Licenciatura en Sistemas - Orientación a Objetos II – 2020

Equipo docente: Mg. María Alejandra Vranic
Lic. Romina Mansilla
Lic. Gustavo Siciliano
Lic. Ezequiel Scordamaglia

Herencia

Teoria (Parte 1)

Herencia simple, herencia múltiple e Interfaces

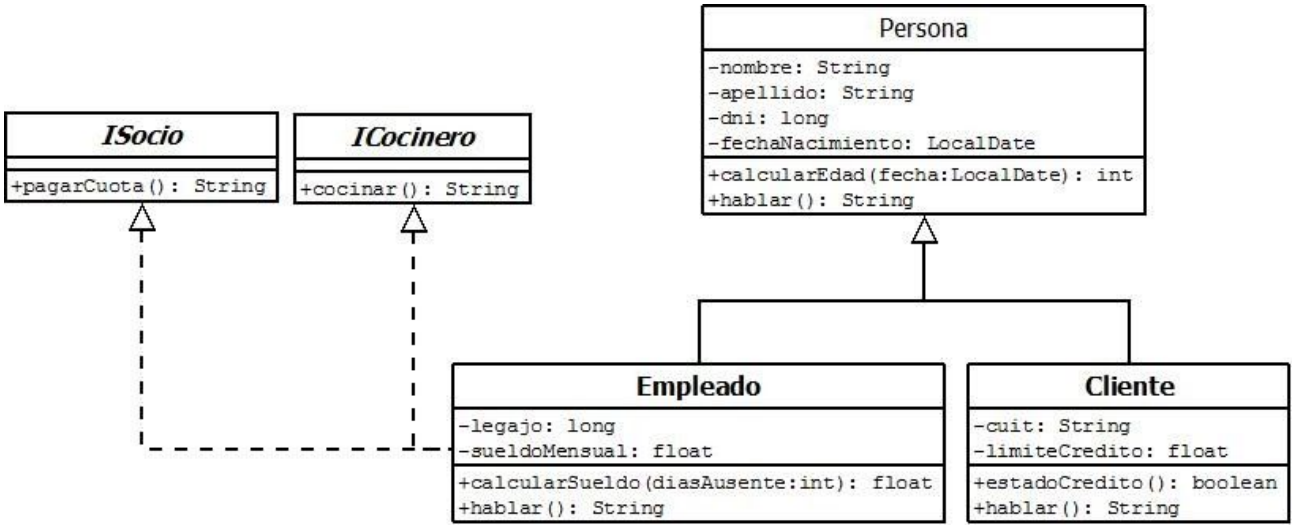
Si llevamos el concepto de la clase abstracta al extremo, nos encontraremos con una clase abstracta que declara todos sus métodos abstractos sin implementación alguna. ¿Para qué sirve?

En primer lugar, sirve para definir un contrato que toda subclase de la misma deberá obedecer, es decir: la interfaz de la clase. Al definir el contrato, también estamos definiendo al tipo: Un Animal es Animal porque come y respira (hace otras cosas también, pero no nos interesan); es difícil que tengamos instancias de Animal (¿Para qué?) pero sí puede ocurrir que queramos tratar a la Paloma o al Pez como Animales. Por ejemplo, podría tener una lista de Animales y debería recorrerla haciendo respirar a cada uno; en ese caso no me importa de qué animal específico se trate. Esto es muy importante, porque nos permite diseñar software que se pueda extender de manera modular especificando la interfaz (el contrato) que debe respetar cada módulo.

Dijimos que en Java la herencia es simple, que sólo se puede heredar de una clase. Hay otros lenguajes de programación que permiten herencia de múltiples superclases (C++, Python, CLOS) y cada uno ofrece distintos medios para solucionar lo que se denomina el "problema del diamante" (diamond problem). Básicamente el problema es el siguiente: Si heredamos de dos clases que implementan un método con el mismo nombre, ¿Cuál de los dos deberá heredarse? Java no intentó resolver el problema, sino que simplemente redujo la jerarquía de clases a una sola superclase por clase. Esto produce un árbol de herencia en lugar de un grafo y la herencia de métodos queda unívocamente determinada. Sin embargo, a veces es necesario poder referirse a un objeto a través de distintos tipos: Un Empleado además de Persona puede ser socio de un club y cocinero. Cada uno de estos tipos ofrecerá distintos comportamientos. El cocinero cocina() y el socio del club pagaCuota(). La herencia múltiple nos permite no sólo definir qué Empleado deberá implementar todos estos métodos, sino que si las superclases tienen una implementación, podemos heredarla.

Java toma un camino intermedio: Por un lado nos permite herencia completa de una sola superclase, tanto de la interfaz como de su implementación si existe, y por el otro nos permite declarar que una clase pertenece a más de un tipo: Esto se logra a través de interfaces. Una **Interfaz** no es más que una clase abstracta con todos sus métodos abstractos. Permite también definir constantes que heredarán las clases que la implementen o las interfaces que la extiendan.

Una clase puede extender una sola superclase, pero puede implementar múltiples interfaces. Una interfaz puede extender múltiples interfaces, pero no implementa ni hereda comportamiento (Esto cambia con Java 8, cuyas interfaces permiten definir la implementación de métodos por defecto, pero no vamos a verlo en este curso).



Las interfaces en Java se denominan comenzando con una I mayúscula, seguidas del nombre de la misma. La sintaxis para su declaración en java es la siguiente:

```
public interface ICocinero {
	String cocinar();
}

public interface ISocio {
	String pagarCuota();
}
```

De este modo declaramos las interfaces ISocio e ICocinero con sus respectivos métodos. La clase que las implemente (Empleado) deberá declararse de la siguiente manera:

```
public class Empleado extends Persona implements ICocinero, ISocio{

	private long legajo;
	private float sueldoMensual;

	public Empleado(String nombre, String apellido, long dni,
			LocalDate fechaNacimiento, long legajo,
			float sueldoMensual){
		super(nombre, apellido, dni, fechaNacimiento);
		this.legajo=legajo;
		this.sueldoMensual=sueldoMensual;

	}

	public float calcularSueldo(int diasAusente){...}

	public String cocinar(){
		return "Estoy cocinando";
	}

	public String pagarCuota(){
		return "Estoy Pagando la cuota";
	}
	public String hablar(){
		return "Soy un Empleado"
	}
}
```

Veremos cuál resulta ser el comportamiento dependiendo de cómo declaremos la variable que contendrá al objeto instanciado:

```
Empleado empleado = new Empleado(nombre, apellido, dni, fechaNacimiento, legajo, sueldoMensual);

float sueldo = empleado.calcularSueldo(2); //válido

int edad = empleado.calcularEdad(LocalDate.now()); //válido

empleado.hablar(); //válido

empleado.pagarCuota(); //válido

empleado.cocinar(); //válido
```

Todos los casos son válidos. Un Empleado se comporta como Empleado, Persona, ISocio e ICocinero.

```
Persona persona = new Empleado(nombre, apellido, dni, fechaNacimiento, legajo, sueldoMensual);

float sueldo = persona.calcularSueldo(2); //no válido

int edad = persona.calcularEdad(LocalDate.now()); //válido

persona.hablar(); // válido (implementado en Empleado)

persona.pagarCuota(); //no válido

persona.cocinar(); //no válido
```

Una Persona no es ni Empleado, ni ICocinero ni ISocio. Recordemos que hablar() está declarado abstracto en Persona, pero Empleado lo implementa. Si bien estamos tratando a un Empleado como Persona, no por eso deja de ser instancia de Empleado, y por eso la implementación del método que se utiliza es la de Empleado.

```
ICocinero cocinero = new Empleado(nombre, apellido, dni, fechaNacimiento, legajo, sueldoMensual);

float sueldo = cocinero.calcularSueldo(2); //no válido

int edad = cocinero.calcularEdad(LocalDate.now()); //no válido

cocinero.hablar(); //no válido

cocinero.pagarCuota(); //no válido

cocinero.cocinar(); //válido
```

Aquí vemos que un ICocinero solo puede comportarse como un ICocinero.

```
ISocio socio = new Empleado(nombre, apellido, dni, fechaNacimiento, legajo, sueldoMensual);

float sueldo = socio.calcularSueldo(2); //no válido

int edad = socio.calcularEdad(LocalDate.now()); //no válido

socio.hablar(); //no válido

socio.pagarCuota(); //válido

socio.cocinar(); //no válido
```

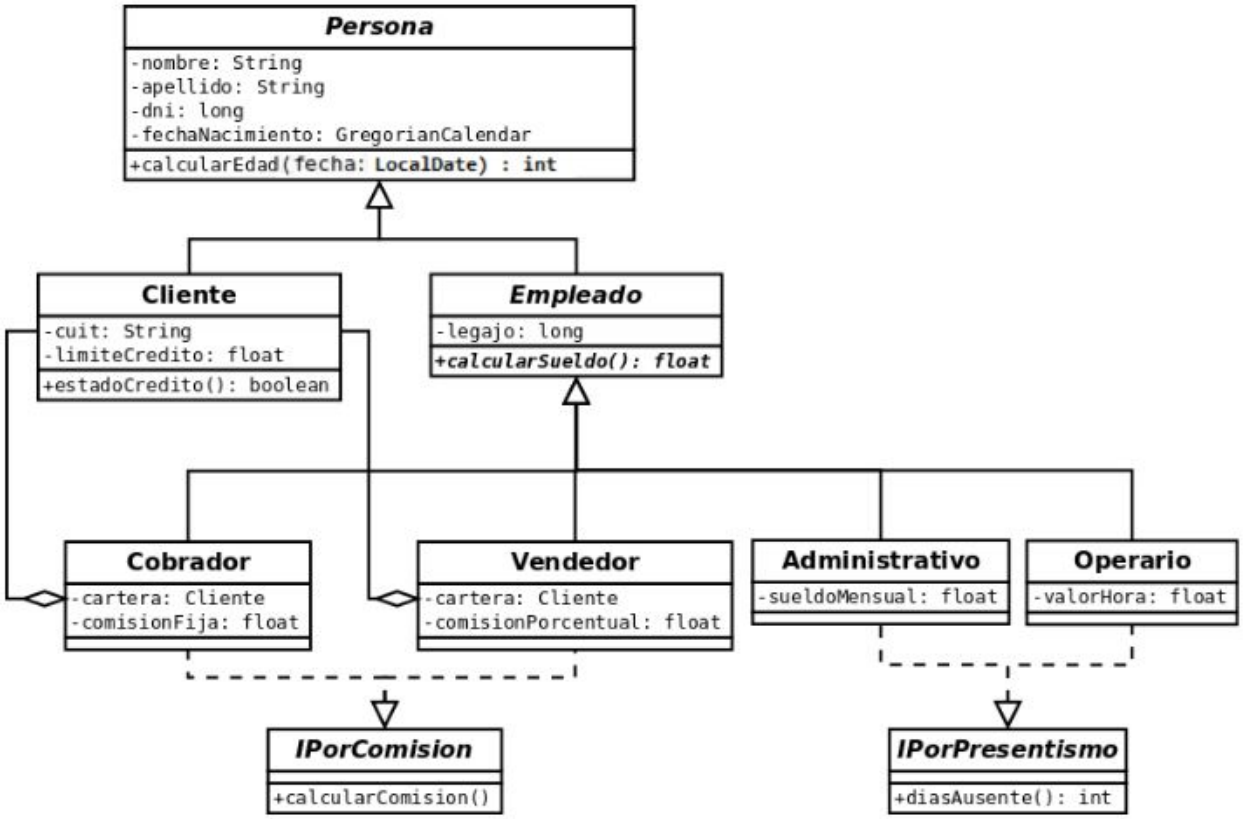
El mismo caso que para un ISocio.

¿Cuándo es mejor utilizar una superclase y cuando una interfaz? Como regla general (que como todas las reglas tiene sus excepciones), cuando hace falta que una clase deba representarse como más de un tipo determinado, las interfaces son inevitables. Sin embargo, una regla sencilla es utilizar interfaces cuando sólo se quiere definir un tipo. Si es necesario heredar comportamiento (métodos implementados) que deberán compartirse entre las distintas subclases, es mejor utilizar una superclase.

REALIZAR PRÁCTICA: 4.A (Herencia Multiple Persona)

Un ejemplo más complejo

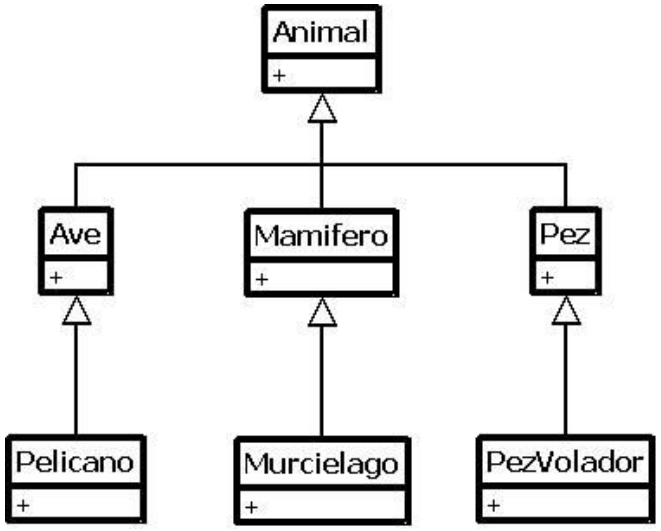
Ampliando un poco el ejemplo anterior veremos cómo pueden usarse simultáneamente cases, clases abstractas e interfaces.



Hay varias cosas que podemos notar en este ejemplo: Hemos agregado especializaciones de Empleado (Cobrador, Vendedor, Administrativo y Operario). Como no planeamos instanciar Empleados o Personas directamente, las hemos declarado abstractas. El método Empleado.calcularSueldo() también es abstracto, ya que cada tipo de empleado cobra de una manera distinta: El Cobrador percibe una suma fija por cada cobranza realizada, el Vendedor un porcentaje de la venta realizada, ambos sobre sus respectivas carteras de clientes. El Administrativo y el Operario cobran por presentismo (al Administrativo, que cobra un sueldo fijo mensual, se le descuentan los días ausente y al Operario, que es quincenal, se le paga un premio si no tuvo días ausentes). Las interfaces IPorComision e IPorPresentismo definen los tipos respectivos y declaran el método a implementar en cada una de las clases.

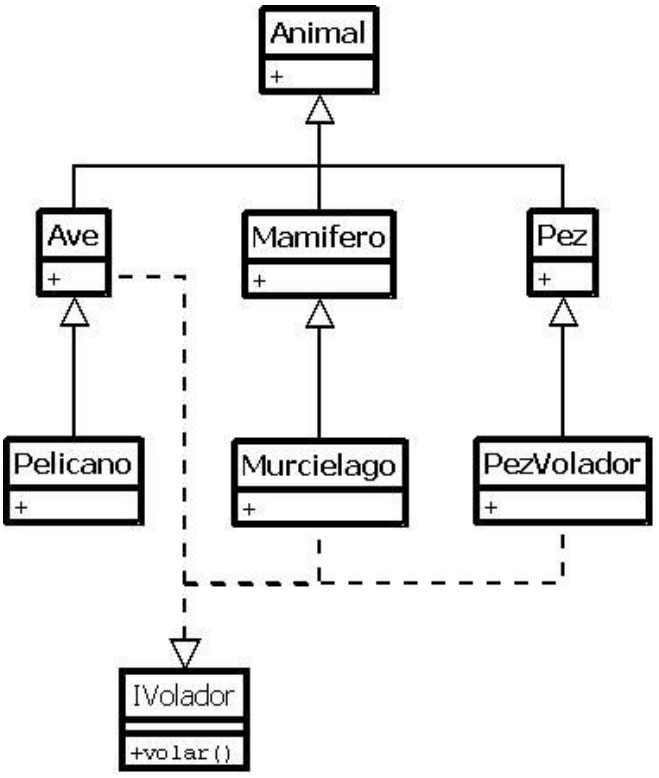
Puede verse en este ejemplo también una de las limitaciones de la herencia múltiple por medio de interfaces: Tanto Cobrador como Vendedor definen el atributo cartera. Esto ocurre porque las Interfaces no deben definir atributos: sólo definen el contrato con el que debe cumplir la clase (los nombres de los métodos que debe implementar, con sus parámetros y valores de retorno, es decir, la signatura de los mismos) para que pertenezca a su tipo. Si Java permitiera la herencia múltiple de clases, IPorComisión podría ser una superclase de Cobrador y Vendedor, la que definiría el atributo cartera y podrían heredarlo sus subclases. Una forma de resolverlo dentro del esquema de herencia simple sería crear una subclase de Empleado, EmpleadoPorComision, abstracta, que definiera el atributo Cartera. Si bien es correcto, agrega más dependencias a las subclases (mayor acoplamiento) y mayor complejidad al árbol de herencia. Por otro lado, si en algún momento tenemos que pagar comisiones a alguien que no sea un empleado (una agencia de cobranzas externa, por ejemplo), la duplicación de la definición del atributo cartera vuelve a aparecer y no podríamos tratar a todas las instancias de los que cobran comisiones del mismo modo (no heredarían de la misma superclase)

Otro ejemplo:



En qué clase(s) debería implementarse el mètodo volar()? Y en cuáles debería definirse para poder tratar a todos los animales que pueden volar de la misma manera? Suponemos, para este ejemplo, que todas las aves vuelan y lo hacen de la misma manera.

Para tratar a todos los animales que vuelan del mismo modo (como pertenecientes a un único tipo) definimos una interfaz IVolador y la implementamos en Ave (recuerden que suponemos que todas la aves vuelan y que lo hacen del mismo modo), en Murcielago y en PezVolador. La implementación de volar de ave es heredada por todas las subclases de la misma, mientras que tanto Murciélago como PezVolador tendràn la propia. De este modo podremos tratar todos los animales que vuelan como instancias de IVolador.



Repaso

¿Qué es lo que no se puede hacer con una clase Abstracta?
¿Qué es una interfaz?

Práctica

4.A (Herencia Multiple Persona)
Terminar de implementar las clases y los casos de uso del diagrama de la Pág. 8.