que define la ley (IVA). Dichos impuestos dependen del tipo del producto, de la siguiente manera:

- Papelería: 16%
- Supermercado: 4%
- Droguería: 12%.

Eso quiere decir que si un lápiz tiene un precio base de \$10, el precio final será de \$11,6 considerando que un lápiz es un producto de papelería, y sobre estos se debe pagar el 16% de impuestos.

El programa de manejo de esta tienda debe permitir las siguientes operaciones:

1. Vender un producto.
2. Abastecer la tienda con un producto.
3. Cambiar un producto.
4. Calcular estadísticas de ventas la tienda. Dichas estadísticas son: (a) el producto más vendido, (b) El producto menos vendido, (c) la cantidad total de dinero obtenido por las ventas de la tienda, (d) la cantidad de dinero promedio obtenido por unidad de producto vendida.

3.1. Comprensión del Problema

Tal como planteamos en el nivel anterior, el primer paso para poder resolver un problema es entenderlo. Este entendimiento lo mostramos descomponiendo el problema en tres aspectos: los requerimientos funcionales, el modelo conceptual y los requerimientos no funcionales. En la primera tarea de este nivel trabajaremos los dos primeros puntos.

Tarea 1



Objetivo: Entender el problema del caso de estudio de la tienda.

1. Lea detenidamente el enunciado del caso de estudio de la tienda.
2. Identifique y complete la documentación de los cuatro requerimientos funcionales.
3. Construya un primer diagrama de clases con el modelo conceptual, en el que sólo aparezcan las clases, las asociaciones y los atributos sin tipo.

Requerimiento Funcional 1

Nombre	R1 – Vender un producto.
Resumen	Permite vender una cantidad dada de unidades de un producto.
Entradas	(1) el nombre del producto, (2) la cantidad de unidades.
Resultado	Si había suficiente cantidad de producto en bodega, se vende (disminuye en bodega) la cantidad total pedida por el cliente. Si no, se vende (disminuye en bodega) la cantidad total existente en bodega. Se guarda en la caja de la tienda el dinero resultado de la venta. Se informa al usuario la cantidad de unidades vendidas.

Requerimiento Funcional 2

Nombre	R2 – Abastecer la tienda con un producto.
Resumen	Se abastece la tienda con la cantidad de unidades indicada por el usuario. El abastecimiento sólo se puede realizar si la cantidad de productos en bodega es menor que la cantidad mínima del producto.
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	R3 – Cambiar un producto.
Resumen	Permite cambiar la información de un producto vendido en la tienda.
Entradas	1) Nombre actual, 2) Nuevo nombre, 3) Tipo, 4) Valor unitario, 5) Cantidad en bodega, 6) Cantidad mínima.
Resultado	Se actualiza la información del producto.

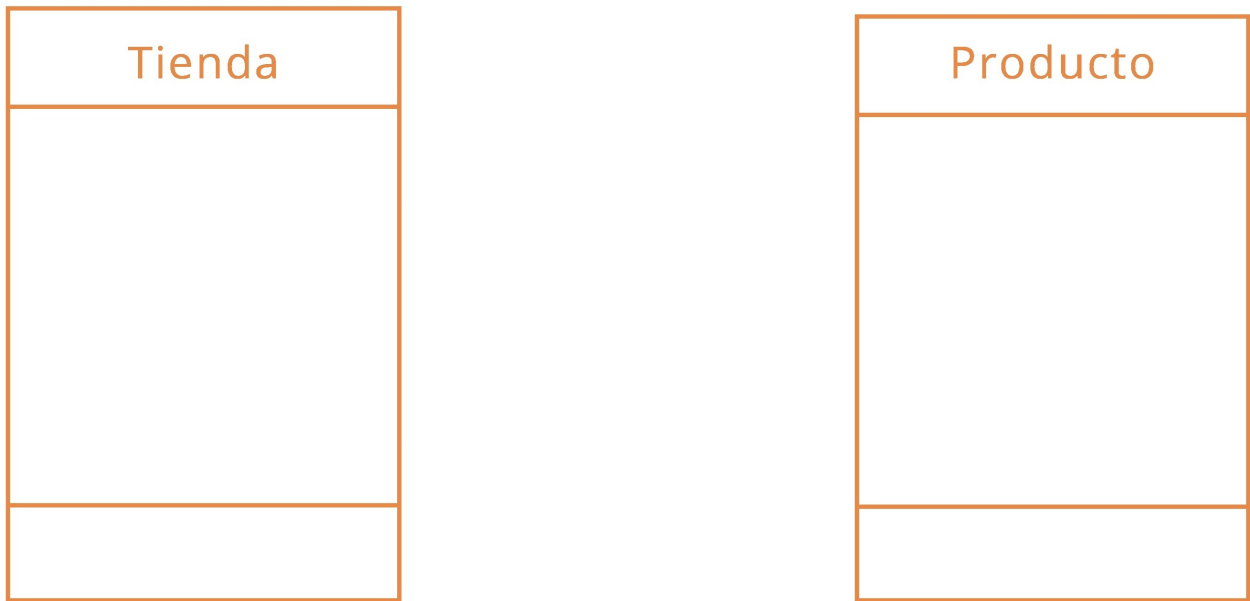
Requerimiento Funcional 4

Nombre	R4 – Calcular estadísticas de ventas.
Resumen	Calcula y muestra las siguientes estadísticas: (a) el producto más vendido; (b) el producto menos vendido; (c) la cantidad total de dinero obtenido por las ventas de la tienda; (d) el promedio de ventas de la tienda.
Entradas	Ninguna.
Resultado	Se muestra la información estadística de ventas.

Modelo conceptual:

En el enunciado se identifican dos entidades: la tienda y el producto. Defina los atributos de cada una de ellas, sin especificar por ahora su tipo.

Dibuje las asociaciones entre las clases y asigne a cada **asociación** un nombre y una dirección.



3.2 Definición de la Interfaz de Usuario

El **diseño** de la **interfaz de usuario** es una de las actividades que debemos realizar como parte del **diseño** de la solución al problema. En la **figura 2.1** presentamos el **diseño** que decidimos para la interfaz del caso de estudio.

Fig. 2.1 Interfaz de usuario para el caso de estudio de la tienda

Lapiz		Aspirina	
	Tipo: Papelería		Tipo: Droguería
Cantidad bodega: 18		Cantidad bodega: 25	
Valor unitario: 550.0 \$		Valor unitario: 109.5 \$	
Cantidad vendidas: 0		Cantidad vendidas: 0	
Cantidad mínima: 5		Cantidad mínima: 8	
Abastecer	Vender	Cambiar	

Borrador		Pan	
	Tipo: Papelería		Tipo: Supermercado
Cantidad bodega: 30		Cantidad bodega: 15	
Valor unitario: 207.3 \$		Valor unitario: 150.0 \$	
Cantidad vendidas: 0		Cantidad vendidas: 0	
Cantidad mínima: 10		Cantidad mínima: 20	
Abastecer	Vender	Cambiar	

Opciones		
Producto más vendido	Producto menos vendido	Promedio ventas
Dinero en caja	Opción 1	Opción 2

- La **ventana** del programa tiene dos zonas: en la primera aparece la información de los productos de la tienda. Allí se tiene el nombre de cada producto, la cantidad disponible en la bodega de la tienda, el IVA que se debe pagar por el producto, su precio antes de impuestos y si ya se debe hacer o no un pedido.
- En esta zona también tenemos tres botones, cada uno asociado con un **requerimiento funcional**. Desde allí podemos vender el producto a un cliente, abastecer la tienda con el producto, o modificar la información del producto.
- Cuando el usuario selecciona las opciones Vender o Abastecer, la aplicación presenta un diálogo en el que el usuario puede indicar el número de unidades deseadas.
- Cuando el usuario selecciona la opción Cambiar, la aplicación presenta un diálogo en el que el usuario puede ingresar la nueva información del producto.
- En la última de las zonas se encuentran los botones que permiten pedir la información estadística correspondiente al último **requerimiento funcional**.

4. Nuevos Elementos de Modelado

4.1. Tipos Simples de Datos

En esta sección presentamos dos nuevos tipos simples de datos (`boolean` y `char`) y volvemos a estudiar algunos aspectos de los tipos introducidos en el capítulo anterior.

Comenzamos con el tipo `double` . Para facilitar el modelado de las características que toman valores reales, la mayoría de los lenguajes de programación proveen un tipo simple denominado `double` . En el caso de estudio de la tienda usaremos un [atributo](#) de este tipo para modelar el precio de cada producto. Esto nos va a permitir tener un producto cuyo precio sea, por ejemplo, \$23,12 (23 pesos y 12 centavos).

Se denomina [literal](#) de un [tipo de datos](#) a un valor [constante](#) de dicho tipo. En la siguiente tabla se dan algunos ejemplos de la manera de escribir literales para los tipos de datos estudiados. A medida que vayamos viendo nuevos tipos, iremos introduciendo la sintaxis que utilizan.

Tipo en Java	Ejemplo de literales	Comentarios
entero (<code>int</code>)	564, -12	Los literales de tipo entero se expresan como una secuencia de dígitos. Si el valor es negativo, dicha secuencia va precedida del símbolo "-".
real (<code>double</code>)	564.78, -98.3	Los literales de tipo real se expresan como una secuencia de dígitos. Para separar la parte entera de la parte decimal se utiliza el símbolo ".".
cadena de caracteres (<code>String</code>)	"esta es una cadena", " ", ""	Los literales de tipo cadena de caracteres van entre comillas dobles. Dos comillas dobles seguidas indican una cadena de caracteres vacía. Es distinta una cadena vacía que una cadena que sólo tiene un carácter de espacio en blanco.

En el ejemplo 1 se muestra la manera de declarar y manipular los atributos de tipo `double` usando el caso de estudio de la tienda. También se presenta la manera de convertir los valores reales a valores enteros.

Ejemplo 1

Objetivo: Repasar la manera de manejar atributos de tipo `double` en el [lenguaje de programación](#) Java, usando el caso de estudio de la tienda.

```
public class Producto
{
    private double valorUnitario;
}
```

- Declaración del **atributo** valorUnitario dentro de la **clase** Producto, para representar el precio del producto por unidad, antes de impuestos (sin IVA).
- Como de costumbre, el **atributo** lo declaramos privado, para evitar que sea manipulado desde fuera de la **clase**.

Las siguientes instrucciones pueden ir como parte de cualquier **método** de la **clase** Producto:

```
valorUnitario = 23.12;
```

- En cualquier **método** de la **clase** se puede asignar un **literal** de tipo real al **atributo**.

```
int valorPesos = ( int ) valorUnitario;
```

- Si en la **variable** valor valorPesos queremos tener la parte entera del precio del producto, utilizamos el **operador** de conversión (`int`). Este **operador** permite convertir valores reales a enteros.
- El **operador** (`int`) incluye los paréntesis y debe ir antes del valor que se quiere convertir.
- Si no se incluye el **operador** de conversión, el **compilador** va a señalar un error (*"Type mismatch: cannot convert from double to int"*).

```
valorUnitario = valorUnitario / 1.07;
```

Para construir una **expresión** aritmética de valor real, se pueden usar los operadores de suma(`+`), resta (`-`), multiplicación (`*`) y división (`/`).

```
int valorPesos = 17 / 3;
```

- La división entre valores enteros da un valor entero. En el caso del ejemplo, después de la **asignación**, la **variable** valorPesos tendrá el valor 5.
- El lenguaje Java decide en cada caso (dependiendo del tipo de los operandos) si utiliza la división entera o la división real para calcular el resultado.

Un **operador** que se utiliza frecuentemente en problemas aritméticos es el **operador** módulo (`%`). Este **operador** calcula el residuo de la división entre dos valores, y se puede utilizar tanto en expresiones enteras como reales. La siguiente tabla muestra el resultado de aplicar dicho **operador** en varias expresiones.

Expresión	Valor	Comentarios
<code>4%4</code>	0	El residuo de dividir 4 por 4 es cero. El resultado de este operador se puede ver como lo que "sobra" después de hacer la división entera.
<code>14%3</code>	2	El resultado de la expresión es 2, puesto que al dividir 14 por 3 se obtiene como valor entero 4 y "sobran" 2.
<code>17%3</code>	2	En esta expresión el valor entero es 5 (<code>5 * 3</code> es 15) y "sobran" de nuevo 2.
<code>3%17</code>	3	La división entera entre 3 y 17 es cero, así que "sobran" 3.
<code>4.5%2.2</code>	0.1	El operador <code>%</code> se puede aplicar también a valores reales. En la expresión del ejemplo, 2.2. está 2 veces en 4.5 y "sobra" 0.1.

Otro tipo simple de datos que encontramos en los lenguajes de programación es el que permite representar valores lógicos (verdadero o falso). El nombre de dicho tipo es `boolean` . Imagine, por ejemplo, que en la tienda queremos modelar una característica de un producto que dice si es subsidiado o no por el gobierno. De esta característica sólo nos interesaría saber si es verdadera o falsa (los únicos valores posibles), para saber si hay que aplicar o no el respectivo descuento. Este tipo de características se podría modelar usando un entero y una convención sobre la manera de interpretar su valor (por ejemplo, 1 es verdadero y 2 es falso). Es tan frecuente encontrar esta situación que muchos lenguajes resolvieron convertirlo en un nuevo **tipo de datos** y evitar así tener que usar otros tipos para representarlo.

El tipo `boolean` sólo tiene dos literales: **true** y **false**. Estos son los únicos valores constantes que se le pueden asignar a los atributos o variables de dicho tipo.

Ejemplo 2

Objetivo: Mostrar la manera de manejar atributos de tipo `boolean` en el **lenguaje de programación** Java.

En este ejemplo se utiliza una extensión del caso de estudio de la tienda, para mostrar la sintaxis de declaración y el uso de los atributos de tipo `boolean` .

```
public class Producto
{
    private boolean subsidiado;
}
```

- Aquí se muestra la declaración del **atributo** "subsidiado" dentro de la **clase** Producto.
- Dicha característica no forma parte del caso de estudio y únicamente se utiliza en este ejemplo para ilustrar el uso del **tipo de datos** `boolean`.

Las siguientes instrucciones pueden ir como parte de cualquier **método** de la **clase** Producto:

```
subsidiado = true;
subsidiado = false;
```

- Los únicos valores que se pueden asignar a los atributos de tipo `boolean` son `true` y `false`. Los operadores que nos permitirán crear expresiones con este tipo de valores, los veremos más adelante.

```
boolean valorLogico = subsidiado;
```

- Es posible tener variables de tipo `boolean`, a las cuales se les puede asignar cualquier valor de dicho tipo.

El último tipo simple de dato que veremos en este capítulo es el tipo `char`, que sirve para representar un carácter. En el ejemplo 3 se ilustra la manera de usarlo dentro del contexto del caso de estudio. Un valor de tipo `char` se representa internamente mediante un código numérico llamado UNICODE.

Ejemplo 3

Objetivo: Mostrar la manera de manejar atributos de tipo `char` en el **lenguaje de programación** Java, usando una extensión del caso de estudio de la tienda.

Suponga que los productos de la tienda están clasificados en tres grupos: A, B y C, según su calidad. En este ejemplo se muestra una manera de representar dicha característica usando un **atributo** de tipo `char`.

```
public class Producto
{
    private char calidad;
}
```

- Aquí se muestra la declaración del [atributo](#) "calidad" dentro de la [clase](#) Producto. Dicha característica será representada con un carácter que puede tomar como valores 'A', 'B' o 'C'.

Las siguientes instrucciones pueden ir como parte de cualquier [método](#) de la [clase](#) Producto:

```
calidad = 'A';  
calidad = 'B';
```

- Los literales de tipo `char` se expresan entre comillas sencillas. En eso se diferencian de los literales de la [clase](#) String, que van entre comillas dobles.

```
calidad = 67;
```

- Lo que aparece en este ejemplo es poco usual: es posible asignar directamente un código UNICODE a un [atributo](#) de tipo `char`. El valor 67, por ejemplo, es el código interno del carácter 'C'. El código interno del carácter 'c' (minúscula) es 99. Cada carácter tiene su propio código interno, incluso los que tienen tilde (el código del carácter 'á' es 225).

```
char valorCaracter = calidad;
```

- Es posible tener variables de tipo `char`, a las cuales se les puede asignar cualquier valor de dicho tipo.

4.2. Constantes para Representar Valores Inmutables

En muchos problemas encontramos algunos valores que no van a cambiar durante la ejecución del programa (inmutables). Considere el caso de la tienda, en el que el valor del precio final del producto depende de los impuestos definidos por la ley. Según lo que vimos en el nivel anterior, cada vez que necesitemos el valor del IVA de los productos de papelería, debemos escribir su valor numérico (0.16). Para facilitar la lectura y escritura del código, los lenguajes de programación permiten asociar un nombre significativo al valor, para así reemplazar el valor numérico dentro del código. Estos nombres asociados se denominan **constantes**.

Estas constantes pueden ser de cualquier [tipo de datos](#) (por ejemplo, puede haber una [constante](#) de tipo String o `double`) y se les debe fijar su valor desde la declaración. Dicho valor no puede ser modificado en ningún punto del programa.

El ejemplo 4 desarrolla esa idea con el caso de la tienda y muestra la sintaxis en Java para declarar y usar constantes.

Ejemplo 4

Objetivo: Mostrar el uso de constantes para representar los valores inmutables, usando el caso de estudio de la tienda.

En este ejemplo ilustramos el uso de constantes para representar los posibles valores del IVA de los productos.

```
public class Producto
{
    //-----
    // Constantes
    //-----

    private final static double IVA_PAPELERIA = 0.16;
    private final static double IVA_SUPERMERCADO = 0.04;
    private final static double IVA_FARMACIA = 0.12;
    ...
}
```

- Declaramos tres constantes que tienen los valores posibles del IVA en el problema: 16%, 12% y 4%. Estas constantes se llaman IVA_FARMACIA, IVA_PAPELERIA e IVA_SUPERMERCADO.
- Son constantes de tipo `double` , puesto que de ese tipo son los valores inmutables que queremos representar.
- Las constantes se declaran privadas si no van a ser usadas por fuera de la [clase](#).
- Para inicializar una [constante](#), se debe elegir un [literal](#) del mismo tipo de la [constante](#), o una [expresión](#).
- Dentro de la declaración de la [clase](#), se agrega una zona para declarar las constantes. Es conveniente situar esa zona antes de la declaración de los atributos.

Las siguientes instrucciones pueden ir como parte de cualquier [método](#) de la [clase](#) Producto:

```
precio = valorUnitario * ( 1 + IVA_SUPERMERCADO );
precio = valorUnitario * ( 1 + 0.04 );
```


- Las constantes sólo sirven para reemplazar el valor que representan. Las dos instrucciones del ejemplo son equivalentes y permiten calcular el precio al consumidor, aplicándole un IVA del 4% al precio de base del producto.
- La ventaja de las constantes es que cuando alguien lee el programa entiende a qué corresponde el valor 0.04 (puesto que también podría corresponder a los intereses o algún otro tipo de impuesto).

Esta práctica de definir constantes en sustitución de aquellos valores que no cambian durante la ejecución tiene muchas ventajas y es muy apreciada cuando hay necesidad de hacer el mantenimiento a un programa. Suponga, por ejemplo, que el gobierno autoriza un incremento en los impuestos y, ahora, el impuesto sobre los productos de supermercado pasa del 4% al 6%. Si dentro del programa siempre utilizamos la [constante](#) IVA_SUPERMERCADO para referirnos al valor del impuesto sobre los productos de supermercado, lo único que debemos hacer es reemplazar el valor 0.04 por 0.06 en la declaración de la [constante](#). Si por el contrario, en el código del programa no utilizamos el nombre de la [constante](#) sino el valor, tendríamos que ir a buscar todos los lugares en el código donde aparece el valor 0.04 (que hace referencia al impuesto sobre los productos de supermercado) y reemplazarlo por 0.06. Si hacemos lo anterior, fácilmente podemos pasar por alto algún lugar e introducir así un error en el programa.

Por convención, las constantes siempre van en mayúsculas. Si el nombre de la [constante](#) contiene varias palabras, es usual separarlas con el carácter "_". Por ejemplo podríamos tener una [constante](#) llamada PRECIO_MAXIMO.

Imaginémonos una nueva [constante](#) en la [clase](#) producto que define el precio máximo que puede tener un producto.

```
public class Producto
{
    //-----
    // Constantes
    //-----

    public final static double PRECIO_MAXIMO = 500000.0;
}
```

El siguiente [método](#) podría pertenecer a la [clase](#) Tienda:

```
public class Tienda
{
    public double ejemplo( )
    {
        return Producto.PRECIO_MAXIMO;
    }
}
```

- Por fuera de la **clase** Producto, las constantes pueden usarse indicando la **clase** en la cual fueron declaradas (siempre y cuando hayan sido declaradas como `public` en esa **clase**).

4.3. Enumeraciones para Definir el Dominio de un Atributo

Considere el caso de la tienda, en el que queremos modelar la característica de tipo de producto, el cual puede ser de tres tipos distintos: supermercado, papelería o droguería. En el nivel anterior vimos que es posible utilizar el tipo entero para representar esta característica, y asociar un número con cada uno de los valores posibles. Sin embargo, para estos casos, los lenguajes de programación permiten agrupar estos posibles valores de la característica, asignando solamente un nombre significativo para cada uno de ellos, sin asignarles ningún valor. Estas agrupaciones de valores de datos se denominan **enumeraciones**. De esta forma, dentro de los métodos podemos usar los nombres existentes en una enumeración.

El ejemplo 5 desarrolla esa idea con el caso de la tienda y muestra la sintaxis en Java para declarar y usar enumeraciones.

Ejemplo 5

Objetivo: Mostrar el uso de enumeraciones para representar los valores posibles de alguna característica.

Usando el caso de estudio de la tienda, en este ejemplo se muestra una manera de crear una enumeración para representar la característica de tipo de producto.

```
public class Producto
{
    //-----
    // Enumeraciones
    //-----
    public enum Tipo
    {
        PAPELERIA,
        SUPERMERCADO,
        FARMACIA
    }

    //-----
    // Atributos
    //-----
    private Tipo tipo;
    ...
}
```

- Se declara una enumeración llamada Tipo, para modelar el conjunto de nombres que podrán representar un tipo de producto.
- Dentro de la declaración del tipo, se agregan los nombres significativos de los tres tipos de producto existentes: PAPELERIA, SUPERMERCADO y FARMACIA.
- Se declara un [atributo](#) llamado "tipo" dentro de la [clase](#) Producto, para representar esa característica. El tipo asignado a este [atributo](#) es la enumeración que creamos arriba.
- Dentro de la declaración de la [clase](#), se agrega una zona para declarar las constantes. Es conveniente situar esa zona antes de la declaración de los atributos.

Para poder usar una enumeración, se debe escribir el nombre de la enumeración y después llamar el valor que se desea asignar. Las siguientes instrucciones pueden ir como parte de cualquier [método](#) de la [clase](#) Producto:

```
tipo = Tipo.PAPELERIA;
tipo = Tipo.SUPERMERCADO;
tipo = Tipo.FARMACIA;
```

Cualquiera de esas tres asignaciones define el tipo de un producto (no las tres a la vez, por supuesto). La ventaja de usar una enumeración (PAPELERIA) en lugar de un valor numérico es que el programa resultante es mucho más fácil de leer y entender.

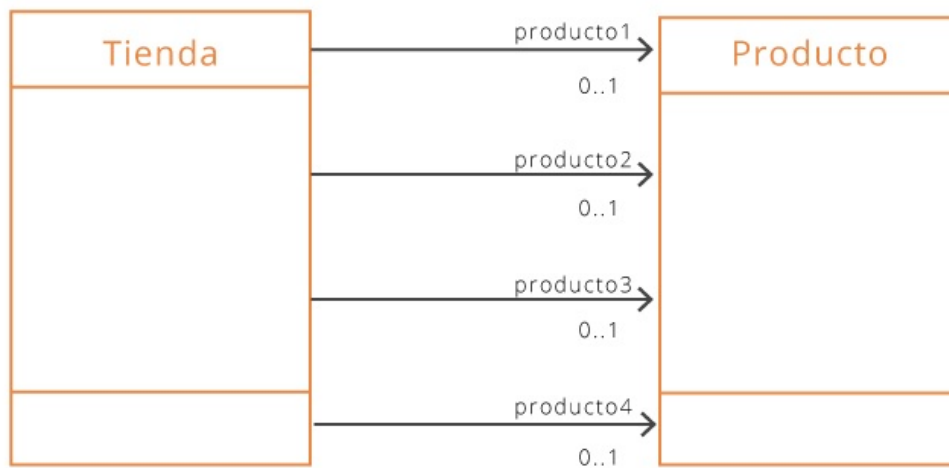
El siguiente [método](#) podría pertenecer a la [clase](#) Tienda:

```
public void ejemplo( )
{
    ...
    tipoVenta = Tipo.PAPELERIA;
    tipoCompra = Tipo.SUPERMERCADO;
    ...
}
```

- Por fuera de la [clase](#) Producto, las enumeraciones se llaman de la misma manera que se llamaba dentro de la [clase](#) donde fueron declaradas (siempre y cuando hayan sido declaradas como public en esa [clase](#)).
- En el ejemplo estamos suponiendo que `tipoVenta` y `tipoCompra` son atributos de la [clase](#) Tienda.

4.4. Manejo de Asociaciones Opcionales

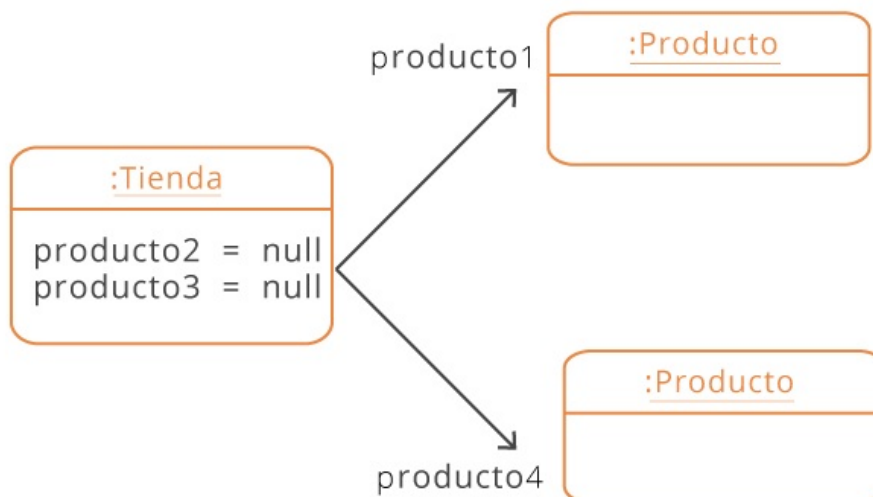
Supongamos que queremos modificar el enunciado del caso de la tienda, para que el programa pueda manejar 1, 2, 3 ó 4 productos. Lo primero que debemos hacer entonces es modificar el diagrama de clases, para indicar que las asociaciones pueden o no existir. Para esto usamos la sintaxis de UML que se ilustra en la [figura 2.2](#), y que dice que las asociaciones son opcionales. Esta característica se denomina **cardinalidad** de la [asociación](#) y se verá más a fondo en el nivel 3. Por ahora podemos decir que la cardinalidad define el número de instancias de una [clase](#) que pueden manejarse a través de una [asociación](#). En el caso de una [asociación](#) opcional, la cardinalidad es 0..1 (para expresar la cardinalidad, se usan dos números separados con dos puntos), puesto que a través de la [asociación](#) puede manejarse un [objeto](#) de la otra [clase](#) o ningún [objeto](#).

Fig. 2.2 Diagrama de clases con asociaciones opcionales

- La cardinalidad de la **asociación** llamada producto1 entre la **clase** Tienda y la **clase** Producto es cero o uno (0..1), para indicar que puede o no existir el **objeto** que representa la **asociación** producto1. Lo mismo sucede con cada una de las demás asociaciones.
- Si en el diagrama no aparece ninguna cardinalidad en una **asociación**, se interpreta como que ésta es 1 (existe exactamente un **objeto** de la otra **clase**).
- En la **figura 2.3** aparece un ejemplo de un **diagrama de objetos** para este diagrama de clases.

Dentro de un **método**, para indicar que el **objeto** correspondiente a una **asociación** que no está presente (que no hay, por ejemplo, un **objeto** de la **clase** Producto para la **asociación** producto1) se utiliza el valor especial `null` (`producto1 = null;`). En la **figura 2.3** se muestra un ejemplo de un **diagrama de objetos** para el modelo conceptual de la figura anterior.

Cuando se intenta llamar un **método** a través de una **asociación** cuyo valor es `null`, el computador muestra el error: *NullPointerException*.

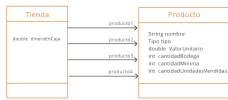
Fig. 2.3 Diagrama de objetos con asociaciones opcionales

El ejemplo anterior lo utilizamos únicamente para ilustrar la idea de una [asociación](#) opcional. En el resto del capítulo seguiremos trabajando con el caso inicial, en el cual todas las asociaciones entre la [clase](#) `Tienda` y la [clase](#) `Producto` tienen cardinalidad 1, tal como se muestra en el ejemplo 6.

Ejemplo 6

Objetivo: Mostrar las declaraciones de las clases `Tienda` y `Producto` que vamos a usar en el resto del capítulo.

En este ejemplo se muestra un [diseño](#) posible para las clases del caso de estudio de la tienda. Se presenta tanto el diagrama de clases en UML como las respectivas declaraciones en Java. En el [diseño](#) se incluyen los métodos de cada una de las clases.



El diagrama de clases consta de las clases Tienda y Producto, con 4 asociaciones entre ellos (todas de cardinalidad 1). Para cada **clase** se muestran los atributos que modelan las características importantes para el problema. Entre los principales atributos de la **clase** Producto están su nombre, su tipo, su valor unitario antes de impuestos, etc.

```

public class Tienda
{
    //-----
    // Atributos
    //-----
    private Producto producto1;
    private Producto producto2;
    private Producto producto3;
    private Producto producto4;
    private double dineroEnCaja;
    ...

    //-----
    //Métodos
    //-----
    public Producto darProducto1( ) { ... }
    public Producto darProducto2( ) { ... }
    public Producto darProducto3( ) { ... }
    public Producto darProducto4( ) { ... }
    public double darDineroEnCaja( ) { ... }
}
  
```

- Se modelan los 4 productos, unidos a la tienda con las asociaciones llamadas producto1, producto2, producto3 y producto4. Fíjese que las asociaciones y los atributos se declaran siguiendo la misma sintaxis. El dinero total que hay en caja de la tienda se modela con un **atributo** de tipo `double` .
- Esta es la lista de firmas de algunos de los métodos de la **clase** Tienda que utilizaremos en la siguiente sección. Esta lista se irá completando poco a poco, a medida que avancemos en el capítulo.

```
public class Producto
{
    //-----
    // Enumeraciones
    //-----
    /**
     * Enumeradores para los tipos de producto.
     */
    public enum Tipo
    {
        PAPELERIA,
        SUPERMERCADO,
        DROGUERIA
    }
    //-----
    // Constantes
    //-----
    private final static double IVA_PAPELERIA = 0.16;
    private final static double IVA_SUPERMERCADO = 0.04;
    private final static double IVA_DROGUERIA = 0.12;

    //-----
    // Atributos
    //-----
    private String nombre;
    private Tipo tipo;
    private double valorUnitario;
    private int cantidadBodega;
    private int cantidadMinima;
    private int cantidadUnidadesVendidas;

    //-----
    //Métodos
    //-----
    public String darNombre( ) { ... }
    public Tipo darTipo( ) { ... }
    public double darValorUnitario( ) { ... }
    public int darCantidadBodega( ) { ... }
    public int darCantidadMinima( ) { ... }
    public int darCantidadUnidadesVendidas( ) { ... }
}
```

- En la **clase** Producto, se declaran primero las constantes para representar los valores de modelado de los atributos. Luego, las constantes que representan valores inmutables.
- En la segunda zona va la declaración de los atributos de la **clase**.
- En la tercera zona se observa la lista de signaturas de algunos de los métodos de la **clase** Producto que utilizaremos en la siguiente sección. Esta lista se irá completando poco a poco, a medida que avancemos en el capítulo.

5. Expresiones

5.1. Algunas Definiciones

Una **expresión** es la manera en que expresamos en un **lenguaje de programación** algo sobre el estado de un **objeto**. Es el medio que tenemos para decir en un programa algo sobre el mundo del problema. En el nivel anterior vimos las expresiones aritméticas, que permitían definir la manera en que debía ser modificado el estado de un elemento del mundo, usando sumas y restas.

Las expresiones aparecen dentro del cuerpo de los métodos y están formadas por **operandos** y **operadores**. Los operandos pueden ser atributos, parámetros, literales, constantes o llamadas de métodos, mientras que los operadores son los que indican la manera de calcular el valor de la **expresión**. Los operadores que se pueden utilizar en una **expresión** dependen del tipo de los datos de los operandos que allí aparezcan.

En algunos casos es indispensable utilizar paréntesis para evitar la ambigüedad en las expresiones. Por ejemplo, la **expresión** $10 - 4 - 2$ puede ser interpretada de dos maneras, cada una con un resultado distinto: $10 - (4 - 2) = 8$, o también $(10 - 4) - 2 = 4$. Es buena idea usar siempre paréntesis en las expresiones, para estar seguros de que la interpretación del computador es la que nosotros necesitamos.

Ejemplo 7

Objetivo: Ilustrar la manera de usar expresiones aritméticas para hablar del estado de un **objeto**.

Suponga que estamos en un **objeto** de la **clase** Producto. Vamos a escribir e interpretar algunas expresiones aritméticas simples.

La expresión ...	Se interpreta como...
<code>valorUnitario * 2</code>	El doble del valor unitario del producto.
<code>cantidadBodega - cantidadMinima</code>	La cantidad del producto que hay que vender antes de poder hacer un pedido.
<code>valorUnitario * (1 + (IVA_PAPELERIA / 2))</code>	El precio final al consumidor si el producto debe pagar el IVA de los productos de papelería (16%) y sólo paga la mitad de éste.
<code>cantidadUnidadesVendidas * 1.1</code>	La cantidad de unidades vendidas del producto, inflado en un 10%.

5.2. Operadores Relacionales

Los lenguajes de programación cuentan siempre con operadores relacionales, los cuales permiten determinar un valor de verdad (verdadero o falso) para una situación del mundo. Si queremos determinar, por ejemplo, si el valor unitario antes de impuestos de un producto es menor que \$10.000, podemos utilizar (dentro de la [clase](#) Producto) la [expresión](#):

```
valorUnitario < 10000
```

Los operadores relacionales son seis, que se resumen en la siguiente tabla:

Significado	Símbolo	Ejemplo
Es igual que	<code>==</code>	<code>valorUnitario == 55.75</code>
Es diferente de	<code>!=</code>	<code>tipo != Tipo.PAPELERIA</code>
Es menor que	<code><</code>	<code>cantidadBodega < 120</code>
Es mayor que	<code>></code>	<code>cantidadBodega > cantidadMinima</code>
Es menor o igual que	<code><=</code>	<code>valorUnitario <= 100.0</code>
Es mayor o igual que	<code>>=</code>	<code>valorUnitario >= 100.0</code>

Ejemplo 8

Objetivo: Ilustrar la manera de usar operadores relacionales para describir situaciones de un [objeto](#) (algo que es verdadero o falso).

Suponga que estamos en un [objeto](#) de la [clase](#) Producto. Vamos a escribir e interpretar algunas expresiones que usan operadores relacionales.

La expresión ...	Se interpreta como..
<code>tipo == Tipo.DROGUERIA</code>	¿El producto es de droguería?
<code>cantidadBodega > 0</code>	¿Hay disponibilidad del producto en la bodega?
<code>totalProductosVendidos > 0</code>	¿Se ha vendido alguna unidad del producto?
<code>cantidadBodega <= cantidadMinima</code>	¿Ya es posible hacer un nuevo pedido del producto?

5.3. Operadores Lógicos

Los operadores lógicos nos permiten describir situaciones más complejas, a partir de la composición de varias expresiones relacionales o de atributos de tipo `boolean`. Los operadores lógicos son tres: `&&` (y), `||` (o), `!` (no), y el resultado de aplicarlos se

resume de la siguiente manera:

- `operando1 && operando2` es cierto, si ambos operandos son verdaderos.
- `operando1 || operando2` es cierto, si cualquiera de los dos operandos es verdadero.
- `!operando` es cierto, si el `operando` es falso.

Los operadores `&&` y `||` se comportan de manera un poco diferente a todos los demás. La **expresión** en la que estén sólo se evalúa de izquierda a derecha hasta que se establezca si es verdadera o falsa. El computador no pierde tiempo evaluando el resto de la **expresión** si ya sabe cual será su resultado.

Ejemplo 9

Objetivo: Ilustrar la manera de usar operadores lógicos para describir situaciones de un **objeto** (algo que es cierto o falso).

Suponga que estamos en un **objeto** de la **clase** `Producto`. Vamos a escribir e interpretar algunas expresiones que usan operadores lógicos. $x = y$

La expresión ...	Se interpreta como...
<code>tipo == Tipo.SUPERMERCADO && cantidadUnidadesVendidas == 0</code>	¿El producto es de supermercado y no se ha vendido ninguna unidad? En este caso, si el producto no es de supermercado o ya se ha vendido alguna unidad, la expresión es falsa.
<code>valorUnitario >= 10000 && valorUnitario <= 20000 && tipo == Tipo.DROGUERIA</code>	¿El producto vale entre \$10.000 y \$20.000 y, además, es un producto de droguería?
<code>!(tipo == Tipo.PAPELERIA)</code>	¿El producto no es de papelería? Note que esta expresión es equivalente a la expresión que va en la siguiente línea. Y también es equivalente a <code>(tipo != Tipo.PAPELERIA)</code> .
<code>tipo == Tipo.SUPERMERCADO tipo == Tipo.DROGUERIA</code>	¿El producto es de supermercado o de droguería?

Operadores sobre Cadenas de Caracteres

El tipo `String` nos sirve para representar cadenas de caracteres. A diferencia de los demás tipos de datos vistos hasta ahora, este tipo no es simple, sino que se implementa mediante una **clase** especial en Java. Esto implica que, en algunos casos, para invocar sus operaciones debemos utilizar la sintaxis de llamada de métodos.

Existen muchas operaciones sobre cadenas de caracteres, pero en este nivel sólo nos vamos a interesar en el **operador** de concatenación (`+`), en el de comparación (`equals`) y en el de extracción de un carácter (`charAt`).

El primer **operador** (`+`) sirve para pegar dos cadenas de caracteres, una después de la otra. Por ejemplo, si quisiéramos tener un **método** en la **clase** `Producto` que calculara el mensaje que se debe mostrar en la publicidad de la tienda, tendría la siguiente forma:

```
public String darPublicidad( )
{
    return "Compre el producto " + nombre + " por solo $" + valorUnitario;
}
```

- Si alguno de los operandos no es una cadena de caracteres (como es el caso del **atributo** de tipo real `valorUnitario`) el **compilador** se encarga de convertirlo a cadena. No es necesario hacer una conversión explícita porque el **compilador** lo hace automáticamente por nosotros, para todos los tipos simples de datos.
- Al ejecutar este **método**, retornará una cadena con algo del siguiente estilo: Compre el producto cuaderno por solo \$100.50.

La segunda operación que nos interesa en este momento es la comparación de cadenas de caracteres. A diferencia de los tipos simples, en donde se utiliza el **operador** `==` , para poder comparar dos cadenas de caracteres es necesario llamar el **método** `equals` de la **clase** `String`. Por ejemplo, si queremos tener un **método** en la **clase** `Producto` que reciba como **parámetro** una cadena de caracteres e informe si el nombre del producto es igual al valor recibido como **parámetro**, éste sería más o menos así:

```
public boolean esIgual( String pBuscado )
{
    return nombre.equals( pBuscado );
}
```

- Se usa la sintaxis de invocación de métodos para poder utilizar el **método** `equals`. La razón es que `String` es una **clase**, y se deben respetar las reglas de llamada de un **método** (`int` , `double` y `boolean` no son clases, y por esta razón se puede utilizar el **operador** `==` directamente).
- El retorno del **método** `equals` es de **tipo boolean**, razón por la cual lo podemos retornar directamente como respuesta del **método** que queremos construir.
- En el ejemplo, el **método** `equals` se invoca sobre el **atributo** de la **clase** `Producto` llamado "nombre" y se le pasa como **parámetro** el valor recibido en "buscado".

La última operación que vamos a estudiar en este nivel nos permite "obtener" un carácter de una cadena. Para esto debemos dar la posición dentro de la cadena del carácter que nos interesa, e invocar el [método](#) `charAt` de la [clase](#) `String`, tal como se muestra en los siguientes ejemplos. Nótese que el primer carácter de una cadena se encuentra en la posición 0.

Suponga que tenemos dos cadenas de caracteres, declaradas de la siguiente manera:

```
String cad1 = "la casa es roja";
String cad2 = "La Casa es Roja";
```

La expresión ...	Tiene el valor..	Comentarios...
<code>cad1.equals(cad2)</code>	false	La expresión es falsa, porque la comparación se hace teniendo en cuenta las mayúsculas y las minúsculas.
<code>cad1.equalsIgnoreCase(cad2)</code>	true	Con este método de la clase <code>String</code> podemos comparar dos cadenas de caracteres, ignorando si son mayúsculas o minúsculas.
<code>cad1 + " y verde"</code>	"la casa es roja y verde"	Se debe prever un espacio en blanco entre las cadenas, si no queremos que queden pegadas.
<code>cad1.charAt(1)</code>	'a'	Los caracteres de la cadena se comienzan a numerar desde cero.
<code>cad2.charAt(2)</code>	' '	El espacio en blanco es el tercer carácter de la cadena. Debe quedar claro que no es lo mismo el carácter ' ' que la cadena de caracteres " ". El primero es un literal de tipo <code>char</code> , mientras que el segundo es un literal de la clase <code>String</code> .

Si en una [expresión](#) aritmética no se usan paréntesis para definir el orden de evaluación, Java aplicará a los operadores un orden por defecto. Dicho orden está asociado con una prioridad que el lenguaje le asigna a cada [operador](#).

Básicamente, las reglas se pueden resumir de la siguiente manera:

- Primero se aplican los operadores de multiplicación y división, de izquierda a derecha.
- Después se aplican los operadores de suma y resta, de izquierda a derecha.

Supongamos que tenemos dos variables `var1` y `var2`, con valores 10 y 5 respectivamente.

La expresión...	Tiene el valor...	Comentarios...
$\text{var1} - \text{var2} - 10$	-5	Aplica el operador de resta de izquierda a derecha.
$\text{var1} - (\text{var2} - 10)$	15	Los paréntesis le dan un orden de evaluación distinta a la expresión : $10 - (5 - 10) = 10 - (-5) = 10 + 5 = 15$.
$\text{var1} * \text{var2} / 5$	10	En esta expresión se hace primero la multiplicación y luego la división: $(10 * 5) / 5 = 50 / 5 = 10$. Esto es así porque ambos operadores tienen la misma prioridad, de modo que se evalúan de izquierda a derecha.
$\text{var1} * (\text{var2} / 10)$	5	Los paréntesis le dan un orden de evaluación distinto a la expresión : $10 (5 / 10) = 10 0.5 = 5$.
$\text{var1} - \text{var2} + 10$	15	En esta expresión se hace primero la resta y después la suma (aplica los operadores suma y resta de izquierda a derecha, puesto que ambos tienen la misma prioridad).
$\text{var1} + \text{var2} * 10$	60	En esta expresión se hace primero la multiplicación, puesto que ese operador tiene más prioridad que la suma.
$\text{var1} + \text{var2} * 10 - 5$	55	En esta expresión se hace primero la multiplicación, luego la suma y, finalmente, la resta.
$\text{var1} + \text{var2} * 10 / 5$	20	En esta expresión se hace primero la multiplicación, luego la división y, finalmente, la suma. Debe ser clara, en este punto, la importancia de los paréntesis en las expresiones.

Llegó el momento de comenzar a trabajar en el caso de la tienda, así que de nuevo manos a la obra.

Tarea 2

Objetivo: Generar habilidad en la construcción e interpretación de expresiones, utilizando el caso de estudio de la tienda.

Utilizando las declaraciones hechas en la sección anterior para las clases Tienda y Producto y el escenario propuesto a continuación, resuelva los ejercicios que se plantean más adelante.

Escenario:

Suponga que en la tienda del caso de estudio se tienen a la venta los siguientes productos:

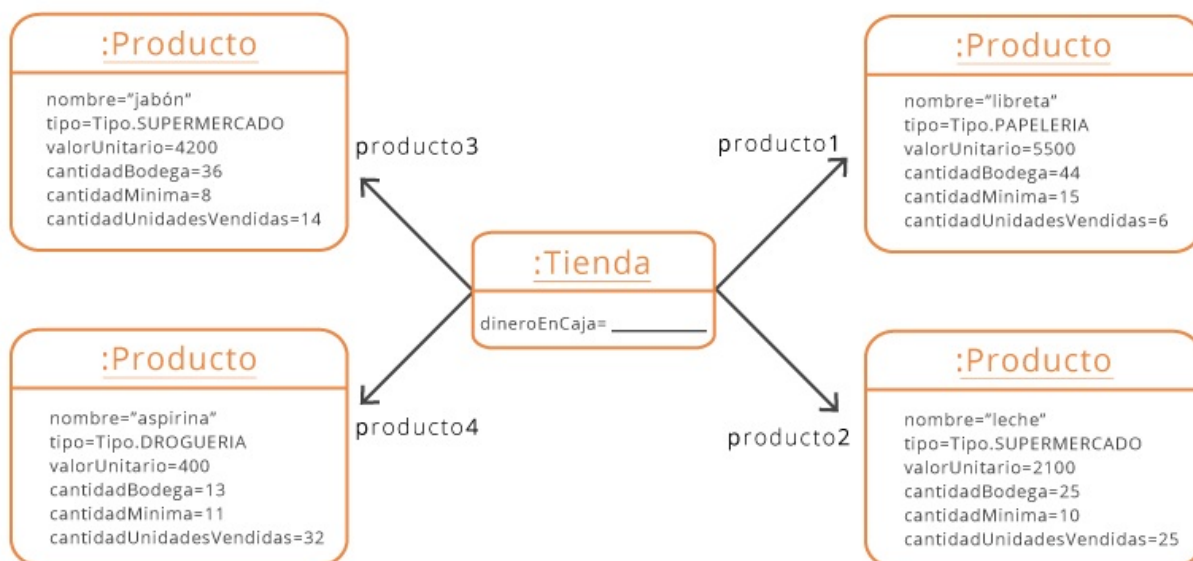
1. Libreta de apuntes, producto de papelería, a \$5.500 pesos la unidad.
2. Leche en bolsa de 1 litro, producto de supermercado, a \$2.100 pesos.
3. Jabón en polvo, producto de supermercado, a \$4.200 el kilo.
4. Aspirina, producto de droguería, a \$400 la unidad.

Suponga además, que ya se han vendido en la tienda 6 libretas, 25 bolsas de leche, 14 bolsas de jabón y 32 aspirinas.

Por último tenemos la siguiente tabla para resumir el inventario de unidades de la tienda y la cantidad por debajo de la cual se puede hacer un abastecimiento.

Producto	Cantidad en bodega	Cantidad mínima
libreta	44	15
leche	25	10
jabón	36	8
aspirina	13	11

En el siguiente [diagrama de objetos](#) puede ver el estado actual de la tienda. Complete la cantidad de dinero en caja que tiene la tienda, teniendo en cuenta las ventas que ya se realizaron.



Parte I – Evaluación de Expresiones (operadores aritméticos):

Para el objeto...	la expresión...	toma el valor...
leche	<code>cantidadBodega - cantidadMinima</code>	15
aspirina	<code>valorUnitario * cantidadBodega</code>	<input type="text"/>
jabón	<code>(cantidadUnidadesVendidas + cantidadBodega) * (valorUnitario + valorUnitario * IVA_SUPERMERCADO)</code>	<input type="text"/>
libreta	<code>valorUnitario * cantidadBodega / cantidadUnidadesVendidas * valorUnitario</code>	<input type="text"/>
leche	<code>valorUnitario * cantidadUnidadesVendidas * IVA_SUPERMERCADO</code>	<input type="text"/>
aspirina	<code>valorUnitario * (1 + IVA_DROGUERIA) * cantidadUnidadesVendidas</code>	<input type="text"/>
la tienda	<code>(producto1.darValorUnitario() + producto2.darValorUnitario() + producto3.darValorUnitario() + producto4.darValorUnitario()) / 4</code>	3050.0
la tienda	<code>(producto1.darCantidadBodega() - producto1.darCantidadMinima()) * (producto1.darValorUnitario() * (1 + producto1.darIVA()))</code>	<input type="text"/>
la tienda	<code>dineroEnCaja - (producto2.darCantidadMinima() * producto2.darValorUnitario())</code>	<input type="text"/>
la tienda	<code>producto3.darCantidadUnidadesVendidas() * (1 + producto3.darIVA())</code>	<input type="text"/>

Parte II – Evaluación de Expresiones (operadores relacionales):

Para el objeto...	la expresión...	toma el valor...
libreta	<code>tipo == Tipo.PAPELERIA</code>	true
libreta	<code>tipo != Tipo.DROGUERIA</code>	<input type="text"/>
leche	<code>cantidadMinima >= cantidadBodega</code>	<input type="text"/>
jabón	<code>valorUnitario <= 10000</code>	<input type="text"/>
aspirina	<code>cantidadBodega - cantidadMinima != cantidadUnidadesVendidas</code>	<input type="text"/>
jabón	<code>cantidadBodega * valorUnitario == cantidadUnidadesVendidas * IVA_PAPELERIA</code>	<input type="text"/>
la tienda	<code>producto1.darCantidadUnidadesVendidas() + producto2.darCantidadUnidadesVendidas() > producto3.darCantidadUnidadesVendidas()</code>	true
la tienda	<code>dineroEnCaja <= producto4.darCantidadUnidadesVendidas() * ((1 + producto4.darIVA()) * producto4.darValorUnitario())</code>	<input type="text"/>
la tienda	<code>(producto1.darCantidadBodega() + producto2.darCantidadBodega() + producto3.darCantidadBodega() + producto4.darCantidadBodega()) <= 1000</code>	<input type="text"/>
la tienda	<code>dineroEnCaja * producto1.darIVA() > producto1.darCantidadUnidadesVendidas() * producto1.darValorUnitario()</code>	<input type="text"/>

Parte III – Evaluación de Expresiones (operadores lógicos):

Para el objeto...	la expresión...	toma el valor...
leche	<code>!(tipo == Tipo.PAPELERIA tipo == Tipo.DROGUERIA)</code>	true
jabón	<code>tipo == Tipo.SUPERMERCADO && valorUnitario <= 10000</code>	<input type="text"/>
aspirina	<code>cantidadBodega > cantidadMinima && cantidadBodega < cantidadUnidadesVendidas</code>	<input type="text"/>
libreta	<code>valorUnitario >= 1000 && valorUnitario <= 5000</code>	<input type="text"/>
leche	<code>tipo != Tipo.PAPELERIA && tipo != Tipo.SUPERMERCADO</code>	<input type="text"/>
aspirina	<code>tipo == Tipo.PAPELERIA && valorUnitario > 50 && !(cantidadMinima < cantidadBodega)</code>	<input type="text"/>
la tienda	<code>producto1.darTipo() == Tipo.PAPELERIA && producto2.darTipo() == Tipo.SUPERMERCADO && producto3.darTipo() != Tipo.DROGUERIA && producto4.darTipo() == Tipo.SUPERMERCADO</code>	false
la tienda	<code>(dineroEnCaja / producto1.darValorUnitario()) >= producto1.darCantidadMinima()</code>	<input type="text"/>
la tienda	<code>((producto2.darCantidadBodega() + producto2.darCantidadBodega())/10 < 100) && ((producto2.darCantidadBodega()+producto2.darCantidadBodega())/10 >= 50)</code>	<input type="text"/>
la tienda	<code>dineroEnCaja * 0.1 <= producto3.darValorUnitario() * (1 + producto3.darIVA())</code>	<input type="text"/>

Parte IV – Creación de Expresiones (operadores aritméticos):

En un método de la clase...	para obtener..	se usa la expresión...
Producto	Valor de venta de un producto con IVA del 16%	<code>valorUnitario * (1 + IVA_PAPELERIA)</code>
Producto	Número de unidades que se deben vender para alcanzar la cantidad mínima.	
Producto	Número de veces que se ha vendido la cantidad mínima del producto.	
Producto	Número de unidades sobrantes si se arman paquetes de 10 con lo disponible en bodega.	
Tienda	Dinero en caja de la tienda incrementado en un 25%	<code>dineroEnCaja * 1.25</code>
Tienda	Total del IVA a pagar por las unidades vendidas de todos los productos.	
Tienda	El número de unidades del producto 3 que se pueden pagar (a su valor unitario) con el dinero en caja de la tienda.	
Tienda	El número de estantes de 50 posiciones que se requieren para almacenar las unidades en bodega de todos los productos (suponga que cada unidad de producto ocupa una posición).	

Parte V – Creación de Expresiones (operadores relacionales):

En un método de la clase ...	para obtener..	se usa la expresión ...
Producto	¿La cantidad en bodega es mayor o igual al doble de la cantidad mínima?	<code>cantidadBodega >= 2 * cantidadMinima</code>
Producto	¿El tipo no es PAPELERIA?	
Producto	¿El total de productos vendidos es igual a la cantidad en bodega?	
Producto	¿El nombre del producto comienza por el carácter 'a'?	
Tienda	¿El nombre del producto 2 es "aspirina"?	<code>producto2.darNombre().equals("aspirina")</code>
Tienda	¿La cantidad mínima del producto 4 es una quinta parte de la cantidad de productos vendidos?	
Tienda	¿El valor obtenido por los productos vendidos (incluyendo el IVA) es menor a un tercio del dinero en caja?	
Tienda	¿El promedio de unidades vendidas de todos los productos es mayor al promedio de unidades en bodega de todos los productos?	

Parte VI – Creación de Expresiones (operadores lógicos):

En un método de la clase ...	para obtener..	se usa la expresión ...

Producto	¿El tipo de producto es SUPERMERCADO y su valor unitario es menor a \$3.000?	<code>tipo == Tipo.SUPERMERCADO && valorUnitario < 3000</code>
Producto	¿En la cantidad en bodega o en la cantidad de productos vendidos está al menos 2 veces la cantidad mínima?	
Producto	¿El tipo no es DROGUERIA y el valor está entre 1000 y 3500 incluyendo ambos valores?	
Producto	¿El tipo es PAPELERIA y la cantidad en bodega es mayor a 10 y el valor unitario es mayor o igual a \$3.000?	
Tienda	¿El tipo del producto 1 no es ni DROGUERIA ni PAPELERIA y el total de unidades vendidas de todos los productos es menor a 30?	<code>producto1.darTipo() != Tipo.DROGUERIA && producto1.darTipo() != Tipo.PAPELERIA && (producto1.darProductosVendidos() + producto2.darProductosVendidos() + producto3.darProductosVendidos() + producto4.darProductosVendidos()) < 30</code>
Tienda	¿Con el valor en caja de la tienda se pueden pagar 500 unidades del producto 1 ó 300 unidades del producto 3 (al precio de su valor unitario)?	
Tienda	¿Del producto 4, el tope mínimo es mayor a 10 y la cantidad en bodega es menor o igual a 25?	

Tienda	¿El valor unitario de los productos 1 y 2 está entre 200 y 1000 sin incluir dichos valores?	
--------	---	--

5.5. Manejo de Variables

El objetivo de las **variables** es permitir manejar cálculos parciales en el interior de un **método**. Las variables se deben declarar (darles un nombre y un tipo) antes de ser utilizadas y siguen la misma convención de nombres de los atributos. Las variables se crean en el momento en el que se declaran y se destruyen automáticamente al llegar al final del **método** que las contiene. Por esta razón es imposible utilizar el valor de una **variable** por fuera del **método** donde fue declarada.

Se suelen usar variables por tres razones principales:

1. Porque es necesario calcular valores intermedios.
2. Por eficiencia, para no pedir dos veces el mismo servicio al mismo **objeto**.
3. Por claridad en el código.

A continuación se muestra un ejemplo de un **método** de la **clase** Tienda que calcula la cantidad disponible del primer producto y luego vende esa misma cantidad de todos los demás.

```
public void venderDeTodo( )
{
    int cuanto = producto1.darCantidadBodega( );
    producto2.vender( cuanto );
    producto3.vender( cuanto );
    producto4.vender( cuanto );
}
```

- Se declara al comienzo del **método** una **variable** de tipo entero llamada " `cuanto` ", y se le asigna la cantidad que hay en bodega del producto 1 de la tienda.
- La declaración de la **variable** y su inicialización se pueden hacer en instrucciones separadas (no hay necesidad de inicializar las variables en el momento de declararlas). La única **condición** que verifica el **compilador** es que antes de usar una **variable** ya haya sido inicializada.
- En este **método** se usa la **variable** " `cuanto` " por eficiencia y por claridad (no calculamos el mismo valor tres veces sino sólo una).

5.6 Otros Operadores de Asignación

El **operador** de **asignación** visto en el nivel anterior permite cambiar el valor de un **atributo** de un **objeto**, como una manera de reflejar un cambio en el mundo del problema. Vender 5 unidades de un producto, por ejemplo, se hace restando el valor 5 del **atributo**

```
cantidadBodega .
```

En este nivel vamos a introducir cuatro nuevos operadores de **asignación**, con la aclaración de que sólo es una manera más corta de escribir las asignaciones, las cuales siempre se pueden escribir con el **operador** del nivel anterior.

- **Operador** `++` . Se aplica a un **atributo** entero, para incrementarlo en 1. Por ejemplo, para indicar que se agregó una unidad de un producto a la bodega (en la **clase** **Producto**), se puede utilizar cualquiera de las siguientes versiones del mismo **método**.

```
public void agregarNuevaUnidadBodega( )
{
    cantidadBodega++;
}
```

- El **operador** de incremento se puede ver como un **operador** de **asignación** en el cual se modifica el valor del **operando** sumándole el valor 1.
- El uso de este **operador** tiene la ventaja de generar expresiones un poco más compactas.

```
public void agregarNuevaUnidadBodega( )
{
    cantidadBodega = cantidadBodega + 1;
}
```

- Esta segunda versión del **método** tiene la misma funcionalidad, pero utiliza el **operador** de **asignación** normal.
- **Operador** `--` . Se aplica a un **atributo** entero, para disminuirlo en 1. Se utiliza de manera análoga al **operador** de incremento.
- **Operador** `+=` . Se utiliza para incrementar un **atributo** en cualquier valor. Por ejemplo, el **método** para hacer un pedido de una cierta cantidad de unidades para la bodega, puede escribirse de las dos maneras que se muestran a continuación. Debe quedar claro que la instrucción `var++` es equivalente a `var += 1` , y equivalente a su vez a `var = var + 1` .


```
public void pedir( int pNum )
{
    cantidadBodega += pNum;
}
```

- Este **método** de la **clase** Producto permite hacer un pedido de "pNum" unidades y agregarlas a la bodega.
- El **operador** `+=` se puede ver como una generalización del **operador** `++`, en el cual el incremento puede ser de cualquier valor y no sólo igual a 1.

```
public void pedir( int pNum )
{
    cantidadBodega = cantidadBodega + pNum;
}
```

- Esta segunda versión del **método** tiene la misma funcionalidad, pero utiliza el **operador** de **asignación** normal.
- La única ventaja de utilizar el **operador** `+=` es que se obtiene un código un poco más compacto. Usarlo o no usarlo es cuestión de estilo de cada programador.
- **Operador** `-=`. Se utiliza para disminuir un **atributo** en cualquier valor. Se utiliza de manera análoga al **operador** `+=`.

Tarea 3

Objetivo: Generar habilidad en la utilización de las asignaciones y las expresiones como un medio para transformar el estado de un **objeto**.

Para las declaraciones de las clases Tienda y Producto dadas anteriormente y, teniendo en cuenta el escenario planteado más adelante, escriba la instrucción o las instrucciones necesarias para modificar el estado, siguiendo la descripción que se hace en cada caso.

Escenario

Suponga que en la tienda del caso de estudio se tienen a la venta los siguientes productos:

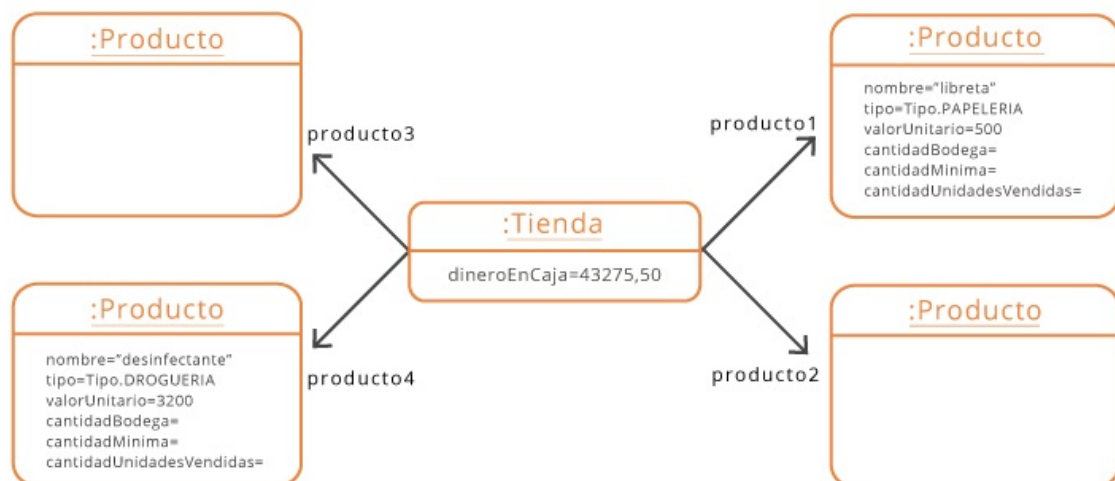
1. Lápiz, producto de papelería, con un valor base de \$500 pesos la unidad.
2. Borrador, producto de papelería, a \$300 pesos.
3. Kilo de café, producto de supermercado, a \$5.600 la unidad.
4. Desinfectante, producto de droguería, a \$3.200 la unidad.

Suponga además, que se han vendido 15 lápices, 5 borradores, 7 kilos de café y 12 frascos de desinfectante, y que en la caja de la tienda hay en este momento \$43.275,50.

Por último tenemos la siguiente tabla para resumir el inventario de unidades de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido:

Producto	Cantidad en bodega	Tope mínimo
lapiz	30	9
borrador	15	5
café	20	10
desinfectante	12	11

Complete el [diagrama de objetos](#) que aparece a continuación, con la información del escenario:



[Signatura](#) de los métodos de la [clase](#) Tienda:

```
//-----  
// Signaturas de métodos  
//-----  
public Producto darProducto1( )  
public Producto darProducto2( )  
public Producto darProducto3( )  
public Producto darProducto4( )  
public double darDineroEnCaja( )
```

Signatura de los métodos de la **clase** Producto:

```
//-----  
// Signaturas de métodos  
//-----  
public String darNombre( )  
public int darTipo( )  
public double darValorUnitario( )  
public int darCantidadBodega( )  
public int darCantidadMinima( )  
public int darCantidadUnidadesVendidas( )  
public double darIVA( )  
public int vender( int pCantidad )  
public void abastecer( int pCantidad )
```

En un método de la clase ...	la siguiente modificación de estado..	se logra con las siguientes instrucciones...
Producto	Se vendieron 5 unidades del producto (suponga que hay suficientes).	<code>cantidadUnidadesVendidas += 5; cantidadBodega -= 5;</code>
Producto	El valor unitario se incrementa en un 10%	
Producto	Se incrementa en uno la cantidad mínima para hacer pedidos.	
Producto	El producto ahora se clasifica como de SUPERMERCADO	
Producto	Se cambia el nombre del producto. Ahora se llama "teléfono".	

En un método de la clase ...	la siguiente modificación de estado..	se logra con las siguientes instrucciones...
Tienda	Se asigna al dinero en caja de la tienda la suma de los valores unitarios de los cuatro productos.	<code>dineroEnCaja = producto1.darValorUnitario() + producto2.darValorUnitario() + producto3.darValorUnitario() + producto4.darValorUnitario());</code>

Tienda	Se venden 4 unidades del producto 3 (suponga que están disponibles).	
Tienda	Se disminuye en un 2% el dinero en la caja.	
Tienda	Se abastece la tienda con la mitad de la cantidad mínima de cada producto, suponiendo que la cantidad en bodega de todos los productos es menor a la cantidad mínima.	
Tienda	Se pone en la caja el dinero correspondiente a las unidades vendidas de todos los productos de la tienda.	
Una clase de la interfaz de usuario	Se vende una unidad de cada uno de los productos de la tienda. Recuerde que este método está por fuera de la clase Tienda, y que por lo tanto no puede utilizar sus atributos de manera directa.	

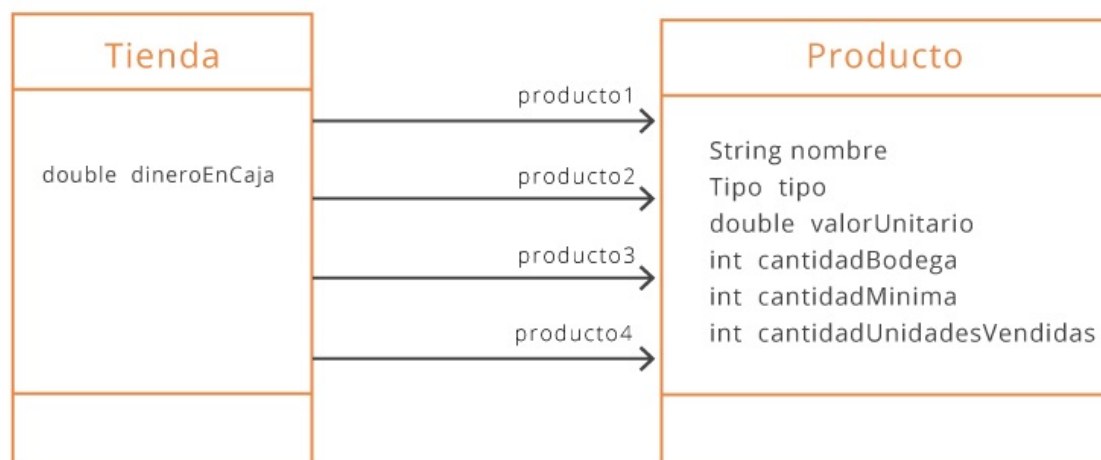
Antes de comenzar a escribir el cuerpo de un **método**, es importante tener en cuenta la **clase** en la cual éste se encuentra. No olvide que dependiendo de la **clase** en la que uno se encuentre, las cosas se deben decir de una manera diferente. En unos casos los atributos se pueden manipular directamente y, en otros, es indispensable llamar un **método** para cambiar el estado (para que la modificación la realice el **objeto** al que pertenece el **atributo**).

6. Clases y Objetos

6.1. Diferencia entre Clases y Objetos

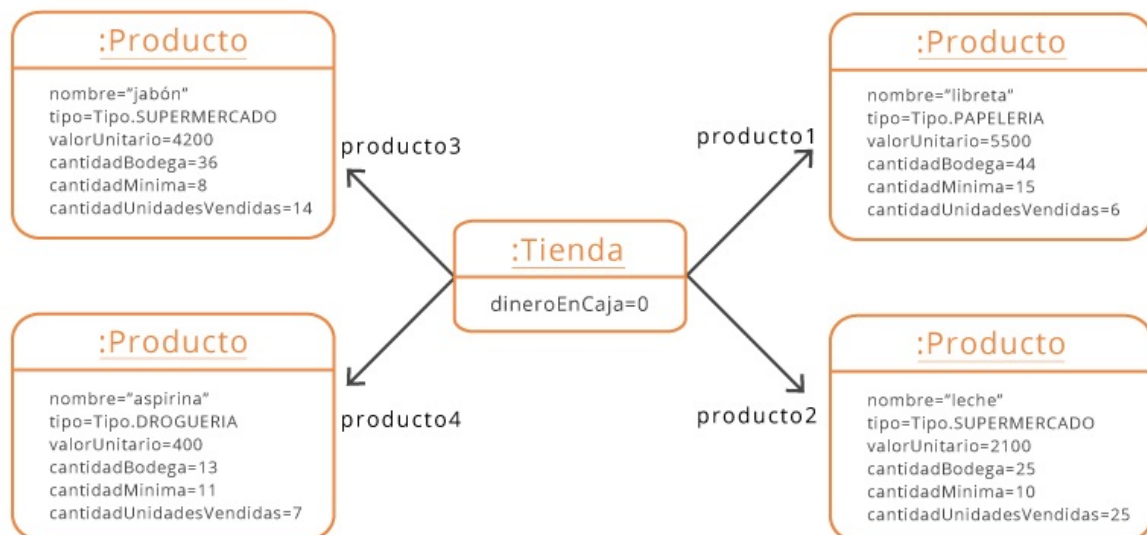
Aunque los conceptos de [clase](#) y objetos son muy diferentes, el hecho de usarlos indistintamente en algunos contextos hace que se pueda generar alguna confusión al respecto. En la [figura 2.4a](#) y [figura 2.4b](#) se muestra, para el caso de la tienda, el correspondiente diagrama de clases y un ejemplo de un posible [diagrama de objetos](#). Allí se puede apreciar que la [clase](#) Tienda describe todas las tiendas imaginables que vendan 4 productos.

Fig. 2.4a Modelo de clases



- Diagrama de clases para el caso de estudio de la tienda.
- El diagrama sólo dice, por ejemplo, que `producto1` debe ser un producto.

Fig. 2.4b Modelo de objetos



- Fíjese como cada **asociación** del diagrama de clases debe tener su propio **objeto** en el momento de la ejecución.

Una **clase** no habla de un escenario particular, sino del caso general. Nunca dice cuál es el valor de un **atributo**, sino que se contenta con afirmar cuáles son los atributos (nombre y tipo) que deben tener los objetos que son instancias de esa **clase**. Los objetos, por su parte, siempre pertenecen a una **clase**, en el sentido de que cumplen con la estructura de atributos que la **clase** exige. Por ejemplo, puede haber miles de tiendas diferentes, cada una de las cuales vende distintos productos a distintos precios. Piense que cada vez que instalamos el programa del caso de estudio en una tienda distinta, el dueño va a querer que los objetos que se creen para representarla reflejen el estado de su propia tienda.

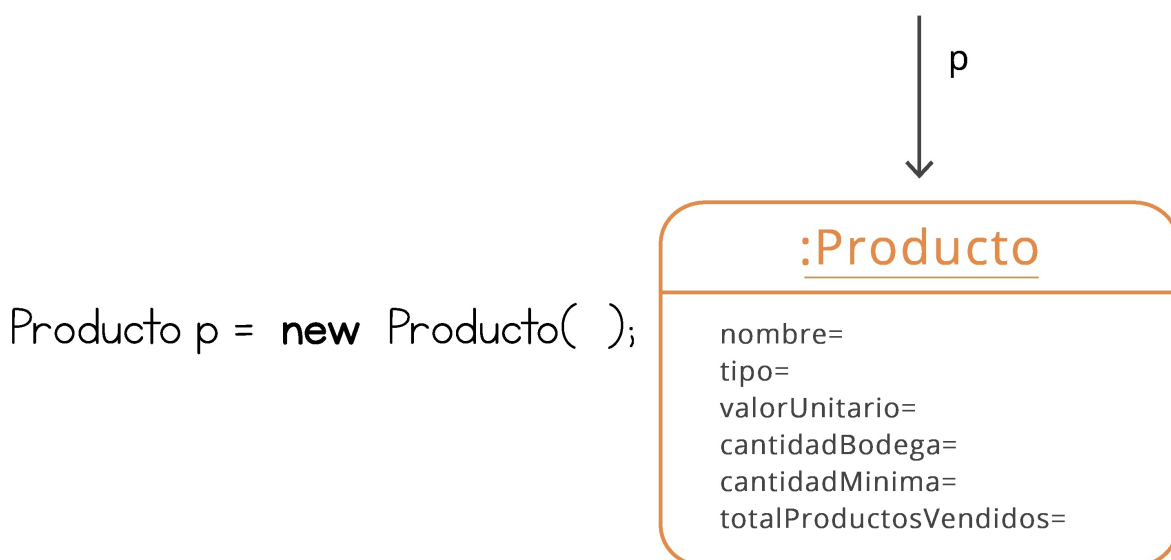
Los métodos de una **clase**, por su parte, siempre están en ella y no copiados en cada uno de sus objetos. Por esta razón cada **objeto** debe saber a qué **clase** pertenece, para poder buscar en ella los métodos que puede ejecutar. Los métodos están escritos de manera que se puedan utilizar desde todos los objetos de la **clase**. Cuando un **método** de la **clase** `Tienda` dice `producto1.darNombre()`, le está pidiendo a una tienda particular que busque en su propio escenario el **objeto** al cual se llega a través de la **asociación** `producto1`, y le pida a éste su nombre usando el **método** que todos los productos tienen para hacerlo. En este

sentido se puede decir que los métodos son capaces de resolver los problemas en abstracto, y que cada **objeto** los aplica a su propio escenario para resolver su problema concreto.

6.2. Creación de Objetos de una Clase

Recordemos la creación de objetos visto en el nivel anterior. Un **objeto** se crea utilizando la instrucción `new` y dando el nombre de la **clase** de la cual va a ser una instancia. Todas las clases tienen un **método** constructor por defecto, sin necesidad de que el programador tenga que crearlo. Como no es **responsabilidad** del computador darle un valor inicial a los atributos, cuando se usa este constructor, éstos quedan en un valor que se puede considerar indefinido. En la **figura 2.5** se muestra el resultado del llamado a este constructor.

Fig. 2.5 Creación de un **objeto usando la instrucción new**



- El resultado de ejecutar la instrucción del ejemplo es un nuevo **objeto**, con sus atributos no inicializados.
- Dicho **objeto** está "referenciado" por p, que puede ser un **atributo** o una **variable** de tipo Producto.

Para inicializar los valores de un **objeto**, se debe definir en la **clase** un constructor propio. En el siguiente ejemplo trabajaremos los conceptos vistos en el capítulo anterior, usando el caso de la tienda.

Ejemplo 10

Se hace la inicialización de los atributos de los objetos de la [clase](#).

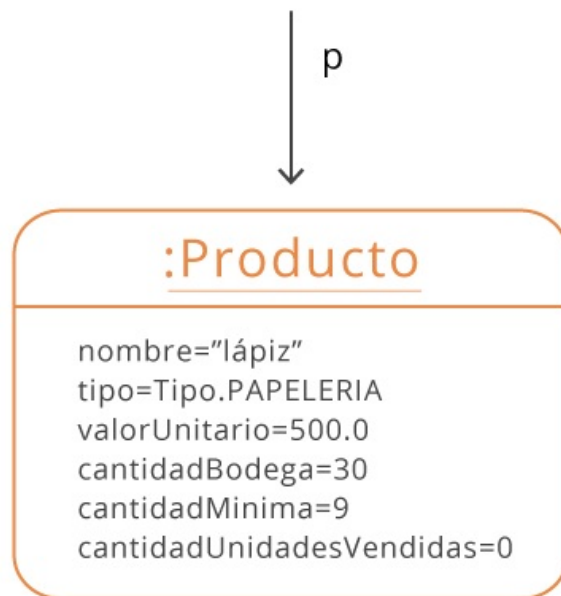
En este ejemplo mostramos los constructores de las clases Tienda y Producto, así como la manera de pedir la creación de un [objeto](#) de cualquiera de esos dos tipos.

```
public Producto(Tipo pTipo, String pNombre, double pValorUnitario, int pCantidadBodega
, int pCantidadMinima)
{
    tipo = pTipo;
    nombre = pNombre;
    valorUnitario = pValorUnitario;
    cantidadBodega = pCantidadBodega;
    cantidadMinima = pCantidadMinima;
    cantidadUnidadesVendidas = 0;
}
```

- El constructor exige 5 parámetros para poder inicializar los objetos de la [clase](#) Producto.
- En el constructor se asignan los valores de los parámetros a los atributos.

```
Producto p=new Producto(Tipo.PAPELERIA, "lápiz", 500.0, 30, 9);
```

- Este es un ejemplo de la manera de crear un [objeto](#) cuando el constructor tiene parámetros.



- Este es el **objeto** que se crea con la llamada anterior.
- El **objeto** creado se ubica en alguna parte de la memoria del computador. Dicho **objeto** es referenciado por el **atributo** o la **variable** llamada " p ".

```
public Tienda( )  
{  
    producto1 = new Producto( Tipo.PAPELERIA, "Lapiz", 550.0, 18, 5 );  
    producto2 = new Producto( Tipo.DROGUERIA, "Aspirina", 109.5, 25, 8 );  
    producto3 = new Producto( Tipo.PAPELERIA, "Borrador", 207.3, 30, 10 );  
    producto4 = new Producto( Tipo.SUPERMERCADO, "Pan", 150.0, 15, 20 );  
    dineroEnCaja = 0;  
}
```

- Puesto que es necesario que la tienda tenga 4 productos, su **método** constructor debe ser como el que se presenta. Supone que en la caja de la tienda no hay dinero al comenzar el programa.

Vamos a practicar la creación de escenarios usando los métodos constructores de las clases Tienda y Producto. En el programa del caso de estudio, la **responsabilidad** de crear el estado de la tienda sobre la cual se trabaja está en la **clase** principal del mundo (**clase** Tienda). En una situación real, dichos valores deberían leerse de un **archivo** o de una base

de datos, pero en nuestro caso se utilizará un escenario predefinido. Si quiere modificar los datos de la tienda sobre los que trabaja el programa, puede darle otros valores en el momento de construir las instancias.

Tarea 4

Objetivo: Generar habilidad en el uso de los constructores de las clases para crear escenarios

Cree los escenarios que se describen a continuación, dando la secuencia de instrucciones que los construyen. Suponga que dicha construcción se hace desde una **clase** externa a las clases Tienda y Producto.

Escenario 1

Una nueva tienda acaba de abrir y quiere usar el programa del caso de estudio con los siguientes productos:

1. Frasco de jarabe (para la gripe), producto de droguería, con un valor base de \$7.200 pesos.
2. Botella de alcohol, producto de droguería, a \$2.800 pesos la unidad.
3. Kilo de queso, producto de supermercado, a \$4.100 la unidad.
4. Resaltador, producto de papelería, a \$3.500 la unidad.

La siguiente tabla resume el inventario inicial de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido. Suponga que el valor inicial en caja es cero pesos.

Producto	Cantidad en bodega	Tope mínimo
jarabe	14	10
alcohol	12	8
queso	10	4
resaltador	20	10

Código

```
public Tienda( )
{
    producto1 = new Producto( Tipo.DROGUERIA, "jarabe", 7200.0, 14, 10 );
    producto2 = new Producto( Tipo.DROGUERIA, "alcohol", 2800.0, 12, 8 );
    producto3 = new Producto( Tipo.SUPERMERCADO, "queso", 4100.0, 10, 4 );
    producto4 = new Producto( Tipo.PAPELERIA, "resaltador", 3500.0, 20, 10 );
}
```

Escenario 2

Una nueva tienda acaba de abrir y quiere usar el programa del caso de estudio con los siguientes productos:

1. Kilo de arroz, producto de supermercado, con valor base de \$1.200 pesos.
2. Caja de cereal, producto de supermercado, a \$7.500 pesos.
3. Resma de papel, producto de papelería, a \$20.000 pesos la unidad.
4. Bolsa de algodón, producto de droguería, a \$4.800 pesos.

La siguiente tabla resume el inventario inicial de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido. Suponga que el valor inicial en caja es cero pesos.

Producto	Cantidad en bodega	Tope mínimo
arroz	6	7
cereal	5	5
papel	50	2
algodón	12	6

Código**Escenario 3**

Una nueva tienda acaba de abrir, y quiere usar el programa del caso de estudio con los siguientes productos:

1. Litro de aceite, producto de supermercado, con un valor base de \$6.500 pesos la unidad.
2. Crema dental, producto de supermercado, a \$5.100 pesos.
3. Kilo de pollo, producto de supermercado, a \$13.800 pesos la unidad.
4. Protector solar, producto de droguería, a \$16.000 la unidad.

La siguiente tabla resume el inventario inicial de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido. Suponga que el valor inicial en caja es cero pesos.

Producto	Cantidad disponible	Tope mínimo
aceite	13	10
crema dental	20	15
pollo	6	5
protector solar	3	3

Código

7. Instrucciones Condicionales

7.1. Instrucciones Condicionales Simples

Una **instrucción condicional** nos permite plantear la solución a un problema considerando los distintos casos que se pueden presentar. De esta manera, podemos utilizar un **algoritmo** distinto para enfrentar cada caso que pueda existir en el mundo. Considere el **método** de la **clase** Producto que se encarga de vender una cierta cantidad de unidades presentes en la bodega. Allí, se pueden presentar dos casos posibles, cada uno con una solución distinta: el primer caso es cuando la cantidad que se quiere vender es mayor que la cantidad disponible en la bodega (el pedido es mayor que la disponibilidad) y el segundo es cuando hay suficientes unidades del producto en la bodega para hacer la venta. En cada una de esas situaciones la solución es distinta y el **método** debe tener un **algoritmo** diferente.

Para construir una **instrucción condicional**, se deben identificar los casos y las soluciones, usando algo parecido a la tabla que se muestra a continuación:

Caso 1:

Expresión que describe el caso:

```
cantidad > cantidadBodega
```

Algoritmo para resolver el problema en ese caso:

```
// Vende todas las unidades disponibles
cantidadUnidadesVendidas += cantidadBodega;
cantidadBodega = 0;
```

Caso 2:

Expresión que describe el caso:

```
cantidad <= cantidadBodega
```

Algoritmo para resolver el problema en ese caso:

```
// Vende lo pedido por el usuario
cantidadUnidadesVendidas += cantidad;
cantidadBodega -= cantidad;
```

En el primer caso la solución es vender todo lo que hay en la bodega. En el segundo, vender lo pedido como [parámetro](#). En Java existe la [instrucción condicional](#) `if-else`, que permite expresar los casos dentro de un [método](#). La sintaxis en Java de dicha instrucción se ilustra en el siguiente fragmento de programa:

```
public class Producto
{
    ...
    public void vender( int pCantidad )
    {
        if( pCantidad > cantidadBodega )
        {
            totalProductosVendidos += cantidadBodega;
            cantidadBodega = 0;
        }
        else
        {
            totalProductosVendidos += pCantidad ;
            cantidadBodega -= pCantidad ;
        }
    }
}
```

- En lugar de una sola secuencia de instrucciones, se puede dar una secuencia para cada caso posible. El computador sólo va a ejecutar una de las dos secuencias.
- Es como si el [método](#) escogiera el [algoritmo](#) que debe utilizar para resolver el problema puntual que tiene, identificando la situación en la que se encuentra el [objeto](#).
- La [condición](#) caracteriza los dos casos que se pueden presentar. Note que con una sola [condición](#) debemos separar los dos casos.
- Los paréntesis alrededor de la [condición](#) son obligatorios.
- La [condición](#) es una [expresión](#) lógica, construida con operadores relacionales y lógicos.
- La parte del "`else`", incluida la segunda secuencia de instrucciones, es opcional. Si no se incluye, eso querría decir que para resolver el segundo caso no hay que hacer nada.

La instrucción `if-else` tiene tres elementos:

1. Una [condición](#) que corresponde a una [expresión](#) lógica capaz de distinguir los dos casos (su evaluación debe dar verdadero si se trata del primer caso y falso si se trata del segundo).
2. La solución para el primer caso.
3. La solución para el segundo caso. Al encontrar una [instrucción condicional](#), el computador evalúa primero la [condición](#) y decide a partir de su resultado cuál de las dos soluciones ejecutar. Nunca ejecuta las dos.

Si el **algoritmo** que resuelve uno de los casos sólo tiene una instrucción, es posible eliminar los corchetes, como se ilustra en el ejemplo 11. Allí también se puede apreciar que una **instrucción condicional** es sólo una instrucción más dentro de la secuencia de instrucciones que implementan un **método**.

Ejemplo 11

En este ejemplo se presentan algunos métodos de la **clase** Producto, para mostrar la sintaxis de la instrucción `if-else` de Java. Los métodos aquí presentados no son necesariamente los que escribiríamos para implementar los requerimientos funcionales del caso de estudio, pero sirven para ilustrar distintos aspectos de las instrucciones condicionales.

```
public boolean haySuficiente( int pCantidad )
{
    boolean suficiente;
    if( pCantidad <= cantidadBodega )
        suficiente = true;
    else
        suficiente = false;
    return suficiente;
}
```

- Como la secuencia de instrucciones de cada caso tiene una sola instrucción, se pueden eliminar los corchetes.
- Fíjese que dejamos el resultado de cada caso en la misma **variable**, de manera que al hacer el retorno del **método** siempre se encuentre allí el resultado. ¿Qué hace este **método**?
- Una **instrucción condicional** se puede ver como otra instrucción más del **método**. Puede haber instrucciones antes y después de ella.

```
public boolean haySuficiente( int pCantidad )
{
    return pCantidad <= cantidadBodega;
}
```

- El **método** anterior también se podría escribir de esta manera, un poco más sencilla. ¿Qué ganamos escribiéndolo así?


```
public double darPrecioPapeleria( boolean conIVA )
{
    double precioFinal = valorUnitario;

    if( conIVA )
        precioFinal = precioFinal * ( 1+IVA_PAPELERIA );

    return precioFinal;
}
```

- Si en el segundo de los casos de una **instrucción condicional** no es necesario hacer nada, no se debe escribir ninguna instrucción, tal como se muestra en el ejemplo.
- ¿Está claro el problema que resuelve el **método**?

```
public void ajustarPrecio( )
{
    if( totalProductosVendidos < 100 )
    {
        valorUnitario = valorUnitario * 80 / 100;
    }
    else
    {
        valorUnitario = valorUnitario * 1.1;
    }
}
```

- En este **método**, si se han vendido menos de 100 unidades, se hace un descuento del 20% en el precio del producto.
- Si se han vendido 100 o más unidades, se aumenta en un 10% el precio.
- En las instrucciones condicionales, incluso si sólo hay una instrucción para resolver cada caso, es buena idea utilizar los corchetes para facilitar la lectura del código, es buena idea utilizar los corchetes. En algunos casos, incluso, son indispensables para evitar ambigüedades.

Tenga cuidado de no escribir un ";" después de la **condición**, porque el computador lo va a interpretar como si la solución al caso fuera no hacer nada (una instrucción vacía).

7.2 Condicionales en Cascada

Cuando el problema tiene más de dos casos, es necesario utilizar una cascada (secuencia) de instrucciones `if-else`, en donde cada **condición** debe indicar sin ambigüedad la situación que se quiere considerar. Suponga por ejemplo que queremos calcular el IVA de

un producto. Puesto que el valor que se paga de impuestos por un producto depende de su tipo, es necesario considerar los tres casos siguientes:

Caso 1

Expresión que describe el caso:

```
( tipo == Tipo.SUPERMERCADO )
```

Algoritmo para resolver el problema en ese caso:

```
return IVA_SUPERMERCADO;
```

Caso 2

Expresión que describe el caso:

```
( tipo == Tipo.DROGUERIA )
```

Algoritmo para resolver el problema en ese caso:

```
return IVA_DROGUERIA;
```

Caso 3

Expresión que describe el caso:

```
( tipo == Tipo.PAPELERIA )
```

Algoritmo para resolver el problema en ese caso:

```
return IVA_PAPELERIA;
```

El **método** de la **clase** Producto para determinar el IVA que hay que pagar sería de la siguiente forma (no es la única solución, como veremos más adelante):

```
public double darIVA( )
{
    if( tipo == Tipo.PAPELERIA )
    {
        return IVA_PAPELERIA;
    }
    else if( tipo == Tipo.SUPERMERCADO )
    {
        return IVA_SUPERMERCADO;
    }
    else
    {
        return IVA_DROGUERIA;
    }
}
```

- Para representar los tres casos posibles, utilizamos una [instrucción condicional](#) en el "else" del primer caso. Esa manera de encadenar las instrucciones condicionales para poder considerar cualquier número de casos se denomina "en cascada".
- Una [instrucción condicional](#) puede ir en cualquier parte donde pueda ir una instrucción del lenguaje Java. Esto lo retomaremos en capítulos posteriores.

```
public double darIVA( )
{
    double resp = 0.0;

    if( tipo == Tipo.PAPELERIA )
    {
        resp = IVA_PAPELERIA;
    }
    else if( tipo == Tipo.SUPERMERCADO )
    {
        resp = IVA_SUPERMERCADO;
    }
    else
    {
        resp = IVA_DROGUERIA;
    }

    return resp;
}
```

En esta segunda solución del [método](#), en lugar de hacer un retorno en cada caso, guardamos la respuesta en una [variable](#) y luego la retornamos al final.

Al usar varias instrucciones if en cascada hay que tener cuidado con la ambigüedad que puede surgir con la parte else. Es mejor usar siempre corchetes para asegurarse de que el computador lo va a interpretar de la manera adecuada.

Tarea 5

Objetivo: Practicar el uso de las instrucciones condicionales simples para expresar el cambio de estado que debe hacerse en un [objeto](#), en cada uno de los casos identificados.

Escriba el código de cada uno de los métodos descritos a continuación. Tenga en cuenta la [clase](#) en la cual está el [método](#) y la información que se entrega como [parámetro](#).

Para la clase: Producto

Aumentar el valor unitario del producto, utilizando la siguiente política: si el producto cuesta menos de \$1000, aumentar el 1%. Si cuesta entre \$1000 y \$5000, aumentar el 2%. Si cuesta más de \$5000 aumentar el 3%.

```
public void subirValorUnitario( )
{

}

}
```

Recibir un pedido, sólo si en bodega se tienen menos unidades de las indicadas en el tope mínimo. En caso contrario el [método](#) no debe hacer nada.

```
public void hacerPedido( int pCantidad )
{

}

}
```

Modificar el precio del producto, utilizando la siguiente política: si el producto es de droguería o papelería debe disminuir un 10%. Si es de supermercado debe aumentar un 5%.

```
public void cambiarValorUnitario()  
{  
  
  
  
  
  
  
  
  
  
}
```

Para la clase: Tienda

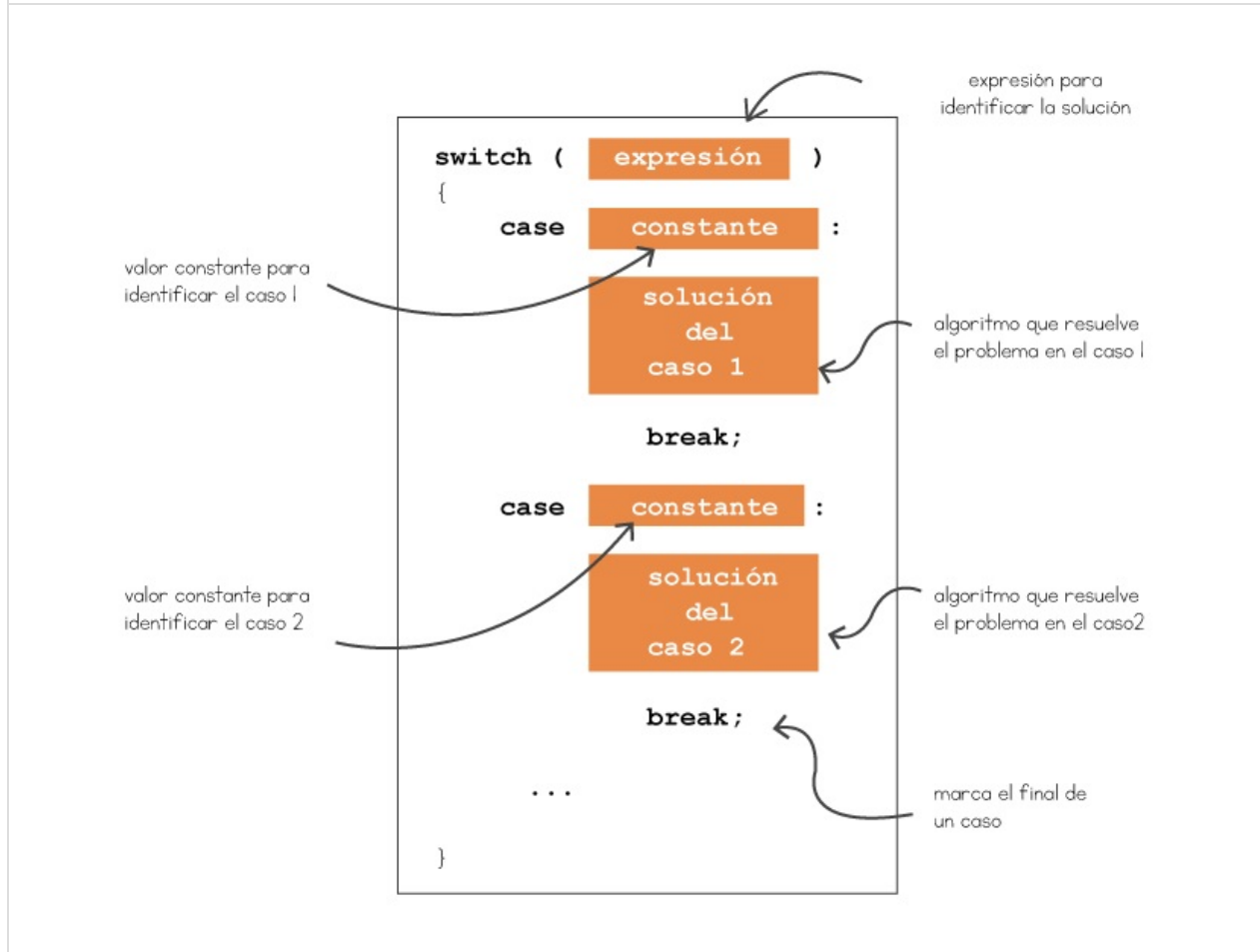
Vender una cierta cantidad del producto cuyo nombre es igual al recibido como **parámetro**. El **método** retorna el número de unidades efectivamente vendidas. Suponga que el nombre que se recibe como **parámetro** corresponde a uno de los productos de la tienda. Utilice el **método** vender de la **clase** Producto como parte de su solución.

```
public int venderProducto( String pNombreProducto, int pCantidad )
{
```

Calcular el número de productos de papelería que se venden en la tienda.

Una **instrucción condicional** compuesta (`switch`) es una manera alternativa de expresar la solución de un problema para el cual existe un conjunto de casos, cada uno con un **algoritmo** distinto para resolverlo. Esta instrucción tiene la restricción de que cada caso debe estar identificado con un valor de tipo entero, String o del tipo de una enumeración.

En el ejemplo 12 se presenta la solución del **método** que calcula el IVA de un producto, usando una instrucción condicional compuesta.

Fig. 2.6 Estructura de la instrucción switch de Java

Ejemplo 12

Objetivo: Ilustrar el uso de instrucciones condicionales compuestas.

En este ejemplo se presenta el [método](#) que calcula el IVA que debe pagar un producto, dependiendo de su tipo.

```
public double darIVA( )
{
    double iva = 0.0;

    switch( tipo )
    {
        case PAPELERIA:
        {
            iva = IVA_PAPELERIA;
            break;
        }
        case SUPERMERCADO:
        {
            iva = IVA_SUPERMERCADO;
            break;
        }
        case DROGUERIA:
        {
            iva = IVA_DROGUERIA;
            break;
        }
    }

    return iva;
}
```

- Este [método](#) de la [clase](#) Producto tiene tres casos posibles, cada uno identificado con un valor de una enumeración (candidato ideal para la instrucción switch).
- La [expresión](#) que va a permitir distinguir los casos se construye simplemente con el [atributo](#) "tipo".
- Cada caso se introduce con la palabra reservada de Java "case" y se cierra con un "break". Después de la instrucción "case" va el valor que identifica el caso. En nuestro ejemplo, el valor se identifica con las constantes que representan los tres tipos posibles de productos: PAPELERIA, SUPERMERCADO o DROGUERIA.

Dado que siempre es posible escribir una [instrucción condicional](#) compuesta como una cascada de condicionales simples, la pregunta que nos debemos hacer es ¿cuándo usar una [instrucción condicional](#) compuesta? La respuesta es que siempre que se pueda utilizar la instrucción `switch` en lugar de una cascada de `if` es conveniente hacerlo, por dos razones:

1. Eficiencia, ya que de este modo sólo se evalúa una vez la [expresión](#) aritmética, mientras que en la cascada se evalúan una a una las condiciones, descartándolas.
2. Claridad en el programa, porque es más fácil de leer y mantener un programa escrito de esta manera.

Y la segunda pregunta es, ¿cuándo no intentar usar una **instrucción condicional** compuesta? La respuesta es: cuando los casos no están identificados por valores enteros, Strings o enumeraciones. Si se tienen, por ejemplo, un valor real como identificadores de los casos, la única opción es usar una cascada de instrucciones `if-else`.

Tarea 6

Objetivo: Utilizar instrucciones condicionales para expresar un conjunto de casos y soluciones asociadas con los mismos.

Escriba el código de cada uno de los métodos descritos a continuación. Tenga en cuenta la **clase** en la cual está el **método** y la información que se entrega como **parámetro**.

Para la clase: Producto

Dar el nombre del tipo del producto. Por ejemplo, si el producto es de tipo SUPERMERCADO, el **método** debe retornar la cadena: "El producto es de supermercado".

```
public String nombreTipoProducto( )
{
```

Aumentar el precio del producto, siguiendo esta regla: si es un producto de droguería debe aumentar el 1%, si es de supermercado el 3% y si es de papelería el 2%.


```
public double darPrecioProducto( String pNombreProducto )
{
```

8. Responsabilidades de una Clase

8.1. Tipos de Método

Los métodos en una **clase** se clasifican en tres tipos, según la operación que realicen:

- **Métodos constructores:** tienen la **responsabilidad** de inicializar los valores de los atributos de un **objeto** durante su proceso de creación.
- **Métodos modificadores:** tienen la **responsabilidad** de cambiar el estado de los objetos de la **clase**. Son los responsables de "hacer".
- **Métodos analizadores:** tienen la **responsabilidad** de calcular información a partir del estado de los objetos de la **clase**. Son los responsables de "saber".

8.2. ¿Cómo Identificar las Responsabilidades?

En esta parte sólo veremos algunas guías intuitivas respecto de cómo identificar las responsabilidades de una **clase**. Utilizamos dos estrategias complementarias que se pueden utilizar en cualquier orden y que se resumen a continuación:

- Una **clase** es responsable de administrar la información que hay en sus atributos. Por esta razón se debe tratar de buscar el conjunto de servicios que reflejen las operaciones típicas del elemento del mundo que la **clase** representa.
- Una **clase** es responsable de ayudar a sus vecinos del modelo del mundo y colaborar con ellos en la solución de sus problemas. En este caso la pregunta que nos debemos hacer es, ¿qué servicios necesitan las demás clases que les preste la **clase** que estamos diseñando? A partir de la respuesta a esta pregunta, iremos agregando servicios hasta que el problema global tenga solución.

Para las dos estrategias es conveniente hacer el recorrido por tipo de **método**, diseñando primero los constructores, luego los modificadores y, finalmente, los analizadores. En el nivel 4 de este libro retomaremos este problema de asignar responsabilidades a las clases.

Una vez que se han definido los servicios que va a prestar una **clase**, debemos definir los parámetros y el tipo de retorno. Para definir los parámetros de un **método**, debemos preguntarnos cuál es la información externa a la **clase** que se necesita para poder prestar el servicio. Para definir el tipo de retorno debemos preguntarnos qué información está esperando aquél que solicitó el servicio.

Tarea 7

Objetivo: Identificar y describir los métodos que representan las principales responsabilidades de una [clase](#).

Para el caso de estudio que se presenta a continuación construya el diagrama de clases e identifique los principales métodos.

Una empresa de transporte tiene 3 camiones para llevar carga de una ciudad a otra del país. De cada camión se tiene su matrícula (6 caracteres), su capacidad (en kilogramos) y el consumo de gasolina por kilómetro (un valor real en litros/kilómetro) y la carga actual (en kilogramos). Se quiere construir un programa que permita optimizar el uso de los camiones. Para esto debe tener una única opción que determina cuál es el mejor camión para transportar una cierta carga entre dos ciudades. El mejor camión es aquél que, siendo capaz de transportar la carga, consume la mínima cantidad de gasolina.

Requerimiento funcional

Nombre	<div></div>	
Resumen	<div></div>	
Entradas	<div></div>	
Resultado	<div></div>	

Diagrama de clases:



Clase EmpresaTransporte

Nombre del **método**:



Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

Nombre del método:

Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

Nombre del método:

Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

Nombre del método:

Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

Clase Camion

Nombre del método:

Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

Nombre del método:

Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

Nombre del método:

Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

Nombre del método:

Tipo de método	<div></div>	
Responsabilidad	<div></div>	
Parámetros	<div></div>	
Retorno	<div></div>	

9. Eclipse: Nuevas Opciones

En esta sección se cubren los siguientes temas:

- Uso de Eclipse para formatear una [clase](#) (concepto de *profile*). Se presentan las ventajas de mantener un correcto formato en los programas.
- Uso de Eclipse para localizar una declaración.
- Uso de Eclipse para localizar todos los clientes de un [método](#) (aquellos que lo usan).
- Uso de Eclipse para cambiar el nombre de un [atributo](#), [variable](#) o [método](#). Ventajas de hacerlo de esta manera y riesgo si hay errores de compilación.
- La siguiente tarea le propondrá una secuencia de acciones, que pretenden mostrarle la manera de hacer lo anteriormente mencionado en el [ambiente de desarrollo](#) Eclipse.

Tarea 8

Objetivo: Trabajar en Eclipse sobre la solución del caso de estudio, mostrando las nuevas opciones del [ambiente de desarrollo](#) que se introducen en este nivel.

Localice en el sitio del proyecto Cupi2 [la solución](#) del caso de estudio de este nivel. Copie en un directorio de trabajo dicha solución. Ejecute Eclipse y cree un nuevo proyecto que la contenga. Siga los pasos que se dan a continuación:

Paso I: ejecutar la aplicación

1. La [clase](#) InterfazTienda es la [clase](#) principal del programa. Localícela y selecciónela en el explorador de paquetes. Si tiene dificultades en esto, consulte la manera de hacerlo en el capítulo anterior.
2. Para ejecutar la [clase](#) principal de un programa, seleccione el comando *Run as Java Application*. Puede hacerlo desde la barra de herramientas, el menú principal o el menú emergente que aparece al hacer clic derecho sobre la [clase](#).

Paso II: dar formato al código fuente

1. Localice el [profile \(perfil\) de formato](#) en el sitio del proyecto Cupi2. Puede encontrarlo bajo el título "*Perfil Cupi2 para Eclipse*". El *profile* reúne un conjunto de preferencias de formato en el [código fuente](#), tales como indentación, posición de los corchetes, manejo de las líneas en blanco, comentarios, etc.
2. Instale el *profile* en Eclipse. Para esto seleccione la opción *Window/Preferences* del menú principal. En la [ventana](#) que aparece localice la zona *Java/Code Style/Code*

Formatter. Utilice el botón *Import...* para cargar el [archivo](#) mencionado en el punto anterior.

3. El hecho de cargar un *profile* no cambia automáticamente el formato de las clases. Seleccione y abra la [clase](#) Producto en el explorador de paquetes. Cambie el formato de los métodos. Elimine algunos espacios en las expresiones o cambie la indentación de las instrucciones.
4. Ahora aplíquelo formato a la [clase](#) Producto seleccionando la opción *Source/Format* en el menú emergente del clic derecho o con *ctrl+mayús+F*. El formato ayuda a organizar el [código fuente](#), mejorando su legibilidad y consistencia. Cuando se aplica formato a una [clase](#), Eclipse utiliza la información que aparece en el *profile* que esté activo. Note cómo el programa recupera su estado inicial.
5. Para darle formato a una sola sección de la [clase](#), seleccione la sección y aplíquelo el formato como en el paso anterior. Adquiera la buena práctica de aplicar el formato a todos sus proyectos antes de entregarlos.

Paso III: localizar rápidamente el código fuente de una clase o método

1. Seleccione y abra la [clase](#) Tienda en el explorador de paquetes. Localice la declaración de cualquiera de los atributos de la [clase](#) Producto (producto1, producto2, producto3 o producto4). Oprima la tecla *ctrl* y al mismo tiempo ubique el cursor sobre la palabra "Producto" en la declaración. La palabra "Producto" se resalta con un subrayado.
2. Haga clic sobre la palabra resaltada: se abrirá la [clase](#) Producto para ser consultada. En general, es posible localizar la declaración de cualquier elemento del programa utilizando esta misma interacción. Basta con posicionarse sobre el elemento cuya declaración queremos consultar y con la combinación *ctrl+clic* llegamos a dicho punto del programa.
3. Si desea puede ver el [video explicativo](#) en el sitio del proyecto.

Paso IV: localizar rápidamente los lugares donde se invoca un método

1. Seleccione y abra la [clase](#) Producto en el explorador de paquetes. Localice en el editor la declaración de cualquiera de los métodos o atributos de esta [clase](#).
2. Busque todos los lugares del programa en donde se invoca dicho [método](#), seleccionando la opción *Navigate_ / Open Call Hierarchy _* en el menú principal o en el menú emergente que aparece al hacer clic derecho sobre el [método](#) o con *ctrl+alt+h*.
3. En la vista de búsqueda de Eclipse se presentan todos los métodos en los que existe un llamado al [método](#) o [atributo](#) seleccionado.
4. Si selecciona un [método](#), señala el lugar dónde este [método](#) fue llamado.

5. Si desea puede ver el [video explicativo](#) en el sitio del proyecto.
6. Repita el procedimiento anterior con el [método](#) constructor de la [clase](#) Tienda, para llegar hasta la [clase](#) de la [interfaz de usuario](#) que crea la tienda.
7. https://sicuaplus.uniandes.edu.co/bbcswebdav/users/cupitaller/Videos/Jerarquia_llamados.mp4

Paso V: cambiar los elementos de una clase

1. Seleccione y abra la [clase](#) Producto en el explorador de paquetes. Localice la declaración del [atributo](#) valorUnitario en dicha [clase](#).
2. Cambie el nombre de este [atributo](#) a valorUnidad, seleccionando la opción *Refactor/Rename* en el menú principal o en el menú emergente que aparece al hacer clic derecho sobre el [atributo](#). Esta operación realiza la modificación en todos los puntos del programa en los cuales se utiliza dicho [atributo](#). La ventaja de hacer de esta manera los cambios es que el [compilador](#) ayuda a no cambiar por error otros elementos del programa.
3. Localice el [método](#) abastecer en la [clase](#) Producto. Cambie el nombre del [parámetro](#) pCantidad a pNumeroUnidades, de la misma manera que en el punto anterior. Esta misma técnica sirve para cambiar los nombres de los métodos. Si en algún punto del programa hay errores de compilación, es un poco arriesgado hacer los cambios de nombre mencionados en esta parte, ya que dichos errores pueden confundir al [compilador](#) y llevar a Eclipse a dejar de hacer algunos cambios necesarios.
4. Si desea puede ver el [video explicativo](#) en el sitio del proyecto.

10. Hojas de Trabajo

10.1 Hoja de Trabajo N° 1: Un Estudiante

Descargue esta hoja de trabajo a través de los siguientes enlaces: [Descargar PDF](#) | [Descargar Word](#).

Enunciado. Analice el siguiente enunciado e identifique el mundo del problema, lo que se quiere que haga el programa y las restricciones para desarrollarlo.

Se desea construir una aplicación para el manejo de información de los cursos que está tomando un estudiante. El estudiante toma solo 4 cursos en el semestre. Los datos personales del estudiante que maneja la aplicación son código, nombre y apellido.

De cada curso se conoce:

- Código. Es el identificador del curso y no pueden haber dos cursos con el mismo código.
- Nombre.
- Departamento. Puede ser Matemáticas, Física, Sistemas o Biología.
- Cantidad de créditos.
- Nota obtenida en el curso. Este valor debe estar entre 1.5 y 5.

Para poder calcular el promedio del estudiante, se deben ponderar las notas, teniendo en cuenta la cantidad de créditos de las materias. Para esto, para cada curso se debe multiplicar la nota del curso con su cantidad de créditos, sumar estos valores y dividir esta suma por la cantidad total de créditos vistos por el estudiante. Por ejemplo, si el estudiante ha terminado dos materias, “Cálculo 1” y “Física 1”, la primera de 4 créditos y la segunda de tres, con las siguientes notas:

- Cálculo 1: 4,5
- Física 1: 3,5

El promedio del estudiante es:

- $(4,5 \times 4 + 3,5 \times 3) / 7 = 4,07$

Adicionalmente, se quiere poder saber si un estudiante está en prueba académica o si es candidato para beca. Para esto se debe tener en cuenta las siguientes reglas.

- Se considera que un estudiante está en prueba académica si su promedio es inferior a 3.25.

- Se considera que un estudiante es candidato a beca si su promedio es igual o superior a 4.75.

La aplicación debe permitir: (1) visualizar la información del estudiante, (2) visualizar la información de los cursos, (3) modificar la información de un curso, (4) asignar una nota a un curso (5) calcular el promedio del estudiante (6) indicar si el estudiante está en prueba académica, (7) indicar si el estudiante es candidato a beca.

La interfaz del programa es la siguiente:

Información del estudiante	
Código:	201612345
Nombre:	Juliana
Apellido:	Ramírez
Promedio:	0

ISIS1204		MATE1203	
Nombre:	AP01	Nombre:	Cálculo diferencial
Departamento:	Ing. Sistemas	Departamento:	Matemática
Créditos:	3	Créditos:	3
Nota:	0.0	Nota:	0.0
<input type="button" value="Cambiar curso"/>	<input type="button" value="Asignar nota"/>	<input type="button" value="Cambiar curso"/>	<input type="button" value="Asignar nota"/>

FISI1100		BIOL1405	
Nombre:	Física 1	Nombre:	Biología celular
Departamento:	Física	Departamento:	Biología
Créditos:	4	Créditos:	4
Nota:	0.0	Nota:	0.0
<input type="button" value="Cambiar curso"/>	<input type="button" value="Asignar nota"/>	<input type="button" value="Cambiar curso"/>	<input type="button" value="Asignar nota"/>

Opciones			
<input type="button" value="Candidato beca"/>	<input type="button" value="Prueba académica"/>	<input type="button" value="Opción 1"/>	<input type="button" value="Opción 2"/>

Requerimientos funcionales. Especifique los siete requerimientos funcionales descritos en el enunciado.

Requerimiento Funcional 1

Nombre	R1 – Visualizar la información del estudiante.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	R2 – Visualizar la información de los cursos.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	R3 – Modificar la información de un curso.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 4

Nombre	R4 – Asignar una nota a un curso.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 5

Nombre	R5 – Calcular el promedio del estudiante.
Resumen	
Entradas	
Resultado	

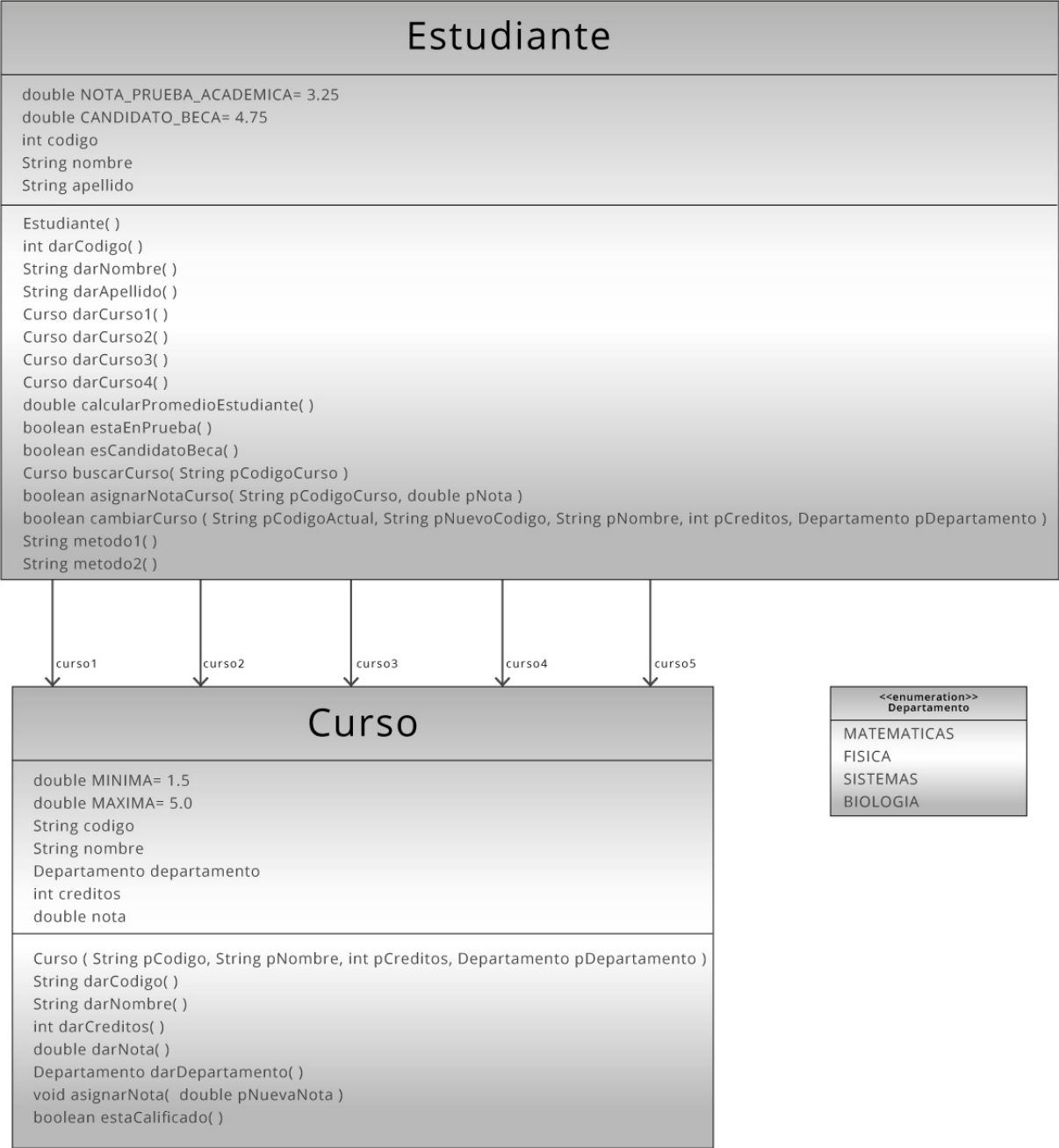
Requerimiento Funcional 6

Nombre	R6 – Indicar si el estudiante está en prueba académica.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 7

Nombre	R7 – Indicar si el estudiante es candidato a beca.
Resumen	
Entradas	
Resultado	

Modelo conceptual. Estudie el siguiente modelo conceptual



Declaración de las clases. Complete las declaraciones de las siguientes clases.

```
public class Estudiante
{
    // -----
    // Atributos
    // -----

}
```

```
public class Curso
{
    // -----
    // Atributos
    // -----

}
```

Creación de Expresiones. Para cada uno de los siguientes enunciados, escriba la **expresión** que lo representa. Tenga en cuenta la **clase** dada para determinar los elementos disponibles.

Curso	¿El nombre del curso es "Cálculo 1"?	
Curso	¿El curso ya tiene una nota asignada?	
Curso	¿El curso tiene más de tres créditos?	
Curso	¿El curso fue aprobado?	
Estudiante	¿El código del estudiante es "1234"?	
Estudiante	¿El primer curso tiene una nota asignada?	
Estudiante	¿El segundo curso pertenece al departamento de matemáticas?	
Estudiante	¿Cuál es el promedio del estudiante?	

Desarrollo de métodos. Escriba el código de los métodos indicados.

Método 1

Clase: Curso

Descripción: Retorna el código del curso.

```
public String darCodigo( )
{

}

}
```

Método 2

Clase: Curso

Descripción: Indica si el curso ya fue calificado (tiene una nota distinta de cero).

```
public boolean estaCalificado( )
{

}

}
```

Método 3

Clase: Estudiante

Descripción: Retorna el nombre del estudiante.

```
public String darNombre( )
{

}

}
```

Método 4

Clase: Estudiante

Descripción: Indica si el estudiante ya tiene los cuatro cursos pertenecen al mismo departamento.

```
public boolean pertenecenMismoDepartamento( )
{

}

}
```

Método 5

Clase: Estudiante

Descripción: Calcula el promedio de los cursos que ya tienen nota. Si ningún curso tiene nota asignada, retorna cero.

```
public double calcularPromedioEstudiante( )
{

}

}
```

Método 6

Clase: Estudiante

Descripción: Busca y retorna el curso que tiene el código que se recibe como **parámetro**. Si ningún curso tiene dicho código, el **método** retorna null.

```
public Curso buscarCurso( String pCodigoCurso )
{

}

}
```

Método 7

Clase: Estudiante

Descripción: Indica si el estudiante se encuentra en prueba académica. Retorna verdadero si está en prueba académica, false de lo contrario.

```
public boolean estaEnPrueba( )
{

}

}
```

