Contenido

Herencia

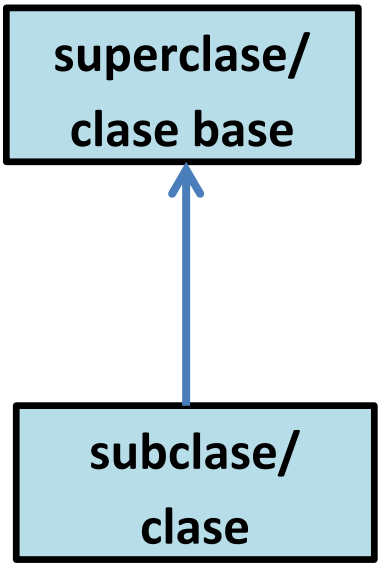
La herencia constituye uno de los pilares más más importantes y potentes de la POO.

Concepto de herencia: Capacidad de crear clases que adquieran de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.

La herencia permite la **reutilización de código**, al evitar tener que reescribir todos los métodos en la nueva clase y el **mantenimiento de las aplicaciones existentes**. En este caso, si tenemos una clase con determinada funcionalidad y necesitamos ampliar dicha funcionalidad, no es necesario modificar la clase existente, sino que podemos crear una clase que herede a la primera, aprovechando toda su funcionalidad y añadiendo la suya propia.

Nomenclatura y reglas

En POO a la clase que va a ser heredada se la llama superclase o clase base y a la que hereda se la denomina subclase o clase derivada. Gráficamente la herencia entre dos clases se representa con una flecha saliendo desde la subclase hacia la superclase, del siguiente modo:



Existen una serie de reglas básicas en Java, relacionadas con la herencia que hay que tenerlas en cuenta:

- En Java no está permitida la herencia múltiple, es decir, una subclase no puede heredar más de una clase.
- Si es posible la herencia multinivel, o sea, A puede ser heredada por B y C puede heredar B.
- Una clase puede ser heredada por varias clases.

La herencia entre dos clases establece una relación entre las mismas de tipo “es un”, lo que significa que un objeto de una subclase es también un objeto de la superclase. Así, vehículo es la superclase de Coche, por lo que Coche también es un Vehículo. De la misma forma Persona es la superclase de Alumno y esta a su vez de la superclase AlumnoUnlam por lo cual AlumnoUnlam “es un” Alumno y “es una” (en este caso por ser femenino el nombre de la clase) Persona.

Creación de herencia en Java

Veamos un ejemplo, para que sea más fácil la comprensión de este pilar de la POO. Sea la clase Cuenta y la subclase CuentaCorriente derivada de ella:

```
package ar.edu.unlam.basica2;

public class Cuenta{
	private Double saldo;

	public Cuenta(Double saldo){
		this.saldo = saldo;
	}

	public Cuenta(){
		this.saldo = 0.0D;
	}

	public void depositarDinero(Double importe){
		this.saldo+=importe;
	}

	public Double extraerDinero(Double importeARetirar){
		Double importeRetirado;

		if(importeARetirar<=this.saldo){
			saldo-=importeARetirar;
			importeRetirado = importeARetirar;
		}
		else{
			importeRetirado = 0.0D;
		}

		return importeRetirado;
	}

	public Double consultarSaldo(){
		return (getSaldo());
	}

	public Double getSaldo(){
		return this.saldo;
	}
}
```

Para definir que una clase va a heredar a otra clase se utiliza la palabra ***extends,*** seguida del nombre de la superclase en la cabecera de la declaración.

```
public class subclase extends superclase
{
	//código de la subclase
}
```

En nuestro ejemplo, la clase CuentaCorriente heredaría de la siguiente manera:

```
package ar.edu.unlam.basica2;

public class CuentaCorriente extends Cuenta {

	private Double sobregiro;

	public Double getSobregiro() {
		return sobregiro;
	}

	public CuentaCorriente(){
		sobregiro = 300.0;
	}

	public CuentaCorriente(Double saldoInicial){
		super(saldoInicial);
		sobregiro = 300.0;
	}

	public Double extraerDinero(Double importeARetirar){
		Double importeRetirado;

		if(importeARetirar<=super.getSaldo()){
			importeRetirado =
super.extraerDinero(importeARetirar);
		}
		else if(importeARetirar<=(super.getSaldo() +
this.sobregiro)){
			sobregiro -= (importeARetirar - super.getSaldo());
			super.extraerDinero(super.getSaldo());
			importeRetirado = importeARetirar;
		}
		else{
			importeRetirado = 0.0;
		}
		return importeRetirado;
	}
```

```
}
```

Como se observa en el ejemplo, la nueva clase CuentaCorriente agrega un atributo (sobregiro) y métodos propios para completar su función (dos constructores; uno donde simplemente se inicializa al atributo sobregiro y otro constructor donde además de inicializar el atributo sobregiro se accede al valor de saldo, que es un atributo propio de la clase Cuenta, además de un método *get* y un método que permite evaluar si es posible retirar un determinado importe.

Todas las clases en Java heredan alguna clase, implícitamente todas heredan la clase **Object** (sin necesidad de especificarlo con extends). En la clase Object se encuentra el paquete java.lang y constituye el soporte básico para cualquier clase en Java. La clase Object, por lo tanto es la superclase de todas las clases de Java.

Aunque una subclase hereda todos los miembros de la superclase, incluidos los privados, no tiene acceso directo a éstos ya que son privados de la clase y solamente accesibles desde el interior de ésta (por el principio de encapsulamiento).

Para poder acceder a los atributos privados de una superclase pueden utilizarse los métodos *set* y *get* de la misma desde la subclase, al ser heredados por ésta. Aunque lo ideal a la hora de inicializar atributos es el uso de constructores. En Java, cada vez que se crea un objeto de una clase, antes de ejecutarse el constructor de dicha clase, se ejecuta primero el de su superclase, dado que Java añade como primera línea de código en todos los constructores de una clase la siguiente instrucción:

```
super();
```

que provoca la llamada al constructor sin parámetros de la superclase. Si en vez de llamar al constructor sin parámetros deseáramos invocar a un constructor de la superclase, se lo debería hacer en forma explícita, añadiendo como primera línea de código del constructor de la subclase la instrucción:

```
super(argumentos);
```

Los argumentos son los parámetros que utiliza el constructor de la superclase. De esta manera el constructor de la subclase puede pasarle al constructor de la superclase los datos necesarios para la inicialización de los atributos privados, que no son accesibles desde la subclase.

Volviendo nuevamente a nuestro ejemplo, comparemos cómo operan los dos constructores:

```
public CuentaCorriente(){
      sobregiro = 300.0;
}

public CuentaCorriente(Double saldoInicial){
      super(saldoInicial);
      sobregiro = 300.0;
```

El primer constructor simplemente se utiliza para inicializar el valor del atributo sobregiro, en tanto que el segundo constructor llama al constructor de la superclase Cuenta y le pasa el parámetro saldoInicial para que inicialice el saldo de la cuenta (atributo propio de la superclase). Veamos el código en la superclase:

```
public Cuenta(Double saldo){
            this.saldo = saldo;
      }
```

donde el constructor recibe el parámetro saldoInicial y lo asigna como valor al atributo privado saldo de la superclase Cuenta. Por tanto existe una forma de invocar a métodos y atributos propios de la superclase desde la clase derivada haciendo uso de la palabra reservada super, en tanto que los atributos propios de la clase derivada serán accedidos usando la palabra reservada *this*.

Métodos y atributos protegidos: Existe un modificador de acceso aplicable a atributos y métodos de una clase pensado para ser utilizado con el manejo de herencia: el modificador ***protected.*** Este modificador de acceso permite que un miembro de una clase (atributo o método) sea accesible desde cualquier subclase dependiente de ésta, independientemente de los paquetes en que esas clases se encuentren.

Clases finales: Si queremos evitar que una clase sea heredada por otra, debemos declararla como clase final, colocando el modificador final antes de class, de esta manera:

```
public final class ClaseA{
}
```

Si otra clase intenta heredar una clase final se producirá un error de compilación

```
Public class ClaseB extends ClaseA{
}
```

Sobreescritura de métodos

Cuando una clase hereda a otra puede suceder que el comportamiento de los métodos que hereda no se ajuste a las necesidades de la nueva clase. En este caso, la subclase puede reescribir el método heredado, lo que se conoce como sobreescritura de un método. Con una salvedad: cuando se sobreescribe el método de una subclase, ésta debe tener exactamente el mismo formato que el método de la superclase que sobreescribe. O sea, que deben llamarse igual, tener los mismos parámetros y el mismo tipo de devolución. Veamos qué sucede en nuestro ejemplo:

En a superclase Cuenta:

```
public Double extraerDinero(Double importeARetirar){
            Double importeRetirado;

            if(importeARetirar<=this.saldo){
                  saldo-=importeARetirar;
                  importeRetirado = importeARetirar;
            }
```

```
                else{
                        importeRetirado = 0.0D;
                }

                return importeRetirado;
        }
```

En la clase derivada CuentaCorriente:

```
public Double extraerDinero(Double importeARetirar){
                Double importeRetirado;

                if(importeARetirar<=super.getSaldo()){
                        importeRetirado =
super.extraerDinero(importeARetirar);
                }
                else if(importeARetirar<=(super.getSaldo() +
this.sobregiro)){
                        sobregiro -= (importeARetirar - super.getSaldo());
                        super.extraerDinero(super.getSaldo());
                        importeRetirado = importeARetirar;
                }
                else{
                        importeRetirado = 0.0;
                }
                return importeRetirado;
        }
}
```

En caso que al sobreescribir un método de una subclase manteniendo el mismo nombre, pero modificando los parámetros, el nuevo método no sobreescribe el de la superclase, pero tampoco se produce un error de compilación, dado que nos encontramos ante un caso de sobrecarga de métodos: dos métodos con el mismo nombre y distintos parámetros.

El método sobreescrito puede tener un modificador de acceso menos restrictivo que el de la superclase. Por ejemplo, un método de la superclase puede ser protected y la versión sobreescrita de la subclase puede ser public (pero nunca uno más restrictivo).

Métodos y campos estáticos

Una variable *static* representa información en toda la clase (todos los objetos de la clase comparten el mismo dato). La declaración de una variable static comienza con la palabra clave *static*. Ahora bien, un método declarado como *static* no puede tener acceso a los miembros no *static* de una clase, ya que un método *static* puede llamarse aun cuando no se hayan creado instancias de objetos de la clase. Por la misma razón, esta referencia *this* no puede usarse en un método *static*; debe referirse a un objeto específico de la clase, y a la hora de llamar a un método *static*, podría no

haber objetos de su clase en la memoria. La referencia *this* se requiere para permitir a un método de una clase acceder a otros miembros no *static* de la misma clase.

La clase **Math** cuenta con métodos *static* para realizar cálculos matemáticos comunes; además, declara dos campos que representan constantes matemáticas de uso común: Math.PI y Math.E. La constante Math.PI (3.14159265358979323846) es la relación entre la circunferencia de un círculo y su diámetro. La constante Math.E (2.7182818284590452354) es el valor de la base para los logaritmos naturales (que se calculan con el método *static* Math log). Math.PI y Math.E se declaran con los modificadores *public*, *final* y *static*. Al hacerlos *public*, otros programadores pueden usar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave *final* es constante; su valor no se puede modificar una vez que se inicializa el campo. Tanto PI como E se declaran *final,* ya que sus valores nunca cambian. Al hacer a estos campos *static*, se puede acceder a ellos a través del nombre de la clase **Math** y un separador punto (.), justo igual que con los métodos de la clase **Math**.

La sintaxis para incluir atributos o métodos estáticos en una clase es la siguiente:

```
static tipo campon;
static tipo métodoX(parámetros)
      {
            //codigo métodoX
      }
```

A diferencia de los atributos en general que suelen ser de tipo privados, los atributos estáticos suelen llevar el modificador de acceso *public(*o *protected* o ninguno) lo cual permite que puedan ser accedidos por afuera de la clase. **Dado que el método estático no hace referencia a ningún objeto en particular, no puede hacer referencia a campos y métodos que si dependan de un objeto.**

A continuación se detalla el código de la clase Alumno donde puede observarse un atributo de tipo estático llamado CANTIDAD_DE_CLASES_BRINDADAS, inicializado en 0 y una constante de tipo estático llamada CANTIDAD_DIAS_DE_CLASE, inicializada en 20. Mientras el primero es de tipo *private*, la constante estática es de tipo *public*, por lo cual puede ser accedida en forma directa desde fuera de la clase Alumno. Luego puede observarse es el método habilitarDiaDeClase, de tipo estático y público que permite incrementar la variable CANTIDAD_DE_CLASES_BRINDADAS. Es importante observar que para llamar a la variable no se utiliza la palabra reservada *this* sino a través del nombre de la clase, por los motivos que se explicaron en párrafos anteriores.

```
public class Alumno {

      private static int CANTIDAD_DE_CLASES_BRINDADAS = 0;
      public static final int CANTIDAD_DIAS_DE_CLASES = 20;

public static void habilitarDiaDeClase(){
            Alumno.CANTIDAD_DE_CLASES_BRINDADAS++;
}
```