MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN

Clases: Operaciones

10

ÍNDICE

OBJETIVOS

Conocer y aplicar el concepto de clase.

Utilizar constructores.

Conocer los ámbitos de visibilidad.

Conocer el concepto de biblioteca.

Profundizar sobre el concepto de encapsulación.

Conocer el concepto de herencia y distinguir entre sus tipos.

Importar clases en un proyecto.

/ 1. Introducción y contextualización práctica

En este tema vamos a profundizar un poco más en los constructores de una clase, agregando el constructor de copia.

También vamos a hablar sobre la importancia de las bibliotecas y los paquetes en los proyectos de programación, ya que nos permitirán importar código que ha realizado otra persona.

Por último, vamos a hacer una introducción al concepto de herencia, ¿qué es y para qué sirve?, además de los diferentes tipos de herencia que existen y soporta Java.

Todos estos conceptos los iremos poniendo en práctica tanto en este tema como en los sucesivos, así que que el ritmo será lento.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

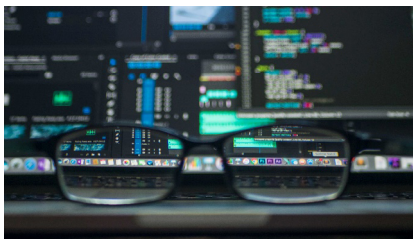

Fig. 1. Visión de programador

Audio Intro. "Las bibliotecas"
https://bit.ly/3emJW5t

/ 2. Constructores II. Constructor de copia

Como hemos visto en los temas anteriores, **los constructores son una parte fundamental en las clases, ya que nos permiten inicializar los objetos de forma correcta para que no se produzcan errores indeseados**. Aparte de los constructores que ya hemos visto, existe uno más, el constructor de copia o constructor de copias.

Este constructor nos ayudará a "clonar" un objeto, es decir, **su función es la de crear un objeto nuevo a partir de otro del mismo tipo con los mismos valores en sus atributos**.

El uso de este constructor viene justificado, ya que si igualamos dos objetos mediante el operador de igualdad (=), no estamos creando un objeto con los mismos atributos que al que igualamos, sino que en realidad, estamos haciendo que el objeto que igualamos apunte[1] a la misma dirección de memoria que el objeto al que lo igualamos.

Esto quiere decir, que cuando igualamos dos objetos del mismo tipo, al estar haciendo que apunten a la misma dirección de memoria, en realidad son el mismo objeto, y si modificamos uno de ellos el otro "también se modificará" por el hecho de apuntar a la misma dirección de memoria.

Este es el problema que nos soluciona el constructor de copia, ya que, gracias a él, vamos a poder crear un objeto independiente del que copiamos, con los mismos valores en sus atributos del objeto con el que lo creamos.

Fig. 2. Creando clases

Audio 1. "Repaso de constructores"
https://bit.ly/304ulgy

2.1. Ejemplo de un constructor de copia

En Java, los constructores tienen que seguir las siguientes reglas:

- **No devuelven ningún tipo de dato**, ni siquiera void.
- **Tienen que ser públicos**, para poder ser utilizados.
- **Tienen que llamarse exactamente igual que la clase a la que pertenecen**.

El constructor de copia también sigue estas normas, lo único que va a cambiar en él, es que va a recibir como parámetro un objeto de la misma clase a la pertenece.

La cabecera de un constructor de copia en Java es la siguiente:

```
public NombreClase ( NombreClase nombreobjeto )
```

Donde NombreClase será el nombre de la clase a la que pertenece el constructor de copia, ya que han de llamarse de la misma forma.

En una clase podremos tener únicamente un constructor de copia.

1. Si quieres saber más de por qué ocurre esto, puedes investigar sobre los punteros.

Un ejemplo de constructor de copia, y cómo invocarlo, lo podemos ver en la siguiente figura.

```
class Persona {
    private String nombre, apellidos;
    private int edad;

    public Persona(String nombre, String apellidos, int edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }

    // Constructor de copia
    public Persona( Persona copia ) {
        nombre = copia.nombre;
        apellidos = copia.apellidos;
        edad = copia.edad;
    }
}

public class EjemploTema6 {

    public static void main(String[] args) {
        Persona personal = new Persona("Pepito", "Pérez", 30);
        Persona persona2 = new Persona(personal);
        // personal y persona2 son dos objetos independientes
        // con los mismos datos
    }
}
```

Fig. 3. Ejemplo de creación y uso de constructor de copia

2.2. Bibliotecas y paquetes en Java

En la gran mayoría de lenguajes de programación que existen hoy en día tenemos el concepto de biblioteca[2] , o library en inglés.

Una biblioteca **es un conjunto de clases que comparten una misma funcionalidad, pudiendo ser encapsuladas en un mismo paquete**.

Las bibliotecas **nos van a permitir llevar la reutilización de código y la ocultación de información a otro nivel,** ya que, cuando utilizamos una biblioteca nos descargaremos un paquete que podremos integrar en nuestro proyecto, como veremos en próximas unidades.

Un punto fuerte de las bibliotecas es que **cualquiera puede crear una y compartirla con el resto del mundo**. Por ejemplo, podemos crear una biblioteca para gestionar el acceso a una base de datos concreta, compilarla y compartirla para que otras personas puedan usarla, resolviendo así un problema concreto, además, ayudando a personas que pueden estar iniciándose en el mundo de la programación, o simplemente gente que necesite esa funcionalidad para sus propios proyectos.

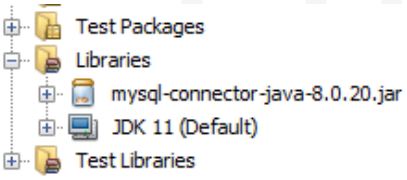

Fig. 4. Bibliotecas importadas en un proyecto de NetBeans

Algunos lenguajes de programación tienen la posibilidad de poder importar bibliotecas hechas en otros lenguajes, aumentando así de una forma enorme su versatilidad.

Vídeo 1. "Crear estructura de paquetes"
https://bit.ly/2WfbraY

2. Puede que en algunas fuentes nos encontremos las bibliotecas como librerías, esto se debe a una mala traducción del inglés library.

2.3. Comando import

Todos los lenguajes de programación que posibilitan el uso de bibliotecas deben de facilitar una forma de hacerlo.

En nuestro caso, el lenguaje de programación Java, nos proporciona el comando import, con el que podremos importar clases de las bibliotecas que usemos, tanto propias como externas.

La sintaxis del comando import es la siguiente:

```
import nombrepaquete.nombreclase;
```

También podremos importar un paquete completo de la siguiente forma:

```
import nombrepaquete.*;
```

Como vemos, mediante el comodín asterisco (*) podremos importar todas las clases que contenga un paquete, aunque lo normal será importar solo las clases que necesitemos, ya que mientras más clases importemos, más código tendrá nuestro proyecto y más tamaño ocupará en disco.

En un proyecto podremos importar tantas bibliotecas, propias o externas, que necesitemos.

Las bibliotecas propias son las que trae incorporadas nuestro JDK, como puede ser la biblioteca System, utilizada para mostrar datos por consola.

Las bibliotecas externas son las que nos descargamos e integramos a nuestros proyectos que están hechas por terceros, como puede ser una biblioteca para el acceso a una base de datos.

2.4. Ejemplo de import en Java

Vamos a ver un ejemplo paso a paso del uso de la importación de clases en un proyecto en Java.

En la siguiente figura, tenemos la estructura de un proyecto Java con varios paquetes y bibliotecas.

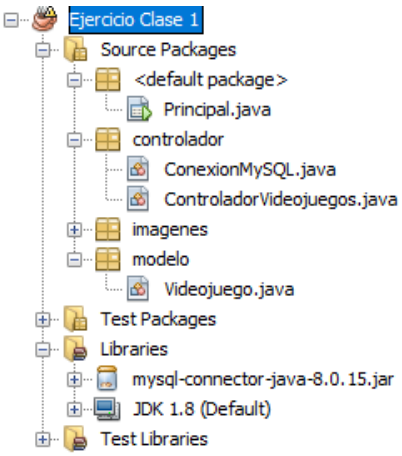

Fig. 5. Proyecto ejemplo import

Los paquetes son controlador, imagenes, modelo y el paquete por defecto.

Dentro de los paquetes tenemos:

- **Paquete modelo**. En este paquete tenemos la clase Videojuego, la cual representará un videojuego en el proyecto, con ciertos atributos.
- **Paquete controlador.** Aquí tenemos las clases ConexionMySQL, la cual nos servirá para gestionar la conexión a una base de datos MySQL, y la clase ControladorVideojuegos, la cual servirá para la gestión de los objetos de tipo Videojuego.
- **Paquete imágenes**. Donde se incluyen las imágenes que usaremos en el proyecto.
- **Paquete por defecto**. En este paquete tenemos la clase principal que ejecuta nuestro proyecto.

Si quisiéramos utilizar, por ejemplo, la clase Videojuego dentro de nuestra clase principal tendríamos que importarla de la siguiente forma:

```
import modelo.Videojuego;
```

2.5. Creación de una biblioteca en Netbeans

NetBeans nos ofrece la posibilidad de crear una biblioteca a partir de nuestras clases, para así poder utilizarla en futuros proyectos.

Para ello tenemos que crear un proyecto para tal fin. Cuando seleccionamos nuevo proyecto, elegiremos la opción **"Java Class Library"**, el cual es el tipo de proyecto para crear una biblioteca.

En la siguiente pantalla, **podremos nombrar a nuestra biblioteca y elegir donde se creará nuestro proyecto**. Por el momento, todo muy parecido a la creación de un proyecto normal y corriente.

Una vez con el proyecto vacío ya listo, **crearemos tantos paquetes como nos hagan falta, y dentro de ellos crearemos las clases que queramos que tenga nuestra biblioteca**.

Como recomendación, antes de crear el proyecto de biblioteca, **sería bueno crear un proyecto normal y corriente y crear la misma estructura de paquetes y clases que tendrá la futura biblioteca**. Con esto podremos crear las clases y probarlas hasta que estas funcionen correctamente.

Una vez tengamos ya montada la estructura de paquetes y clases, previamente testeadas, **solo nos falta crear el fichero de biblioteca**.

Los ficheros de biblioteca en Java son ficheros cuya extensión es .jar, y serán los que podamos importar más adelante.

Para crear este fichero **pulsaremos en el menú Run y seleccionaremos la opción Clean and Build proyect**. Cuando termine de generar la biblioteca tendremos una carpeta llamada build en nuestro proyecto con el fichero .jar de nuestra biblioteca creado y listo para usarse.

Fig. 6. Compartir nuestras bibliotecas facilitará el trabajo de muchos y muchas

2.6. Importar una biblioteca en Netbeans

Ya sabemos qué son las bibliotecas, también, que prácticamente todos los lenguajes de programación de la actualidad las soportan (incluso hechas entre lenguajes de programación distintos), sabemos además cómo crearlas, y cómo importar las clases que contienen para poder utilizarlas, pero **¿cómo podemos importar una biblioteca externa a nuestro proyecto de NetBeans?**

Una vez que tenemos un proyecto creado, **tenemos una carpeta que se llama Libraries, donde están todas las bibliotecas que usamos en dicho proyecto**, ya sean bibliotecas propias o externas.

Para agregar una nueva biblioteca, pulsaremos con el botón derecho sobre la carpeta Libraries y nos aparecerá un menú contextual donde tendremos las siguientes opciones:

- **Add Proyect.**
- **Add Library.**
- **Add JAR/Folder**

Elegiremos la opción **"Add JAR/Folder"**, y se nos abrirá una ventana para poder elegir un fichero .jar, que será la biblioteca que vamos a usar.

Una vez seleccionada la biblioteca, **observamos que nos aparecerá bajo la carpeta Libraries, siendo esto el indicador de que hemos realizado correctamente el proceso de importación**.

A partir de este momento, **cada vez que compilemos nuestro proyecto también se compilará el código de la biblioteca**, y podremos importar todos los elementos que posea, clases, imágenes... mediante la sentencia import.

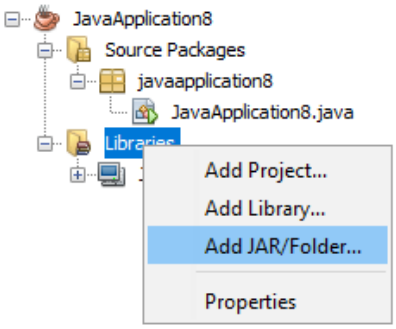

Fig. 7. Agregando una biblioteca

/ 3. Caso práctico 1: "El problema del diseño"

Planteamiento. Pilar y José están creando una clase y ya tienen claro cuáles van a ser los atributos que necesitan, pero lo que no tienen tan claro es cuáles son los constructores que van a necesitar. Durante sus investigaciones han encontrado que existen tres tipos diferentes de constructores, cada uno para un fin.

Nudo. ¿Qué piensas sobre ello? ¿Crees que es necesario crear todos los tipos de constructores en las clases? ¿O por el contrario crees que solo creando uno de ellos tendremos suficiente?

Desenlace. Cuando estamos desarrollando una clase lo que hacemos en realidad es diseñar una estructura de datos que nos va a resolver un problema. El diseño de las estructuras de datos es de vital importancia, ya que gracias a ello podremos sortear un obstáculo difícil de una forma fácil, o podremos convertir un problema fácil en un verdadero dolor de cabeza. Con el tema de los constructores no va a ser menos, ya que tenemos que elegir cuáles van a ser los apropiados para el diseño de nuestra clase, lo cual puede llegar a ser una tarea complicada. Cada uno de ellos va a resolver un problema distinto, y lo más recomendable sería crear como mínimo el constructor por defecto y el constructor con parámetros con todos los datos de nuestra clase, dejando el constructor de copia para casos muy especiales. No obstante, no olvides que siempre podremos modificar nuestras clases y agregar o quitar el código que deseemos.


Fig. 8. Diseñando

/ 4. Ámbitos de visibilidad en las bibliotecas

Como ya vimos en unidades pasadas, concretamente en la unidad 4, existen diferentes tipos de ámbitos de visibilidad dentro del uso de las clases:

Modificador	public	protected	private	Defecto
Acceso desde la propia clase	SI	SI	SI	SI
Acceso desde otras clases del mismo paquete	SI	SI	NO	SI
Acceso desde una subclase en el mismo paquete	SI	SI	NO	SI
Acceso desde subclases en otros paquetes	SI	SI	NO	NO
Acceso desde otras clases en otros paquetes	SI	NO	NO	NO

Tabla 1. Recordatorio de ámbitos de visibilidad

Como ya sabemos, las bibliotecas están compuestas de clases, y en las clases tendremos atributos a los que debemos aplicar un ámbito de visibilidad obligatoriamente.

Como no podrían ser menos, **las bibliotecas también se ven afectadas por los ámbitos de visibilidad**. En este caso, podremos crear clases privadas, las cuales podremos utilizar en nuestra biblioteca sin ningún problema, pero que no se podrán utilizar fuera de ella, es más, desde fuera ni siquiera se sabrá que existen.

Esto puede ser muy útil si necesitamos de alguna clase intermedia para poder llegar a solucionar un problema, pero, al ser intermedia, no nos interesa que sea visible desde fuera.

También puede darse el caso de que nos interese declarar alguna variable de clase pública, para que pueda ser accedida desde fuera, en este caso, podremos utilizar esa variable pública de una clase de la biblioteca.

Vídeo 2. "Recordatorio de ámbitos de visibilidad"
https://bit.ly/2DEjwj9

/ 5. Encapsulamiento en las bibliotecas

Como ya vimos en la unidad 4, uno de los pilares fundamentales de la programación dirigida a objetos es el encapsulamiento.

Este **nos proporciona cohesión en los datos que forman un objeto**, pero, hablando de bibliotecas, ¿se siguen cumpliendo este pilar de la programación dirigida a objetos?

Cuando nos centramos en las bibliotecas, podemos ver claramente que sí cumplen con el principio de encapsulamiento, y además de éste, cumplen también con el de ocultación de la información, ya que estos dos principios van de la mano.

Empecemos por el principio de encapsulamiento. Las bibliotecas lo cumplen ya que todos los paquetes y clases que las forman deberán estar organizados de tal forma que sean coherentes y que cumplan una funcionalidad común.

Si las bibliotecas no estuvieran compuestas por paquetes, que a su vez contienen clases con funcionalidades con el mismo propósito, **el encapsulamiento de éstas sería mucho más débil**, ya que estarían todas mezcladas sin ningún sentido.

En cuanto a la ocultación de información, **las bibliotecas lo cumplen a la perfección, ya que todas las clases que las componen se pueden utilizar, pero no se puede ver su contenido**, así que todo lo que haya sido declarado con un ámbito de visibilidad privado será totalmente opaco a quien esté utilizando dicha biblioteca.

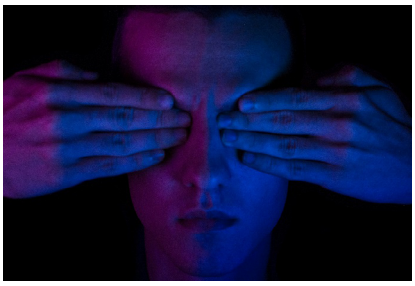

Fig. 9. Ocultación de la información

/ 6. Caso práctico 2: "Organización del código"

Planteamiento: Pilar y José acaban de realizar un ejercicio bastante largo y en el que están involucradas muchas clases, de las cuales unas están relacionas entre sí y otras no. El caso es que según lo que han aprendido, se disponen a organizar el código del proyecto, puesto que con tanta clase sin organizar es bastante lioso.

Nudo: ¿Cómo podrían organizar el código del proyecto? ¿Utilizando paquetes o utilizando bibliotecas?

Desenlace: Siempre que en un programa tengamos muchas clases, y, por lo tanto, muchos ficheros de código, lo más sensato será organizar el código para hacerlo mucho más fácil de leer, entender y, en su caso, modificar. Para ello podemos optar por dos opciones, crear paquetes o crear bibliotecas.

La opción de crear paquetes sería la más fácil y más recomendada cuando las clases no se van a compartir con terceras personas, mientras que, si las vamos a compartir con terceros, lo más lógico sería crear una biblioteca por paquete que creemos, para que así, solo tengamos que compartir el fichero creado con toda la funcionalidad ya operativa.

Esto no quiere decir que no se puedan crear bibliotecas para nuestros proyectos, la idea es que no se pueda modificar el código en una biblioteca, y si necesitásemos hacerlo, tendríamos que volver a recompilarla por completo.

Fig. 10. La organización es la clave

/ 7. Herencia

Otro de los pilares de la programación dirigida a objetos es la herencia. Cualquier lenguaje que soporte la PDO soportará también la herencia.

El concepto de herencia en programación es muy parecido al biológico, básicamente en éste, un ser vivo, al reproducirse, transmitirá toda su información genética a su descendencia, de modo que esta habrá heredado dicha información.

Cuando nos referimos en programación al concepto de herencia nos estamos refiriendo a **la capacidad de una clase de heredar de otra** (que se llamará clase padre o superclase) todo su comportamiento.

El comportamiento al que se refiere son los atributos, con cualquier ámbito y a todos sus métodos, incluidos constructores, métodos get, métodos set, el método toString y todos los métodos que tenga la clase padre.

Este comportamiento fomenta enormemente la reutilización de código, ya que, por el mero hecho de heredar una clase de otra, se dispone de toda la información de la clase padre sin tener que escribir ni una línea de código.

Si una clase B hereda de una clase A, entonces diremos que B es un tipo específico de A.

Hay que tener cuidado, ya que, si la jerarquía de clases es demasiado compleja, podremos tener muchos problemas a la hora de comprenderla y trabajar con ella.

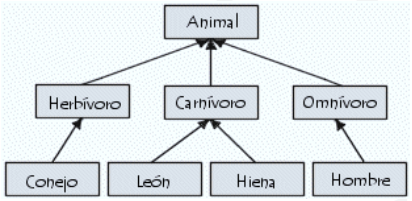

Fig. 11. Ejemplo de herencia simple

7.1. Tipos de herencia

En programación orientada a objetos existen dos tipos de herencia:

- **Herencia simple.**
- **Herencia múltiple**.

La herencia simple **es aquella en la que una clase únicamente puede heredar de otra**. Es el método de herencia más simple y fácil de implementar. Algunos de los lenguajes de programación que únicamente admiten herencia simple son: C#, Java y Ada.

La herencia múltiple **es aquella en la que una clase puede heredar de varias clases**, es decir, puede tener varios padres, heredando de esta forma todas las variables de clase y métodos de todos y cada uno de sus padres.

El hecho de poder heredar de varias clases ya trae un gran inconveniente de por sí, y es el hecho de que los métodos y las variables de clase pueden entrar en conflicto entre ellos, ya que podrían llamarse igual en una clase padre y en otra.

Debido a este problema, que **es mucho más grave de lo que parece, muchos lenguajes de programación decidieron no soportar la herencia múltiple**, siendo éstos la gran mayoría de lenguajes de programación orientados a objetos.

Algunos lenguajes de programación que soportan la herencia múltiple son: Python y C++.

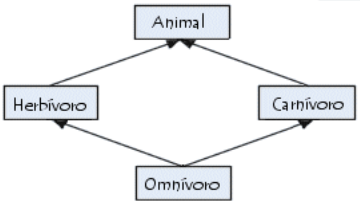

Fig. 12. Ejemplo de herencia múltiple

/ 8. Resumen y resolución del caso práctico de la unidad

A lo largo de este tema, hemos visto el concepto de biblioteca y la gran importancia que tiene dentro del mundo de desarrollo de software.

Hemos podido comprobar que las bibliotecas cumplen con los pilares de la programación dirigida a objetos, el encapsulamiento y la ocultación de información.

También hemos aprendido cómo podemos crearlas en NetBeans y cómo podemos importar una biblioteca a uno de nuestros proyectos.

Por último, hemos hecho una breve introducción al concepto de herencia y sus tipos, que veremos con mucho más detenimiento en unidades futuras.

Resolución del caso práctico de la unidad

Cuando estamos aprendiendo a programar, una cosa que nos suele chocar mucho es el concepto de biblioteca.

Todos los lenguajes de programación que existen hoy en día pueden hacer uso de las bibliotecas, y es que gracias a ellas, los desarrolladores pueden resolver un problema y compartirlo con los demás.

Es gracias a esto que las bibliotecas son una herramienta poderosísima en cuanto a la resolución de problemas se refiere y el hecho de empezar a crear alguna en nuestros primeros pasos hará que nuestra visión sobre este concepto se aclare. Por tanto, no hay lugar a dudas, de que sería altamente recomendable que nuestros amigos Pilar y José se pusieran manos a la obra en el desarrollo de una biblioteca para su proyecto.

Fig. 13. Biblioteca

/ 9. Bibliografía

González, J. D. M. (2019, septiembre 15). Librerías o Bibliotecas. Recuperado 19 de mayo de 2020, de https://www.programarya.com/Cursos/Java/Librerias

Hm, S. (2015, noviembre 11). Creación de Librerías en Java. Recuperado 19 de mayo de 2020, de https://salvadorhm.blogspot.com/2015/11/creacion-de-librerias-en-java.html