MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN

Utilización avanzada de clases I. Composición y herencia

12

ÍNDICE

OBJETIVOS

Conocer la diferencia entre composición y herencia

Conocer la diferencia entre herencia simple y múltiple

Realizar programas con clases heredadas

Utilizar la sobreescritura de métodos

Aplicar el concepto de herencia a constructores

/ 1. Introducción y contextualización práctica

En esta unidad, hablaremos sobre **la herencia, un concepto bastante importante en la programación dirigida a objetos.**

Además de la herencia, vamos a tratar el concepto de composición, otro tipo de relación entre clases.

Dentro de la herencia vamos a ver los diferentes tipos que existen y cómo influyen en distintas partes de las partes de las clases estudiadas anteriormente.

Por último, aprenderemos cómo podemos sobrescribir un método de una clase heredada para que tenga un comportamiento diferente.

Todos estos conceptos los iremos poniendo en práctica tanto en éste, como en todos los temas restantes, así que no te preocupes, que tenemos mucho tiempo por delante.

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

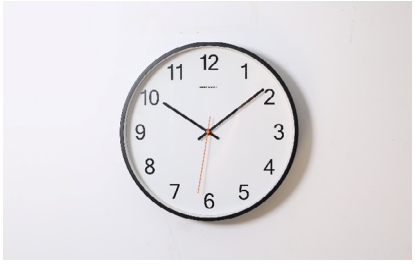

Fig. 1. Tiempo de aprendizaje.

Audio Intro. "Ahorro de código".
https://bit.ly/3g5581B

/ 2. Relaciones entre clases

Ya vimos en unidades pasadas el concepto de clase que, si recordamos, definíamos como un mecanismo por el que podíamos representar elementos del mundo real, en nuestros programas, para posteriormente, poder crear objetos de ellas, es decir, una clase es una "plantilla" que nos permitirá definir objetos.

El siguiente paso con las clases es **establecer relaciones entre ellas,** para que así, todo nuestro proyecto esté conectado y tenga un diseño sólido y lógico, aunque puede ocurrir que no haya relación entre clases en algunos casos.

Podemos distinguir dos grandes casos principales en las relaciones entre clases, la composición y la especialización.

Estos dos grandes bloques, engloban al resto de relaciones entre clases, pudiendo distinguir cuatro posibles casos:

- **Relación de composición:** Este tipo de relación se dará cuando una clase no tenga sentido sin otra. La veremos más adelante.
- **Relación de herencia:** Esta se dará cuando una clase necesite implementar toda la funcionalidad de otra. Profundizaremos también sobre ella a continuación.
- **Relación de clientela:** Este tipo de relación se dará cuando una clase utiliza a otra en el paso de parámetros de los métodos. Este tipo de relación la estamos utilizando desde que empezamos a trabajar con las clases, ya que vimos que a los métodos podíamos pasarle objetos, y que éstos se pasaban mediante el paso por referencia.
- **Relación de anidamiento:** En este caso, la relación tendrá lugar cuando una clase defina a otra en su interior, lo que se conoce como anidamiento de clases o clase interna.

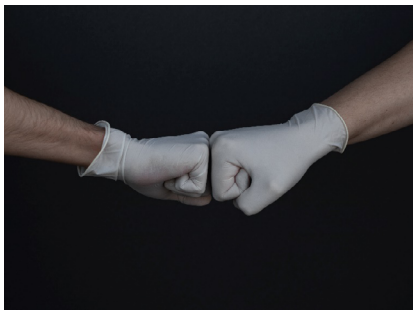

Fig. 2. La relación es cooperación.

Veamos a continuación la relación de composición y herencia.

2.1. Composición

La relación de composición se da cuando **una clase declara variables en su interior que son objetos de otra clase**, de forma que una instancia de esa clase va a contener en su interior, varias instancias de otras clases.

Cuando utilizamos la relación de composición hay que tener mucho cuidado a la hora de realizar los constructores.

Como ya vimos en unidades anteriores, los constructores son los encargados de instanciar de forma correcta los objetos, para que no haya futuros fallos.

Cuando tenemos objetos como variables de una clase debemos **instanciar obligatoriamente esos objetos mediante sus constructores** respectivos en los constructores de la clase.

Con esto queremos decir que:

- Si tenemos un constructor por defecto en nuestra nueva clase, deberemos llamar a los constructores por defecto de esos objetos.
- Si tenemos un constructor con parámetros en nuestra clase, deberemos llamar a los constructores con parámetros de esos objetos.

También puede ocurrir que en un constructor de nuestra clase llamemos a otro constructor de los objetos, por ejemplo, en el constructor con parámetros de la clase llamar al constructor por defecto de un objeto.

La relación de composición se puede dar en cualquier lenguaje de programación que soporte la orientación a objetos. Un ejemplo de composición puede ser que tengamos una clase *Persona,* dentro de esta clase tendremos atributos para guardar su nombre, apellidos y DNI, los cuales serán *String*, y, además, tendremos que almacenar su fecha de nacimiento, lo cual podemos hacer mediante un objeto de la clase *GregorianCalendar.*

```
public class Persona
{
    private String nombre, apellidos, DNI;
    // Ahora definimos la relación de composición
    private GregorianCalendar fechanacimiento;
}
```

Código 1. Ejemplo de uso de composición.

2.2. Herencia

La herencia es uno de los pilares de la programación orientada a objetos. Gracias a ella vamos a poder reutilizar nuestro código de una manera no vista hasta ahora, reduciendo así la complejidad de nuestros proyectos.

Utilizando la herencia vamos a poder crear clases que se apoyan en otras, creando una jerarquía de clases y evitando así la recodificación y prueba de muchos métodos.

Aquí tenemos que distinguir ahora dos conceptos muy importantes:

- **Clase padre o superclase:** Esta es la clase de la que se ha heredado.
- **Clase hija o subclase:** Esta es la clase que hereda de otra.

Cuando una clase hereda de otra se dice que ésta hereda el comportamiento de su clase padre. Esto se refiere a que la clase que hereda, por el mero hecho de heredar, ya tiene todos los atributos y métodos de la clase padre.

Cuando creemos un objeto de la clase hija podremos ver que se pueden llamar los métodos de la clase padre, teniendo el mismo comportamiento que tenían ahí.

Tenemos que tener cuidado ya que cuando heredemos de una clase, no podremos ver su parte privada. Si quisiéramos acceder a esa parte deberemos hacerlo mediante métodos, como hemos estado haciendo hasta ahora normalmente. Las ventajas de la herencia son muchas, entre ellas tenemos:

- Reduce la cantidad de código, ya que no hace falta volver a implementar los métodos heredados.
- No será necesario testear los métodos heredados, ya que se testearon cuando se hicieron en la clase padre.

Cuando una clase hereda de otra decimos que “es un” de la clase heredada, por ejemplo: la clase Perro es un Animal, siendo Animal otra clase.La clave para saber si hay que utilizar la herencia es cuando la clase heredada añade una funcionalidad extra al comportamiento de la clase padre.

/ 3. Caso práctico 1: “¿Cómo accedemos a los métodos en las composiciones?”

Planteamiento: Pilar y José están repasando el concepto de composición. Se lo explican entre ellos y lo simplifican argumentando que se trata simplemente de declarar como una variable de clase, un objeto, nada más.

Pilar, extraña y pensativa, cae en un detalle que no habían pensado, “¿qué ocurre si queremos utilizar algún método de ese objeto?, al fin y al cabo, se ha de declarar como privado, entonces no podremos acceder a él”. José le mira y se queda pensativo.

Nudo: ¿Qué piensas sobre ello? ¿Crees que hay alguna forma de poder llamar a los métodos de un objeto cuando tenemos una composición? ¿Habrá que declarar a ese objeto como público?

Desenlace: Ya sabemos que cuando utilicemos una relación de composición, lo que tenemos en realidad es un objeto como una variable de clase, pero, como variable de clase que es, hay que declararlo como privado, ya que declararlo como público va a incumplir los principios de encapsulación y ocultación de la información, aunque las variables de dicho objeto sean privadas, por lo que esa solución no es factible.

Si pensamos un poco, la solución es muy sencilla, y es que lo que hace falta declarar en esta situación es un "puente" entre la clase que implementa la composición, y el objeto que hemos declarado. Esto básicamente consiste en crear un método en la clase que llame al método que queremos utilizar del objeto, y para evitar confusiones, podremos ponerle el mismo nombre, los mismos parámetros, y el mismo valor devuelto, obteniendo así, un método con la funcionalidad desea y una única línea de código, la de llamar al método del objeto deseado.

```
public class Alumno extends Persona
{
    // Atributos necesarios

    // Método puente

    public int getEdad()
    {
        return super.getEdad();
    }
}
```

Código 2. Ejemplo de método puente.

/ 4. Herencia múltiple

Hay varios tipos de herencia, y entre ellos, podemos destacar la herencia múltiple. En la herencia múltiple **una clase va a poder heredar de varias clases**, es decir, va a tener varias clases padre, y heredará el comportamiento de todas y cada una de ellas.

Hay varios lenguajes de programación que soportan herencia múltiple, como por ejemplo C++, Python, Perl... Sin embargo, Java no es uno de ellos. Este tipo de herencia es muy compleja, ya que puede contener ambigüedades en su definición. Esto puede venir motivado porque las herencias no se hayan definido en el orden correcto.

Un ejemplo de ambigüedades puede ser que haya dos clases de las que se hereda que tengan un atributo o un método que se llamen igual, entonces, si queremos llamar a ese atributo o método, ¿a cuál nos referimos?

La solución a este problema debe aportarla cada compilador de cada lenguaje de programación, es decir, probablemente C++ no lo solucionará de la misma forma que lo haga Python.

Concretamente, en C++, para solucionar este problema el compilador nos obliga a indiciar el nombre de la clase padre a la que pertenece el atributo o método que queremos utilizar. Como vemos, el uso de la herencia múltiple requiere de un comportamiento que no se va a obtener ni tratar a lo largo de estas unidades, por lo cual, y como en Java no se implementa la herencia múltiple, no lo vamos a utilizar.

/ 5. Herencia simple

En nuestro lenguaje de programación, Java, el tipo de herencia que vamos a poder utilizar es la herencia simple.

En la herencia simple **una clase únicamente va a poder heredar de otra**, es decir, solo va a tener una clase padre.

Su funcionamiento es en base a la relación entre clases que hemos visto anteriormente, es decir, cuando una clase hereda de otra adquiere todo su comportamiento, tanto variables como métodos, incluyéndose también toda variable y método privado que haya en la clase padre.

El concepto de herencia es muy simple, pero muy potente a la vez, ya que si conseguimos aprender a utilizarlo bien nos simplificará en gran medida nuestros programas.

Ya sabemos que Java es un lenguaje puramente orientado a objetos, es decir, todo son clases, y estas clases deben estar relacionadas entre sí, como es de esperar. Esta relación la tenemos con la clase *Object*. Toda clase en Java, desde las internas, hasta una que podamos crear nosotros mismos, heredan de *Object*, es decir, que todo son objetos.

Para representar en Java que una clase hereda de otra utilizamos la palabra reservada *extends*, que va en la declaración de la clase hija, seguida del nombre de la clase de la que va a heredar. Al ser herencia simple con lo que vamos a trabajar, no podremos poner más de un *extends* en la declaración de una clase Sí que podemos sin ningún tipo de limitación, hacer que una clase herede de otra, que a su vez hereda de otra, por ejemplo:

A hereda de B y B hereda de A, aquí tendremos que A es un B, B es un C y por lo tanto A es un C.

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

class Profesor extends Persona
{
    private int idprofesor;
}
```

Código 3. Profesor hereda de persona.

/ 6. Atributos heredados

Como ya hemos comentado, cuando una clase hereda de otra, hereda sus atributos, tanto públicos como privados. Es muy importante recalcar aquí que, **si una clase hereda de otra, ésta no podrá acceder a la parte privada de la clase padre** (tanto atributos como métodos), ya que es privada. Sin embargo, sí que podrá acceder a la parte pública, y si hay algún atributo definido como público, lo podrá usar sin ninguna restricción.

En el caso de que, por cualquier circunstancia, se necesite acceder a la parte privada de la clase, **deberemos crear un método que nos lo permita**, con visibilidad pública, como es de esperar.Existe una forma, por tanto, con la que sí podremos acceder a la parte privada de una clase de la que se hereda, y es con el modificador de visibilidad *protected*.

Cuando declaramos una variable con visibilidad *protected* dentro de una clase, estamos indicando que cualquier clase que herede de ella podrá acceder a dicha propiedad, es decir, como si fuese pública a su vista, mientras que cualquier clase que no herede de ella no podrá utilizarla, como si fuese privada a su vista.

Hay que tener mucho cuidado con *protected*, ya que las clases que estén en el mismo paquete también podrán acceder a esa parte privada, así que tendremos que crear un paquete para almacenar la jerarquía de clases que hayamos creado.

El modificador *protected* se puede aplicar tanto a atributos como a métodos. A continuación, completamos la tabla de visibilidad que ya vimos en la unidad 4 con el modificador *protected*.

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

Tabla 1. Modificadores de visibilidad.

Audio 1. "Recomendación con protected".
https://bit.ly/3jJYiks

/ 7. Constructores y herencia

Cuando estamos utilizando la herencia y heredamos de una clase, ¿qué ocurre con los constructores? Los **constructores se heredan de forma indirecta**, es decir, se heredan, pero no se pueden utilizar de forma directa con el operador *new*, sino que se deben de crear los constructores propios de la clase hija.

Como no podemos acceder a la parte privada de la clase para poder iniciarla al valor oportuno, debemos poder hacerlo de alguna forma, esto se consigue con la instrucción *super*. La instrucción *super* permite **llamar a un método de la clase padre**, y concretamente, la vamos a utilizar para los constructores.

Al definir el constructor de la clase hija, lo primero que deberemos hacer de forma obligatoria es llamar al constructor de la clase padre utilizando *super*, y después de ello ya podremos escribir el código que sea necesario. En caso de que no llamemos a *super* en primer lugar el compilador nos mostrará un fallo. Esta llamada la podremos hacer en cualquiera de los constructores que hemos estudiando en unidades anteriores, pudiendo llamar a *super* con los parámetros que tenga el constructor al que queramos llamar.

Este método es un claro ejemplo de la reutilización de código que nos proporciona la herencia.

```
class Profesor extends Persona
{
    private int idprofesor;

    public Profesor()
    {
        super();
        idprofesor = 0;
    }
}
```

Código 4. Constructor por defecto de profesor.

Vídeo 1. "Creación de constructores de una clase heredada".
https://bit.ly/2OWuums

/ 8. Modificadores y herencia

En unidades anteriores, hemos visto que las clases tenían dos tipos de métodos que nos permitían obtener y modificar los valores de las variables privadas:

- **Observadores:** Estos métodos permiten observar el valor de una variable privada. Son los métodos *get*.
- **Modificadores:** Estos métodos permiten modificar el valor de una variable privada. Son los métodos set.

Al heredar de una clase ya sabemos que obtenemos todo su comportamiento, métodos incluidos, pero ¿qué ocurre con los métodos observadores y modificadores?

Estos métodos también se heredan, como métodos que son, así que en la clase hija no habría que volver a implementarlos.

Hay que tener cuidado porque los métodos observadores y modificadores de las variables nuevas de la clase hija, sí que hay que implementarlos, y cuando vayamos a usar un objeto de la clase, veremos que podremos invocar tanto a los métodos observadores y modificadores de las variables de la clase, como a los de su clase padre.

Con esto, observamos nuevamente que la herencia nos reduce de forma significativa el código que debemos escribir en nuestras clases. Una mala práctica sería el volver a definir los métodos observadores y modificadores de la clase padre en la clase hija y llamarlos con *super*.

Esto no tendría ningún sentido, ya que no hará más que complicar la implementación de la clase de forma innecesaria.

Fig. 3. El ahorro de código es importante.

/ 9. Sobreescritura de métodos

Al heredar de una clase, sabemos que heredamos todos sus métodos que funcionarán de la misma forma que en la clase padre, pero ¿y si por algún motivo quisiéramos cambiar el comportamiento, o simplemente añadirles alguna nueva funcionalidad a esos métodos?

Esto es posible hacerlo gracias a la sobreescritura de métodos. Esta funcionalidad va ligada a la herencia, ya que, si por cualquier motivo el funcionamiento de un método en una clase hija varía, se pueda implementar.

En una clase **pueden ser sobrescritos todos los métodos salvo los constructores**, ya que éstos, como ya hemos visto, han de llamarse dentro de los constructores de la clase hija.

Un método puede ser sobrescrito en una clase una vez, lo que no impide que, si otra clase hereda de la clase que tiene el método sobrescrito, ésta pueda volver a sobrescribirlo sin problema.

Para poder sobrescribir un método, como es lógico, primero **tiene que estar implementado en la clase padre**, posteriormente, debemos **volver a crearlo con el mismo nombre**, mismo tipo devuelto, y mismos parámetros en la clase hija, solo que ahora tendremos que indicar la siguiente sentencia en la parte superior: *@override*.

Con esta sentencia, estamos indicando al compilador que ese método está sobrescrito y que cuando se llame desde un objeto de la clase que lo ha sobrescrito, deberá tener esa funcionalidad, no la de la clase padre.

Si lo que queremos es agregarle una nueva funcionalidad al método, lo que tenemos que hacer es llamar al mismo método de la clase padre mediante: *super.*nombredelmetodo.

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

    public Persona()
    {...6 lines }

    public void saludar()
    {
        System.out.println("Hola me llamo " + nombre);
    }
}

class Profesor extends Persona
{
    private int idprofesor;

    public Profesor()
    {...4 lines }

    @Override
    public void saludar()
    {
        System.out.println("Soy un profesor");
        super.saludar();
    }
}
```

Código 5. Ejemplo de sobreescritura.

Vídeo 2. "Ejemplo de sobreescritura de métodos".

https://bit.ly/31596RR

/ 10. Caso práctico 2: "¿Hasta dónde llegar con la herencia?"

Planteamiento: Pilar y José están realizando un ejercicio en el que tienen que aplicar herencia sobre varias clases.

José le dice a Pilar que ya cree que domina el concepto de herencia, que en realidad no es tan complicado. Pilar le pide a José que le enseñe un momento su código, ya que ella sigue un poco perdida con esto de la herencia.

Cuando Pilar está investigando el código de José se da cuenta de una cosa, José ha creado varias clases que heredan de otra pero que no tienen funcionalidad ninguna, es decir, no tienen nuevos atributos ni nuevos métodos, solo heredan.

Nudo: ¿Qué piensas al respecto? ¿Crees que una clase que hereda de otra pero que no aporta funcionalidad ninguna tiene sentido en un proyecto? ¿O por el contrario piensas como José y crees que si tiene sentido?

Desenlace: Cuando estamos aprendiendo el concepto de herencia podemos caer en este fallo.

Si repasamos el concepto de herencia visto anteriormente, una clase tiene sentido que herede de otra cuando agrega alguna funcionalidad nueva, tanto atributos como métodos. Se decía entonces que ésta es un tipo de la clase que hereda.

Si por el contrario nos sucede como a José y creamos clases que heredan de otras sin que aporten ninguna nueva funcionalidad, no estamos aplicando bien el concepto de herencia, ya que esas clases no tienen un objetivo propio dentro del programa.

También puede ser que no las hayamos definido bien y se nos haya olvidado algún detalle para que les proporcione algún tipo de funcionalidad.

En cualquier caso, si tenemos una clase que hereda de otra y no aporta ninguna funcionalidad, lo único que podemos hacer con ella es eliminarla.

/ 11. Clases y métodos finales

Puede ocurrir que, por ciertos motivos, no queramos que se pueda heredar de una clase, esto es posible gracias a las clases finales.

Recordemos de unidades anteriores que el calificador final, al ser aplicado a una variable, hacía que ésta fuese constante, es decir, que no se pueda modificar su valor durante toda la ejecución del programa.

Cuando declaramos una **clase** y declaramos escribiendo final delante de *class,* en realidad estamos declarando una clase final, lo que quiere decir que **no se podrán crear clases que hereden de ella,** por lo que la jerarquía a la que pertenezca terminará con ella.

La sintaxis de una clase final es:

```
[public / private] final NombreClase [extends ClasePadre]
```

Otro elemento que podemos declarar como *final* son los métodos.

Cuando declaramos un **método** final estamos haciendo que **no se pueda sobrescribir,** y en caso de hacerlo se producirá un error de compilación en el programa.

La sintaxis de un método final es:

```
[public / private] [tipo devuelto / void] final nombreMetodo(
[parámetros])
```

El modificador *final,* además de poder aparecer en variables, clases y métodos, podrá aparecer en un **parámetro** de un método, haciendo que dicho parámetro **sea constante** y no se pueda modificar su valor.

La clase *String* que tanto utilizamos, está declarada como final en su implementación.

```
public final class Cuadrado {
    private double lado;

    public Cuadrado()
    {
        lado = 0;
    }

    public final double area()
    {
        return Math.pow(lado, 2.0);
    }
}
```

Código 6. Clase String

/ 12. Herencia en otros lenguajes

Cualquier lenguaje de programación que soporte la orientación a objetos podrá hacer uso de la herencia.

Como ya sabemos, no solo existe el lenguaje de programación Java, hay muchos más, y muy interesantes.

Vamos a ver algún ejemplo de herencia en otros lenguajes de programación:

Python: El lenguaje de programación Python es muy popular, debido a su sencillez y a su gran potencia. Este lenguaje también **soporta la orientación a objetos**, por lo que se pueden crear clases y aplicar la herencia.

Para crear una clase en Python usamos la palabra class (igual que en Java), pero para hacer que herede de otra debemos indicar el nombre la clase padre entre paréntesis:

```
class Alumno (Persona):
```

De esta forma estamos creando la clase Alumno que hereda de la clase Persona.

Python soporta herencia múltiple, por lo que para hacer que herede de más de una clase tendremos que separarlas mediante comas dentro de los paréntesis.

C++: El lenguaje de programación C++ es un lenguaje de programación muy conocido, debido a su uso en una gran cantidad de aplicaciones.

Para crear una clase en C++ volvemos a utilizar la palabra class pero para hacer que herede utilizamos el operador dos puntos (:).

```
class Alumno : public Persona
```

Como vemos, la herencia es aplicable a distintos lenguajes de programación, lo único que cambiará entre ellos en su sintaxis.

/ 13. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad, hemos aprendido cómo se pueden **relacionar las clases.**

Una clase se puede relacionar con otra simplemente declarando como variable dentro de ella un objeto de otra clase, es decir, la **composición.**

También hemos estudiado el concepto de **herencia**, uno de los pilares de la programación dirigida a objetos, así como los diferentes tipos de herencia que existen, destacando que en Java solo se puede usar la herencia simple.

Otro concepto nuevo que hemos expuesto a lo largo de esta unidad han sido los datos de tipo ***protected*.** Un aspecto en el que también hemos podido profundizar, ha sido el **comportamiento de los atributos y métodos d**e las clases cuando heredan, pudiendo incluso llegar a **sobrescribirlos.**

Por último, hemos comprobado qué son las **clases y métodos finales** y hemos dado un breve vistazo a la herencia en otros lenguajes de programación.

Resolución del caso práctico de la unidad

Analizando el programa con el que se encuentran lidiando Pilar y José podemos considerar varios aspectos.

Por un lado, tienen que crear muchas clases, ya que hay muchos animales en el zoológico, y como es lógico, cada clase tendrá sus atributos, constructores y métodos.

Por otro lado, muchas clases comparten código, es decir, hay código repetido.

Cuando tenemos estos indicios lo más probable es que necesitemos utilizar la herencia. Esta consideración es trivial, ya que en un zoológico habrá animales que compartan propiedades y atributos, como pueden ser mamíferos, reptiles, aves...

Utilizando la herencia ahorraremos mucho tiempo ya que al heredar ya tendremos toda la funcionalidad de la clase padre, así que, no tendremos que reescribir todo el código repetido.

Fig. 4. Con la herencia aprovechamos elementos comunes

/ 14. Bibliografía

Guille, E. (2006, abril 3). *Composición en JAVA*. Recuperado 24 de mayo de 2020, de http://www.ingenieriasystems.com/2016/03/composicion-en-java.html

Relaciones entre Clases. (s. f.). Recuperado 24 de mayo de 2020, de http://personales.upv.es/rmartin/cursoJava/Java/OO/RelacionesClases.htm

Colaboradores de Wikipedia. (2020, abril 17). *Herencia (informática)* - Wikipedia, la enciclopedia libre. Recuperado 24 de mayo de 2020, de https://es.wikipedia.org/wiki/Herencia_(inform%C3%A1tica)

Colaboradores de Wikipedia. (2016, septiembre 25). *Herencia múltiple* - Wikipedia, la enciclopedia libre. Recuperado 24 de mayo de 2020, de https://es.wikipedia.org/wiki/Herencia_m%C3%BAltiple