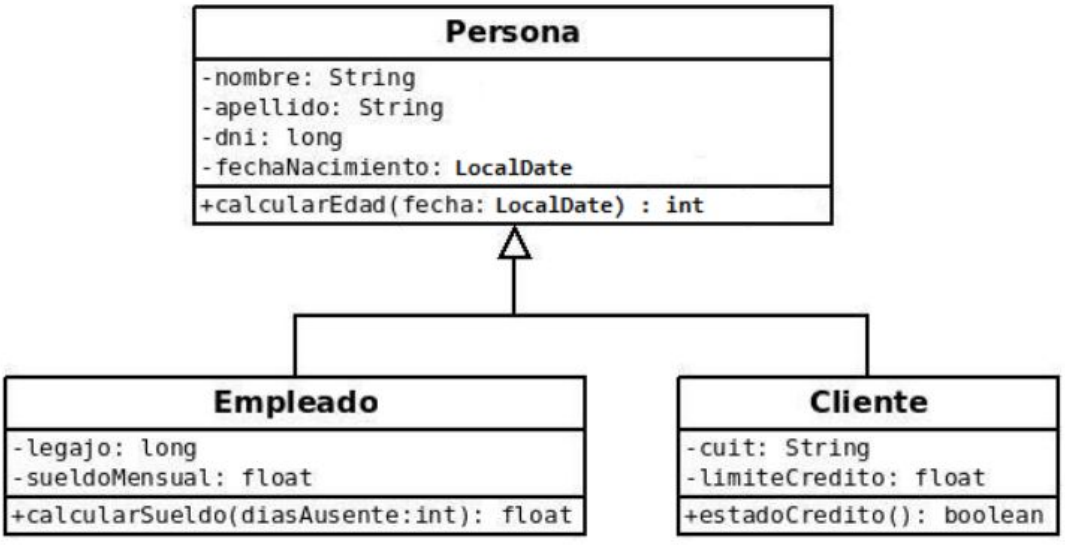
Licenciatura en Sistemas - Orientación a Objetos I – 2019

Prof. Titular:	Mg. María Alejandra Vranić	alejandravranic@gmail.com
Prof. Ayudantes:	Lic. Romina Mansilla	romina.e.mansilla@gmail.com
	Lic. Leandro Ríos	leandro.rios.unla@gmail.com
	Lic Gustavo Siciliano	gussiciliano@gmail.com

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 observándolas desde fuera, lo que diferencia a una clase de otra son las operaciones -métodos- que implementa. Un Animal puede comer() y respirar(). Una Paloma también lo hace, pero además puede volar(). Un Pez además puede nadar(). Si omitimos a los métodos volar() y nadar() 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 -volar()- 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 todos los atributos de Persona y 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);
```

La sintaxis Java para crear una clase que hereda de otra es la siguiente. Primero la superclase:

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

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

	public int calcularEdad(){...}

}
```

Y luego, la subclase:

```
public class Empleado extends Persona {
	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){...}
}
```

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 la llamada a **`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 sin definirlo, 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 de múltiples superclases. Más adelante veremos 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, seguirá subiendo por la jerarquía de clases hasta llegar a Object, que por defecto lo tiene implementado. 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, actuando como una referencia a la misma.

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 dentro de las subclases. De todos modos, tengamos en cuenta que al acceder a 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 de ser así una subclase de Persona en otro paquete no podría acceder a los atributos de la superclase.

Polimorfismo

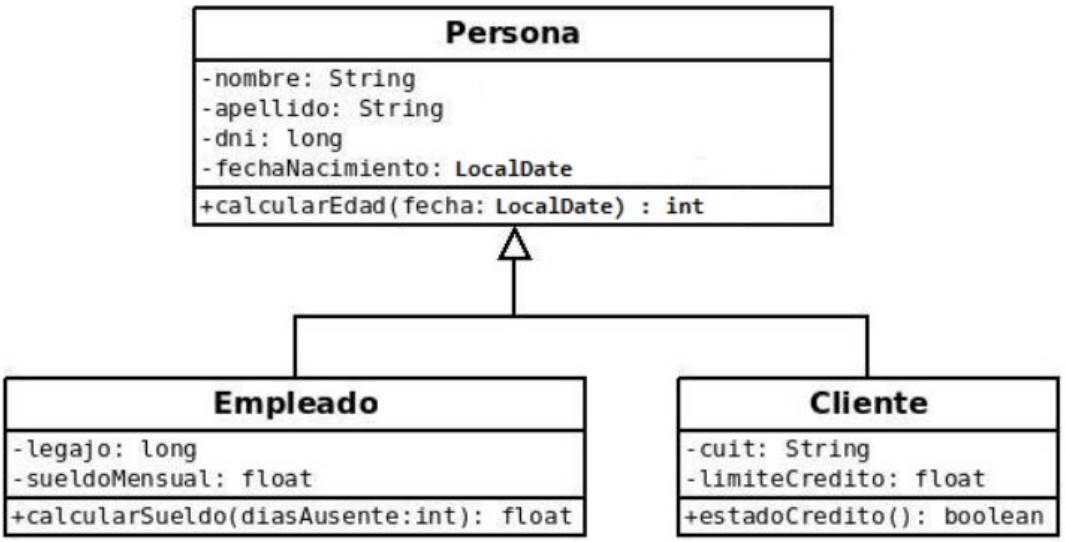
Existen varias definiciones de polimorfismo, que si bien son similares, no son exactamente iguales. La definición más general exige que para que dos objetos puedan tratarse de manera polimórfica, debe poder enviárseles los mismos mensajes (esto significa que deben exhibir el mismo comportamiento: deben exponer los mismos métodos con exactamente las mismas firmas). En la jerga de objetos original, se dice que los objetos interactúan entre sí enviándose mensajes. Estos mensajes están representados por los métodos del objeto y en éstos términos, “enviar un mensaje” es sinónimo de “invocar un método”.

Un ejemplo extremo de este tipo de polimorfismo es lo que se denomina “duck typing” (Tipado del pato), según el cual los tipos se definen sólo por los métodos que implementan y dos objetos son del mismo tipo si implementan todos los métodos del tipo. Esta es una definición dinámica, y por lo general no especificada en el código, salvo como un comentario. Al no existir especificación del tipo en el código, tanto la IDE como el compilador deben ser muy sofisticados para detectar errores de tipado, y por lo general lo que ocurre es que los compiladores y la ide simplemente no intentan detectar dichos errores. Esto implica que si hay un error de tipado, nos enteraremos de su existencia en tiempo de ejecución. Se llama “duck typing” por aquel dicho “si camina como un pato, nada como un pato y hace cuac como un pato, debe ser un pato”.

A fin de poder especificar este tipo de polimorfismo y que tanto la IDE como el compilador y los programadores que lean nuestro código tengan información al respecto de los tipos que implementan nuestras clases, se emplean lo que se denomina interfaces. Éstas definen qué métodos debe implementar una clase para pertenecer a dicho tipo. Java permite polimorfismo por múltiples interfaces, que veremos más adelante en la cursada.

Una definición más estricta exige que para poder tratar a dos objetos como pertenecientes al mismo tipo deben pertenecer a una jerarquía de clases y tener una raíz común. La clase que se encuentra en dicha raíz común es la que define el tipo al que pertenecen ambos objetos, en nuestro ejemplo, empleado y cliente son polimórficas porque tienen a la clase Persona en común. Java permite también este tipo de polimorfismo, al que suele denominarse polimorfismo de subclase.

Qué problema resuelve el polimorfismo? Supongamos que queremos que nuestro sistema calcule las edades de nuestros clientes y empleados para hacer algo sólo con aquellos mayores de 30 años. Para ello tendremos un array con instancias tanto de Persona como Empleado y Cliente:




```
Persona personas[] = new Persona[4]; // el con el que declaramos el array es el de la superclase
personas[0]=new Persona(...); // instanciamos una persona, todo bien
personas[1]=new Empleado(...); // instanciamos un empleado. Se puede?
personas[2]=new Cliente(...); // Y con cliente?
personas[3]=new Empleado(...);

for(int i=0; i<4;i++){
    if(personas[i].calcularEdad()>=30)
      ... hacemos algo con la instancia ...
    }
}
```

El código anterior funciona sin problemas. En una variable declarada como de la superclase (clase Persona) se pueden almacenar instancias tanto de Persona (por supuesto) como de cualquiera de sus subclases (Cliente y Empleado). Si no contáramos con polimorfismo, deberíamos tener tres arrays distintos (para Persona, Cliente y Empleado) y repetir el código del loop para cada uno de ellos. Utilizando polimorfismo resolvimos el problema con menos código y de manera mucho más elegante.

Casteo (Casting)

Algo que no podemos hacer es invocar métodos de la subclase en un objeto almacenado en una variable declarada como del tipo de la superclase:

```
Persona persona= New Empleado(...);
int edad=persona.calcularEdad(); // válido
float sueldo=persona.calcularSueldo(); // 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 creada antes, podemos hacer

```
Empleado empleado=(Empleado) persona; //casting
float sueldo=empleado.calcularSueldo(); // 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:

```
Persona personas[] = new Persona[10];
// ... agregamos los empleados y clientes que haga falta ...
personas[0]=new Empleado(...);
personas[1]=new Cliente(...);

// ...

// ...y recorremos el array mostrando:
for(int i=0; i<10;i++){
	Persona persona=personas[i];
	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. De todos modos, cuidado: si nos encontramos utilizando `instanceof` a menudo, ello es señal de que algo en el diseño no está bien.

Redefinición o sobreescritura de métodos (override)

La sobreescritura de métodos nos permite cambiar el comportamiento que heredamos de la superclase (los métodos) redefiniéndolo en la subclase. Un ejemplo lo tenemos en el método toString: Si no lo definimos en nuestra subclase, se utilizará el de la superclase. Si esta tampoco lo tiene definido, seguirá buscando hacia arriba por la jerarquía de clases hasta la clase Object, que lo tiene definido por defecto. No se debe confundir la redefinición o sobreescritura con la sobrecarga, que tiene otra aplicación. La sobrecarga nos permite acceder al mismo comportamiento con distintos datos. En el ejemplo de la clase Círculo el método setRadio se sobrecarga, ya que se puede invocar tanto con un int (la magnitud del radio) como con un Punto (para calcular la distancia).

Preguntas:

¿Qué es lo que las subclases heredan de la superclase? ¿Atributos, métodos o ambos?
¿Para qué sirve el polimorfismo?
¿Para qué redefinimos métodos heredados?
¿Es lo mismo redefinir que sobrecargar? ¿Para qué sobrecargamos métodos?
¿Para qué sirve el casteo?