Licenciatura en Sistemas - Orientación a Objetos I – 2016

Profesora Titular: Lic. María Alejandra Vranić alejandravranic@gmail.com

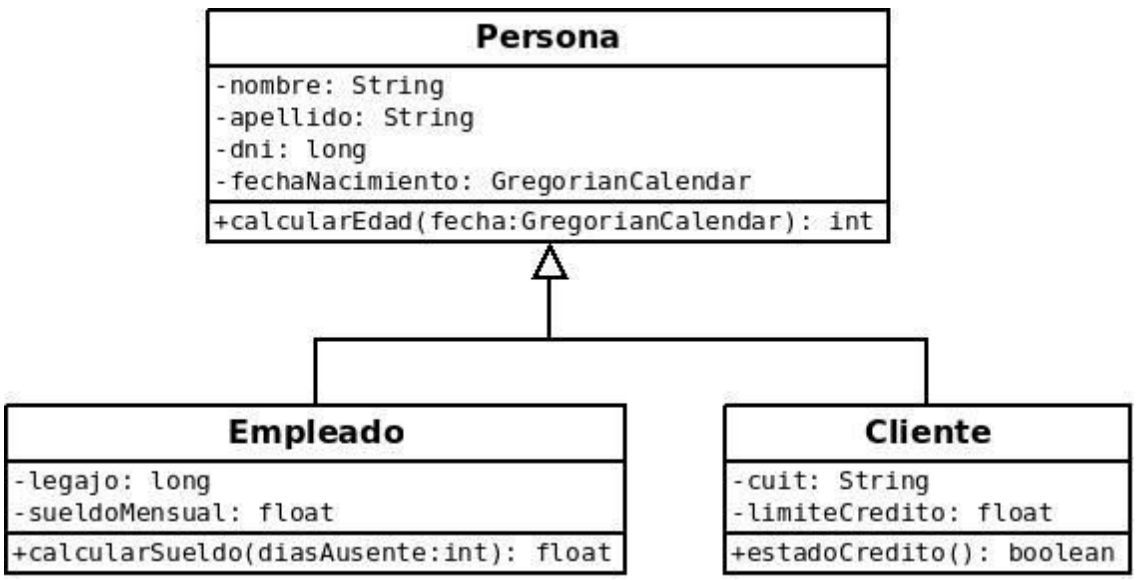
Prof. Ayudantes: APU Leandro Ríos leandro.rios.unla@gmail.com
. APU Gustavo Siciliano gussiciliano@gmail.com

Grupo en facebook: UNLa2016OO1

Herencia

El concepto de herencia es intuitivamente sencillo: Se trata de aumentar a una clase agregándole características (estado y comportamiento), pero preservando también el tipo original. Tengamos esto en cuenta: En el diseño de nuestras clases, los atributos son privados (sólo podemos accederlos a través de métodos mutadores – setters y getters), lo que implica que lo único que tenemos disponible a través de la interfaz del objeto es el comportamiento (los métodos). Esto nos permite decir que lo que diferencia a una clase de otra son las operaciones (métodos) que implementa, Un Animal come() y respira(). Una Paloma también lo hace, pero además vuela(). Un Pez además nada(). Si omitimos a los métodos vuela() y nada() vemos que podemos tratar a una Paloma y a un Pez como Animal. La inversa no es cierta, ya que un Animal le falta un método (vuela()) para tratarlo como Paloma.

En UML podemos representar la relación de herencia del siguiente modo:



Que Empleado extienda (herede de) Persona quiere decir que además de tener los atributos y métodos propios recibe los de Persona. Es decir, en un objeto Empleado (o Cliente) tendremos disponible el método calcularEdad(fecha). Un lenguaje orientado a objetos con herencia (como Java) se encarga de mantener la relación de herencia por nosotros. Además, un objeto de tipo Empleado puede tratarse como uno de tipo Persona con sólo declararlo. La siguiente sintaxis Java es perfectamente válida:

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

Y ahora empezamos a comprender algunas cosas que veníamos haciendo, como por ejemplo:

```
List<Pelicula> lista = new ArrayList<Pelicula>();
Calendar fecha= new GregorianCalendar();
```

es que estamos usando un tipo de herencia. Por ahora solo diremos que List es una Interfaz y Calendar es una clase abstracta; más adelante se explicarán ambos conceptos.

La sintaxis Java para crear una clase que hereda de otra es:

```
public class Persona {
	protected String nombre;
	protected String apellido;
	protected long dni;
	protected GregorianCalendar fechaNacimiento;

	public Persona(String nombre, String apellido, long dni,
			GregorianCalendar fechaNacimiento) {
		this.nombre = nombre;
		this.apellido = apellido;
		this.dni = dni;
		this.fechaNacimiento = fechaNacimiento;
	}

	public int calcularEdad(){...}

}

public class Empleado extends Persona {
	private long legajo;
	private float sueldoMensual;

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

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

Podemos ver que Persona es una clase común y corriente, como las que venimos usando hasta ahora. En la clase empleado hay dos cambios: La sentencia extends en la declaración de la clase y super en el constructor. Extends indica que la clase Empleado extiende a (hereda de) la clase Persona. Si la omitimos, la clase por defecto hereda de Object. Object es la raíz del árbol de herencia, y es la única clase que no hereda de ninguna otra. Todas las clases que hemos implementado hasta ahora heredan de Object

de manera implícita. Por eso siempre podemos llamar al método toString, por ejemplo: porque está definido en la clase Object. Las expresiones “Empleado extiende a Persona”, “Empleado hereda de Persona”, “Empleado es subclase de Persona” y “Persona es superclase de Empleado” son todas equivalentes.

Java es un lenguaje de herencia simple; esto quiere decir que todas las clases heredan de una sola. Varias clases pueden heredar de la misma (como Empleado y Cliente de Persona) pero no al revés. Existen otros lenguajes que implementan herencia múltiple. Más adelante veremos por qué y qué ventajas e inconvenientes tiene cada implementación.

La llamada a super(...) es la llamada al constructor de la superclase. Si se omite, java automáticamente llama al constructor de la superclase sin parámetros, si ese constructor no existe, tendremos un error de compilación. La llamada a super siempre debe ser la primera sentencia en el constructor. Super también se puede utilizar en métodos sobreescritos (override), para acceder a la funcionalidad del método de la superclase.

Otra diferencia con respecto a lo que venimos haciendo es la declaración de visibilidad de los atributos: en lugar de private, los estamos declarando como protected. Esta declaración permite que los atributos sean visibles por las subclases de la clase que los declara. El siguiente cuadro sumariza la visibilidad de atributos y métodos para cada modificador:

MODIFICADOR	CLASE	PAQUETE	SUBCLASE	TODOS
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado	Sí	Sí	No	No
private	Sí	No	No	No

No deben establecerse los atributos de la superclase directamente: Aunque el modificador protected los hace visibles en la subclase, el constructor de la superclase puede implementar validaciones que estaríamos salteando al hacerlo. El modificador protected se emplea para no tener que acceder a los atributos a través de los getters todo el tiempo. De todos modos, tengamos en cuenta que al acceder los atributos de la superclase directamente estaremos aumentando el acoplamiento entre clase y subclase. Incluso en el caso de las superclases, es mejor acceder a los atributos a través de los setters y getters. En nuestro caso, podríamos haber omitido el modificador ya que ambas clases se encuentran en el mismo paquete, pero una subclase de Persona en otro paquete no podrá acceder a los atributos de la superclase.

Clases y métodos abstractos

Una clase puede definirse como abstracta (utilizando el modificador abstract en la definición) cuando queremos definir una clase para ser extendida pero que no debe instanciarse. En nuestro caso, como en nuestro sistema no nos interesa tener instancias de la clase Persona (no es lo mismo tener una instancia de Persona que tratar a una instancia de Empleado o Cliente como Persona), pero sí la queremos utilizar como base para Empleado o Cliente. Para ello, podemos declararla como abstracta:

```
public abstract class Persona {...}
```

Esto implica que no podremos tener instancias de persona, es decir, lo siguiente es inválido:

```
Persona persona = new Persona(nombre, apellido, dni,
                              fechaNacimiento); //No
```

Pero esto sí se puede:

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

Persona expondrá los métodos de Persona (calcularEdad, por ejemplo) pero no tendrá ninguno de los de Empleado. Podemos pensar que al almacenar una instancia de la subclase en una variable del tipo de la superclase los métodos de la primera se "enmascaran" y ya no estarán disponibles.

Un método abstracto es un método que sólo se declara, que no tiene cuerpo. Se declara para que lo implementen las subclases, es una forma de obligar a las mismas a respetar una interfaz o contrato determinado. Por ejemplo, en la clase persona podríamos declarar:

```
public abstract String hablar();
```

En Empleado:

```
public String hablar(){
      return "Soy un Empleado";
}
```

Y en Cliente:

```
public String hablar(){
      return "Soy un Cliente";
}
```

Si en la subclase no se implementa el método, tendremos un error de compilación. Una subclase puede no implementar un método abstracto de la superclase, pero para eso deberá declararse abstracta ella misma y por lo tanto, tampoco podrá instanciarse. Basta con que una clase declare un método abstracto para que ella misma sea abstracta.

Herencia simple, herencia múltiple e Interfaces

Si llevamos el concepto de la clase abstracta al extremo, nos encontraremos con una clase abstracta que declara 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,

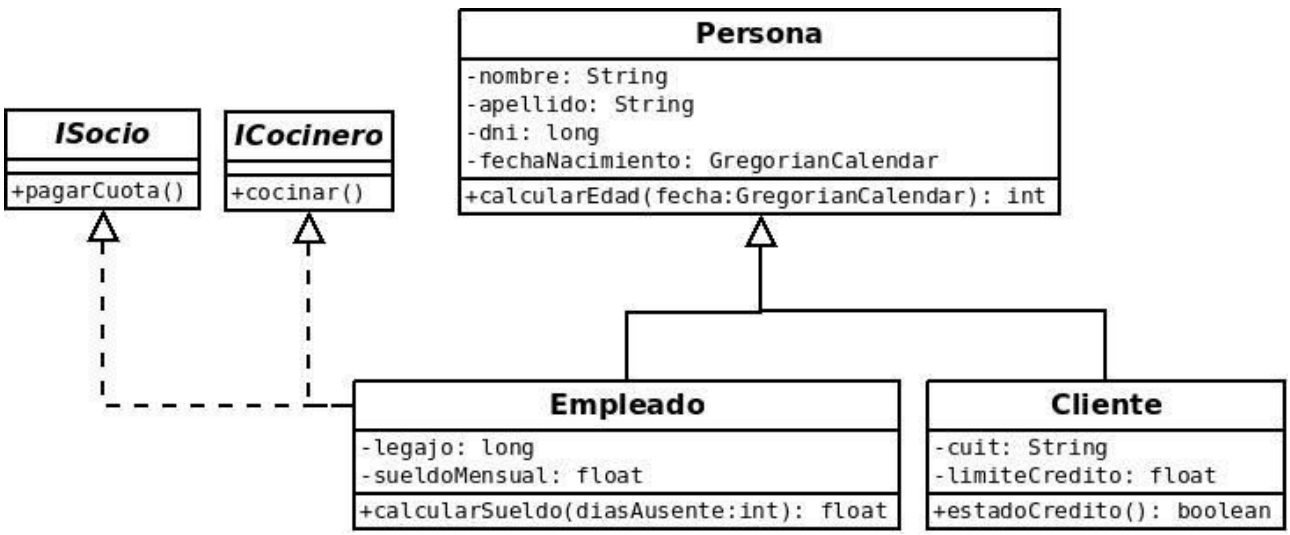
tengo una lista de Animales y quiero recorrerla haciendo respirar a cada uno. En ese caso no me importa de qué animal específico se trate.) Esto es muy importante, porque nos permite diseñar software que se pueda extender de manera modular especificando la interfaz (el contrato) que debe respetar cada 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 múltiple (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.
Pero 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 que 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 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,
			GregorianCalendar 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"
	}
}
```

Vamos a ver el comportamiento dependiendo de cómo declaremos el objeto instanciado:

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

float sueldo = empleado.calcularSueldo(2); //válido
int edad = empleado.calcularEdad(new GregorianCalendar()); //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(new GregorianCalendar()); //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(new GregorianCalendar()); //no válido
cocinero.hablar(); //no válido
cocinero.pagarCuota(); //no válido
cocinero.cocinar(); //válido
```

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

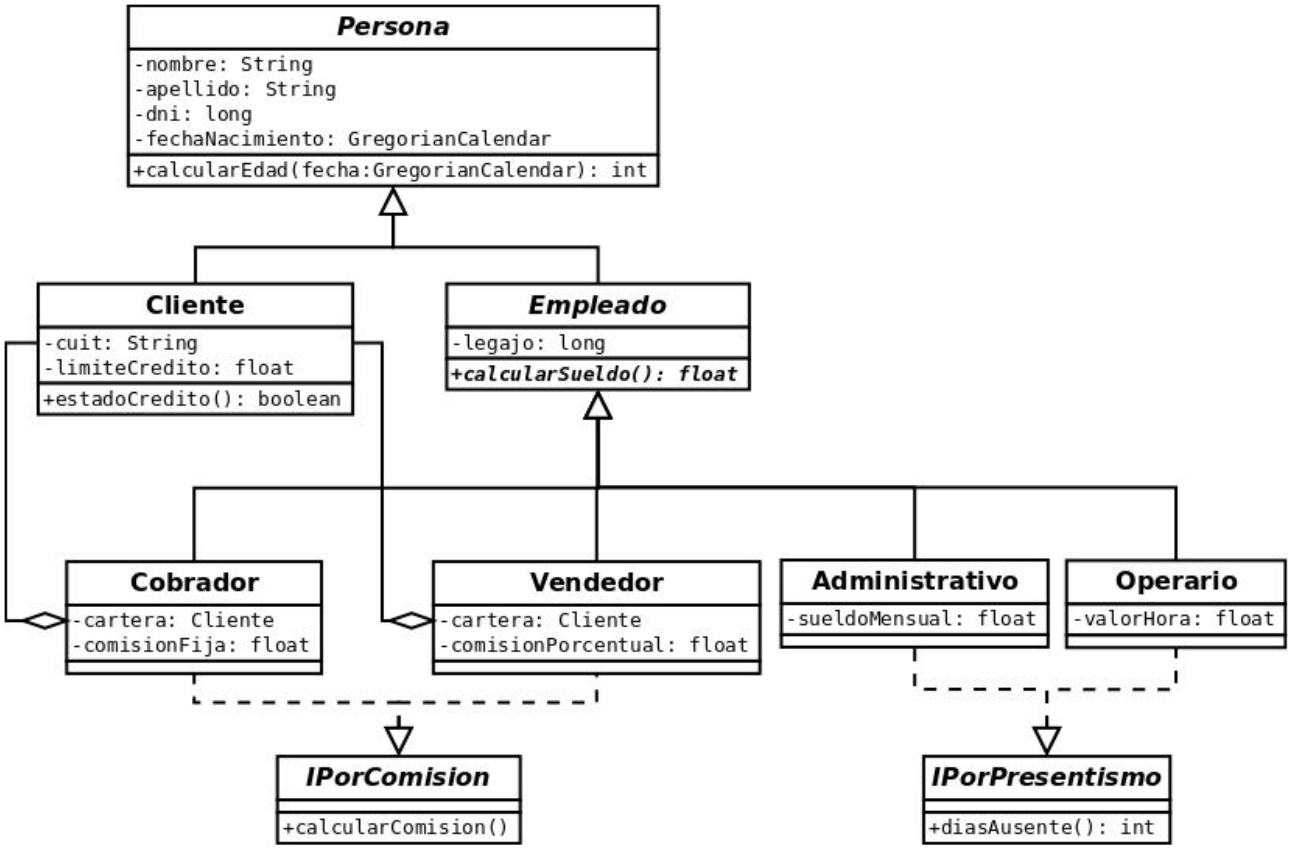
```
ISocio socio = new Empleado(nombre, apellido, dni, fechaNacimiento,
          legajo, sueldoMensual);

float sueldo = socio.calcularSueldo(2); //no válido
int edad = socio.calcularEdad(new GregorianCalendar()); //no válido
socio.hablar(); //no válido
socio.pagarCuota(); //válido
socio.cocinar(); //no válido
```

El mismo caso que para un ICocinero.

¿Cuándo es mejor utilizar una superclase y cuando una interfaz? 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.

Un ejemplo más complejo

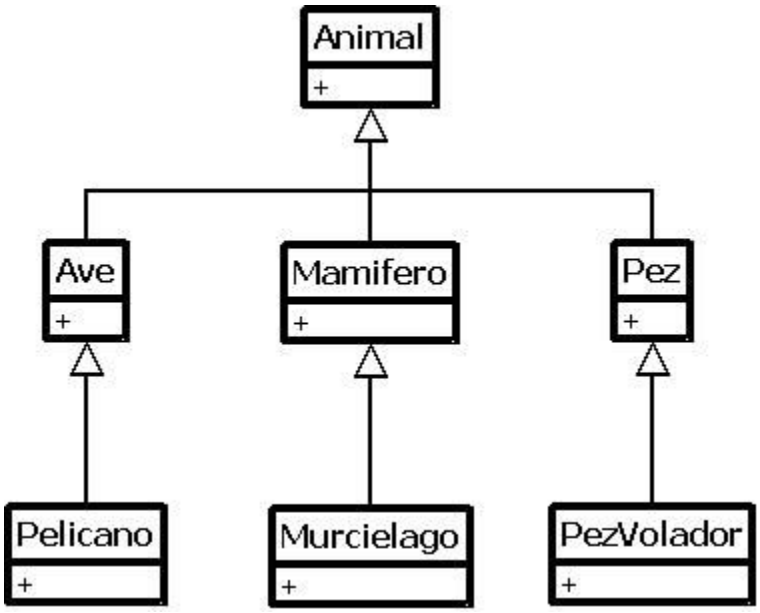
Ampliando un poco el ejemplo anterior veremos cómo pueden usarse simultáneamente cases, clases abstractas e interfaces.



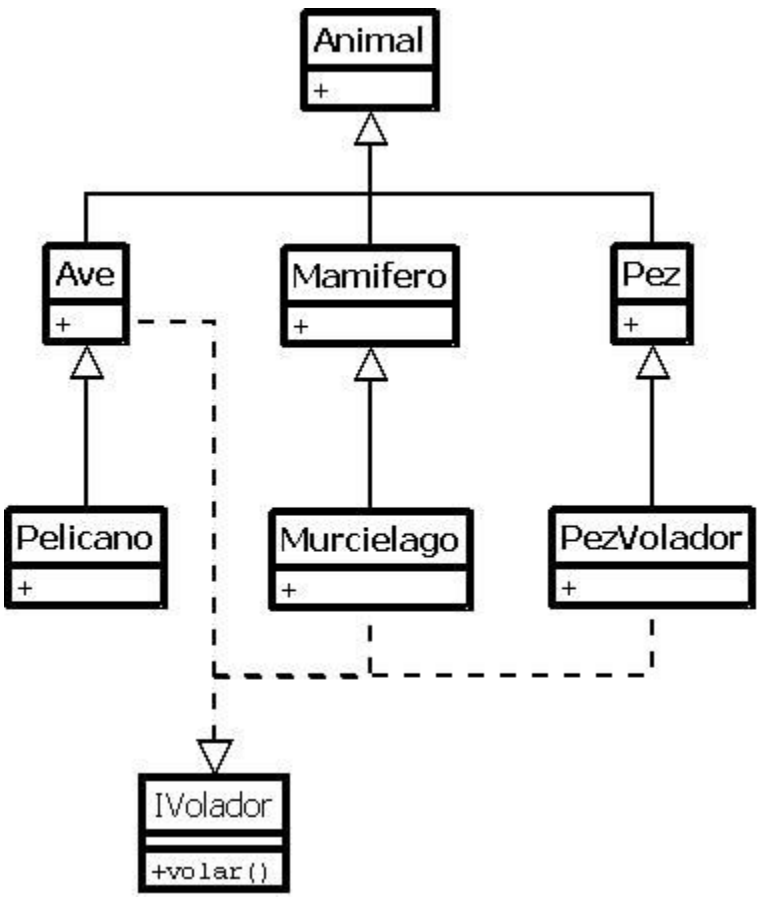
Hay varias cosas que podemos notar en este ejemplo: Hemos agregado especializaciones de Empleado (Cobrador, Vendedor, Administrativo y Operario). Como no planeamos instanciar Empleados o Personas directamente, las hemos declarado abstractas. El método Empleado.calcularSueldo() también es abstracto, ya 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 que clase(s) debería implementarse el mètodo volar()? Y en cuales 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.

Casteo (casting)

Cuando vimos superclases e interfaces dijimos que sólo podíamos invocar los métodos definidos en el tipo de la variable donde estamos almacenando la instancia del objeto. Por ejemplo:

```
Persona persona = new Empleado(nombre, apellido, dni, fechaNacimiento,
                                legajo, sueldoMensual);
float sueldo = persona.calcularSueldo(2); //no válido
```

Si bien esto es inválido, persiste el hecho de que la instancia que creamos es de Empleado, no de Persona. Existe una forma de tratarla como Empleado e invocar sus métodos, a pesar de que no la hayamos declarado como tal. Lo que hacemos es decirle a Java que esta instancia que poseemos en realidad es de tipo Empleado y la almacenamos en una variable de tipo empleado. Así, con la instancia definida recién, podemos hacer

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

Esta operación -`(Empleado) persona`- se denomina *casteo* (casting en inglés) y se emplea precisamente cuando queremos pasar de un supertipo (Persona) a un subtipo (Empleado) cuando tenemos una instancia del subtipo declarada como del supertipo. No podemos castear Empleado a Cliente, ya que son dos subclases distintas de la misma superclase. El casteo sólo puede realizarse desde un tipo más general, representado por una superclase, a un tipo más específico: una subclase de la misma.

Como programadores, la responsabilidad es nuestra: Le estamos diciendo al compilador Java que ***nos crea*** que la instancia que tenemos en persona es en realidad del tipo Empleado y que lo trate como tal. De todos modos, Java no nos deja absolutamente sin protección, ya que si tratáramos de castear un Empleado como Cliente obtendríamos una excepción en tiempo de ejecución ClassCastException; o en el caso de querer castear un String como Empleado (por dar un ejemplo extremo), daría un error al compilar.

Un caso de uso posible:

Supongamos que tenemos una lista de Personas, que contiene instancias tanto de Empleado como de Cliente y que queremos recorrerla calculando la edad de todas y el sueldo de las que son empleados:

```
List<Persona> personas =  new ArrayList<Persona>();
// ...agregamos todas las instancias que queramos...
personas.add(new Empleado(...));
```

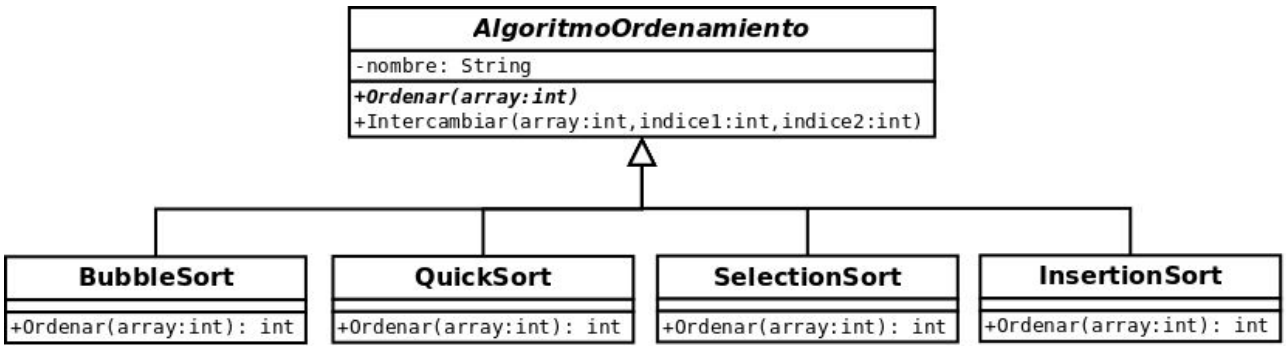
```
personas.add(new Cliente(...));

// ...y recorremos la lista mostrando:
for(Persona persona:personas){
      System.out.println(persona);
      System.out.println("Edad: "+persona.calcularEdad());
      if(persona instanceof Empleado){
            Empleado empleado = (Empleado) persona; // casteamos
            System.out.println("Sueldo: "+empleado.calcularSueldo(2));
      }
}
```

El operador `instanceof` se utiliza para saber si un objeto es instancia de (instance of) una clase determinada. Con el mismo nos aseguramos de castear a Empleado sólo las instancias de esa clase y no, por ejemplo de Cliente, lo que nos generaría un error. De este modo podemos recorrer una lista de objetos declarados como pertenecientes a la superclase e invocar los métodos específicos de las subclases de la misma.

Práctica: Un caso concreto

El problema: Crear un pequeño banco de pruebas para distintos algoritmos de ordenamiento sobre el mismo set de datos. Para ello creamos una superclase de los distintos tipos de algoritmos de ordenamiento. Cada subclase del mismo implementa el algoritmo específico (burbuja, quicksort, inserción, etc). Esto permite que nuestra implementación del banco de pruebas pueda invocar al método ordenar() de los distintas instancias de métodos de ordenamiento sin saber a qué clase específica pertenece cada uno, ya que a todos los trata como AlgoritmoOrdenamiento. La primera ventaja de este diseño modular es que podemos agregar métodos de ordenamiento implementándolos como subclases de AlgoritmoOrdenamiento, y nuestro banco de pruebas podrá invocarlos sin hacerle ninguna modificación..

El diagrama de clases es el siguiente:



Crear dos paquetes, ordenamiento y test. En ordenamiento, crear las siguientes clases:

AlgoritmoOrdenamiento.java:

```
package ordenamiento;
```

```java
import java.util.Arrays;

// Superclase de todas las clases que representan un algoritmo de
ordenación
public abstract class AlgoritmoOrdenamiento {
	String nombre;

	// Constructor
	public AlgoritmoOrdenamiento(String nombre) {
		super();
		this.nombre = nombre;
	}

	public String getNombre() {
		return nombre;
	}

	// el nombre lo carga la clase especifica (la subclase)
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}


	// Método de ordenamiento, implementar en cada subclase
	public abstract int[] ordenar(int[] array);

	// Método ayudante, varios algoritmos lo usan
	protected void intercambiar(int[] array, int indice1, int
indice2){
		//intercambia los valores de las posiciones indice1 e
indice2 en
		//array
		int tmp=array[indice1];
		array[indice1]=array[indice2];
		array[indice2]=tmp;
	}

	@Override
	public String toString() {
		return nombre;
	}

}
```

BubbleSort.java (método de la burbuja):

```java
package ordenamiento;

public class BubbleSort extends AlgoritmoOrdenamiento {

	public BubbleSort() {
		super("Método de la Burbuja");
	}
```

```
	@Override
	public int[] ordenar(int[] array) {
		return burbuja(array);
	}

	public int[] burbuja(int[] array) {
		int lenD = array.length;
		int tmp = 0;
		boolean ordenado=false;
		for(int i = 0;i<lenD && !ordenado;i++){
			ordenado=true;
			for(int j = (lenD-1);j>=(i+1);j--){
				if(array[j]<array[j-1]){
					ordenado=false;
					intercambiar(array,j,j-1);
				}
			}
		}
		return array;
	}
}
```

QuickSort.java:

```
package ordenamiento;

public class QuickSort extends AlgoritmoOrdenamiento {

	public QuickSort() {
		super("QuickSort");
	}

	@Override
	public int[] ordenar(int[] array) {
		return quickSort(array,0,array.length);
	}

	// version recursiva de quicksort
	public int[] quickSort(int[] arr, int comienzo, int fin) {
	    if (fin - comienzo < 2) return arr; //clausula de finalización
	    int p = comienzo + ((fin-comienzo)/2);
	    p = particionar(arr,p,comienzo,fin);
	    quickSort(arr, comienzo, p);
	    quickSort(arr, p+1, fin);
	    return arr;

	}

	// Metodo para efectuar la particion en el pivote
	private int particionar(int[] array, int p, int comienzo, int fin)
{
```

```
            int c = comienzo;
            int f = fin - 2;
            int pivote = array[p];
            intercambiar(array,p,fin-1);

            while (c < f) {
                if (array[c] < pivote) {
                    c++;
                } else if (array[f] >= pivote) {
                    f--;
                } else {
                    intercambiar(array,c,f);
                }
            }
            int indice = f;
            if (array[f] < pivote) indice++;
            intercambiar(array,fin-1,indice);
            return indice;
        }

}
```

En el paquete test, crear la siguiente clase:

TestOrdenamiento.java:

```
package test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

import ordenamiento.AlgoritmoOrdenamiento;
import ordenamiento.BubbleSort;
import ordenamiento.QuickSort;

public class TestOrdenamiento {

      public static void main(String[] args) {

            // generamos los arrays de datos
            int ordenMagMax=1; // orden de magnitud máximo
                               // de la cantidad de elementos
            // en este caso, sólo un array de 10 elementos
            // Pueden probar, con 2 genera un array de 10 y otro
            // de 100 elementos, con 3...
            List<int[]> arrays=new ArrayList<int[]>();
            for(int k=1;k<=ordenMagMax;k++){
                  arrays.add(generarArray(0,(int)Math.pow(10,(k+1)),
                                    (int)Math.pow(10,k)));
            }
```

```
        System.out.println("Arrays creados");

// instanciamos los algoritmos
// Vean como la lista se declara para la superclase
        List<AlgoritmoOrdenamiento> listaAlgoritmos=new
                                ArrayList<AlgoritmoOrdenamiento>();

// Y sin embargo la cargamos con instancias de la clase
        listaAlgoritmos.add(new BubbleSort());
        listaAlgoritmos.add(new QuickSort());

        System.out.println("Algoritmos creados");

        // ejecutamos el test
        for(int[] array: arrays){ // recorremos la lista de arrays
                // mostramos el array original
                mostrarArray("Array original: ", array);
                for(AlgoritmoOrdenamiento algoritmo: listaAlgoritmos){
                        // recorremos la lista de algoritmos
                        // ordenamos y mostramos. hacemos una copia del
                        // array para asegurarnos
                        // de que el mismo no se encuentra ordenado por el
                        // método anterior
                        int[] ordenado = algoritmo.ordenar(Arrays.copyOf(
                                        array,array.length));
                        mostrarArray(algoritmo+": ", ordenado);
                }
        }
}


public static int[] generarArray(int inicio, int fin,int
                                        cantidad){
        // generamos una secuencia ordenada y luego la mezclamos
        // esto es para evitar elementos repetidos
        int[] array=crearSecuencia(inicio,fin,cantidad);
        mezclarArray(array);
        return array;
}

// recibe un array y lo mezcla (intercambia los elementos)
// en el lugar al azar.
private static void mezclarArray(int[] array)
{
    int indice, temp;
    Random random = new Random();
    for (int i = array.length - 1; i > 0; i--)
    {
        indice = random.nextInt(i + 1);
        temp = array[indice];
        array[indice] = array[i];
        array[i] = temp;
    }
```

```
    }

    // Crea una secuencia de cantidad de enteros de inicio a fin
    // se genera con intervalos iguales entre valores
    public static int[] crearSecuencia(int inicio, int fin, int
                                        cantidad)
    {
        int[] array = new int[cantidad];
        int salto=(fin-inicio)/cantidad;

        for (int i=0; inicio < fin; i++){
            array[i] = inicio+=salto;
        }
        return array;
    }

    public static void mostrarArray(String mensaje, int[] array){
         System.out.println(mensaje+Arrays.toString(array));
    }

}
```

1- Asegurarse de que funcione. Así como está, la ejecución de TestOrdenamiento.java debe mostrar por consola:

Arrays creados
Algoritmos creados
Array original: [10, 80, 30, 70, 90, 40, 100, 20, 60, 50]
Método de la Burbuja: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
QuickSort: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

2. Investigar los algoritmos de ordenamiento por selección y por inserción. Crear clases que los implementen y que hereden de AlgortimoOrdenamiento, reutilizando el método intercambiar() si es necesario. Agregarlo a la lista de algoritmos en test y probar que funcione. El resultado debe ser:

Arrays creados
Algoritmos creados
Array original: [60, 70, 100, 10, 50, 40, 80, 30, 20, 90]
Método de la Burbuja: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
QuickSort: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Método de Seleccion: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Método de Insercion: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

3. Cambiar AlgoritmoOrdenamiento a una interfaz IAlgoritmoOrdenamiento. Modificar las subclases para que en lugar de extender AlgoritmoOrdenamiento, implementen IAlgoritmoOrdenamiento. Correr el test y escribir un informe donde se consignen las diferencias observadas con la implementación anterior, incluyendo el diagrama de clases del nuevo diseño.