Programación Orientada a Objetos. Práctica 2.1

Tema 2. Clases, Objetos y Composición

Características de la Práctica

En esta práctica, el alumno aprenderá la definición de clases y la creación de objetos. Así como que los objetos se componen de atributos (variables de instancia, que también pueden ser constantes) que almacenan el estado interno de los objetos. Los constructores permiten crear los objetos y asignar valores iniciales a los atributos del objeto. Los métodos permiten manipular y acceder al estado interno de los objetos.

Además, en la segunda parte de la práctica, el alumno aprenderá el concepto de **composición**, como uno de los pilares fundamentales en el diseño de las Clases, de tal forma que nos permite definir nuevas clases más complejas mediante la composición de clases más simples previamente definidas. También aprenderá cómo un constructor puede invocar a otro constructor en la creación de un objeto.

Ejercicio 1. (proyecto prJarras)

El objetivo de este ejercicio es crear una clase `Jarra` perteneciente al paquete `prJarras` del proyecto `prJarras`. Utilizaremos esta clase `Jarra` para *simular* algunas de las acciones que podemos realizar con una jarra.

Nuestras jarras van a poder contener cierta cantidad de agua. Así, cada jarra tiene una determinada capacidad (en litros) que será la misma (constante) durante la vida de la jarra, y su valor será especificado en el momento de la construcción del objeto. En un momento determinado una jarra dispondrá de una cantidad de agua que podrá variar en el tiempo. Las acciones que podremos realizar sobre una jarra son:

- Llenar la jarra por completo desde un grifo.
- Vaciar la jarra completamente.
- Llenar la jarra con el agua que contiene otra jarra (bien hasta que la jarra receptora quede llena o hasta que la jarra que volcamos se vacíe por completo).
- Acceder a su representación textual.

Por ejemplo: Disponemos de dos jarras **A** y **B** de capacidades 7 y 4 litros respectivamente. Podemos llenar la jarra **A** (no podemos echar menos del total de la jarra porque no sabríamos a ciencia cierta cuánta agua tendría). Luego volcar **A** sobre **B** (no cabe todo, por lo que en **A** quedan 3 litros y **B** está llena). Ahora vaciar **B**. Después volver a volcar **A** sobre **B**. En esta situación, **A** está vacía y **B** tiene 3 litros.

Hay que definir la clase `Jarra` con los métodos necesarios para realizar las operaciones que acabamos de describir. Por lo tanto, la clase `Jarra` contiene dos variables de instancia, de tal forma que la variable de instancia `capacidad` es constante (`final`) y determina la capacidad del objeto `Jarra`, mientras que la variable de instancia `contenido` determina la cantidad de líquido contenido en un momento dado en el objeto `Jarra`, considerando que el contenido de la jarra podrá variar a lo largo del tiempo. Además, la clase `Jarra` define los siguientes constructores y métodos públicos que permitirán manipular los objetos de la clase `Jarra`:

- `Jarra(int)`

  Construye un nuevo objeto `Jarra` con la *capacidad* que se recibe como parámetro, y el *contenido* de la jarra vacío. Si el valor recibido como parámetro es menor o igual a cero, entonces lanzará la excepción `RuntimeException`.

- `capacidad():int`

  Devuelve la *capacidad* del objeto jarra.

- `contenido():int`

  Devuelve el *contenido* actual del objeto jarra.

- `llena():void`

  Llena el *contenido* del objeto jarra al máximo de su capacidad.

- `vacia():void`

  Vacía el *contenido* del objeto jarra completamente.

- `llenaDesde(Jarra):void`

  Llena el *contenido* de la jarra actual (receptora) con el *contenido* de la jarra que se recibe como parámetro (emisora), bien hasta que la jarra receptora quede llena o hasta que la jarra emisora se vacíe por completo. Si el objeto actual (`this`) es el mismo objeto que el objeto recibido como parámetro, entonces este método **no realizará** ninguna acción y lanzará la excepción `RuntimeException`.[1]

- `toString():String  // @Redefinición`

  Devuelve un `String` con la representación textual del objeto jarra en el formato `J(`*cap*`,`*cnt*`)`.

La Figura 1 muestra el diagrama UML de la clase `Jarra` que se encuentra en el paquete `prJarras`.

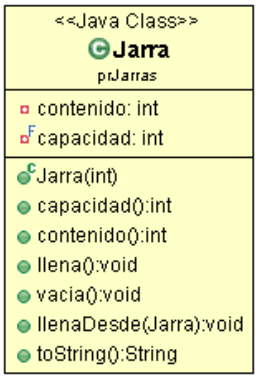

Figura 1: Diagrama de clases UML

Para probar nuestra nueva clase vamos a construir una aplicación (clase distinguida `EjemploUsoJarras1`) en el paquete *por defecto* (*anónimo*):

- `EjemploUsoJarras1`: se crearán dos jarras, una (`jarraA`) con capacidad para 7 litros y otra (`jarraB`) para 4 litros. Una vez creadas hemos de realizar las siguientes operaciones: llenar `jarraA`, mostrar por pantalla ambas jarras, volcar `jarraA` sobre `jarraB`, mostrar por pantalla ambas jarras, vaciar `jarraB`, mostrar por pantalla ambas jarras, volcar `jarraA` sobre `jarraB`, mostrar por pantalla ambas jarras.

```
J(7, 7), J(4, 0)
J(7, 3), J(4, 4)
J(7, 3), J(4, 0)
J(7, 0), J(4, 3)
```

Programación Orientada a Objetos. Práctica 2.1

Ejercicio 2. (proyecto prJarras)

En este ejercicio, definiremos una clase `Mesa` perteneciente también al paquete `prJarras` del proyecto `prJarras`. Esta clase contiene dos jarras (definidas en el ejercicio anterior). Así, cuando se construye

[1] Nótese que el operador de comparación `==` es útil para comprobar si dos variables referencian al mismo objeto.

una mesa, se especificará la capacidad de cada una de las dos jarras que componen el objeto mesa que se está construyendo. Las dos jarras se construirán con la capacidad especificada y con contenido vacío. Además, también será posible construir una mesa proporcionando las dos jarras que contendrá la mesa. También podremos realizar las siguientes acciones con las jarras de la mesa, especificando cada jarra según su número de identificación, 1 ó 2 (en caso de recibir una identificación de jarra errónea, se lanzará la excepción `RuntimeException`):

- `Mesa(Jarra,Jarra)`

  Construye un nuevo objeto `Mesa` que contiene dos jarras, que se reciben como parámetros. Si ambos objetos que se reciben como parámetros son el mismo objeto, entonces lanzará la excepción `RuntimeException`.[2]

- `Mesa(int,int)`

  Construye un nuevo objeto `Mesa` que contiene dos jarras, que serán también construidas con las *capacidades* que se reciben como parámetros, y el *contenido* de las jarras vacío.

- `capacidad(int):int`

  Devuelve la *capacidad* de la jarra especificada como parámetro, que podrá ser 1 ó 2.

- `contenido(int):int`

  Devuelve el *contenido* actual de la jarra especificada como parámetro, que podrá ser 1 ó 2.

- `llena(int):void`

  Llena el *contenido* de la jarra especificada como parámetro, que podrá ser 1 ó 2.

- `vacia(int):void`

  Vacía el *contenido* de la jarra especificada como parámetro, que podrá ser 1 ó 2.

- `llenaDesde(int):void`

  Llena el *contenido* de la jarra receptora con el *contenido* de la jarra emisora, especificada como parámetro, que podrá ser 1 ó 2, bien hasta que la jarra receptora quede llena o hasta que la jarra emisora se vacíe por completo.

- `toString():String  // @Redefinición`

  Devuelve un `String` con la representación textual del objeto mesa en el formato `M(J(`*cap*`,`*cnt*`),J(`*cap*`,`*cnt*`))`.

La Figura 2 muestra el diagrama UML de la clase `Mesa` que se encuentra en el paquete `prJarras`.

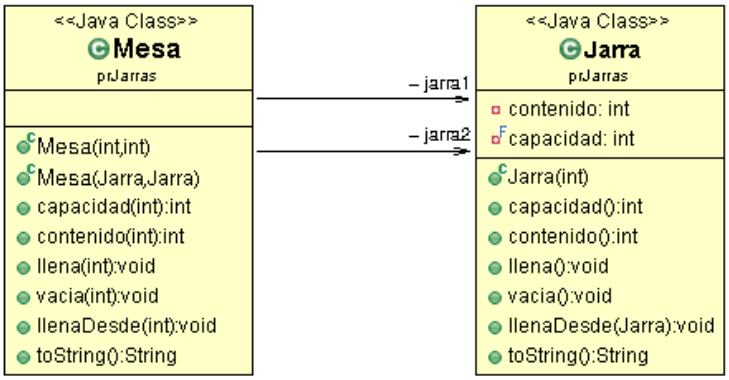

Figura 2: Diagrama de clases UML

Para probar nuestra nueva clase vamos a construir una aplicación (clase distinguida `EjemploUsoMesa1`) en el paquete *por defecto* (*anónimo*):

[2] Nótese que el operador de comparación `==` es útil para comprobar si dos variables referencian al mismo objeto.

- `EjemploUsoMesa1`: se creará una mesa con dos jarras, una con capacidad para 5 litros y otra para 7. Una vez creada, hemos de realizar las operaciones necesarias para dejar en una de las jarras exactamente un único litro de agua.

```
M(J(7, 0), J(5, 5))
M(J(7, 5), J(5, 0))
M(J(7, 5), J(5, 5))
M(J(7, 7), J(5, 3))
M(J(7, 0), J(5, 3))
M(J(7, 3), J(5, 0))
M(J(7, 3), J(5, 5))
M(J(7, 7), J(5, 1))
```