Tema 09. Herencia y polimorfismo

Contenido

1 Introducción

Puede suceder que entre algunas clases existan relaciones, por ejemplo, los libros y las revistas tienen propiedades que son comunes, o los distintos tipos de cuentas bancarias que existen, tiene propiedades que son comunes a todas.

¿Podríamos crear una clase a partir de otra?

La **herencia** consiste en crear una clase, a la que se llama clase **hija** o **subclase** o **derivada**, partiendo de otra ya creada anteriormente, que recibe el nombre de clase **padre** o **base** o **superclase**.

La clase **hija** tiene todas las características de la clase **base** y además sus características propias; podrá acceder a los miembros de la clase **base** dependiendo de los permisos de visibilidad que tenga, es decir, la clase **hija** no puede acceder directamente a los atributos privados ni usar los métodos privados declarados en la clase padre. Lo que es común a ambas clases queda comprendido en la clase **base** y lo que es específico, queda restringido a la clase **hija**. En un proyecto puede haber varias clases que hereden de la misma clase.

Con esto permitimos:

- La reutilización del código.
- Añadir nuevos datos y métodos a la clase.
- Modificar el código de algún método de la clase heredada.
- Crear una jerarquía de clases.

Existen dos tipos de herencia:

- **Herencia simple**: implica que se puede crear una nueva clase, partiendo de una sola clase ya existente, no se puede heredar de más de una clase.
- **Herencia múltiple**: se puede crear una nueva clase, partiendo de varias clases ya existentes, se puede heredar de varias clases.

Java solo permite la herencia simple.

Para que una subclase o clase hija pueda heredar todas las propiedades (atributos) y métodos de otra clase (superclase o clase padre) hay que añadir la palabra **extends** seguida del nombre de la clase padre como se expone a continuación:

```
class ClaseHija extends ClasePadre {
……
}
```

`extends` especifica que la clase hija hereda datos y métodos de la clase padre. Aunque hereda todo, la clase hija solo tiene acceso a los miembros de la clase padre que los permisos de visibilidad le permitan.

La clase **`java.lang.Object`** es la base de toda la jerarquía de clases de Java, es decir, toda clase que no hereda de ninguna clase, hereda por defecto de la clase **Object**. Dicha clase proporciona métodos como `toString()`, `equals()`, `getClass()`, etc.

Ejemplo1

Ejercicio 1

2 Superclases y subclases

La subclase o **clase derivada** está formada por todos los métodos y atributos que tenga además de los atributos y métodos que tenga la superclase o **clase base**.

En el *ejemplo 1*, la clase base es **Empleado** y la clase derivada es **Jefe**.

Las características de las clases derivadas son:

1. Una clase derivada puede, a su vez, ser una clase base, dando lugar a una jerarquía de clases. Con lo cual, todos los miembros de una clase derivada (incluidos los heredados) pueden ser heredados por otra clase derivada de ella.
2. Una clase derivada hereda todos los miembros de la clase base y puede acceder a los miembros que sean públicos, protegidos y package (si la subclase está en el mismo paquete que la superclase).
3. Una clase derivada puede añadir a los miembros heredados sus propios datos y métodos.

Ejemplo 2

Escribe un programa en Java que permita:

- Pedir datos de seres vivos
- Visualizar dichos datos.
- Alimentarlos

El programa tendrá que utilizar 4 clases:

- **La clase Principal**. Esta clase contiene el método main, que es por donde va a arrancar y cerrar el problema.
- **La clase Animal**. De los animales nos interesa conocer el número de crías que tienen. Esta clase tendrá métodos para:
  - Pedir sus datos.
  - Visualizar sus datos.
  - Alimentar al animal.
- **La clase Mamifero**. Contiene datos específicos como el número de huesos que tiene y el número de extremidades. Hay que escribir los métodos necesarios para manejar sus datos exclusivos (como mamífero).
- **La clase Persona**. Contiene datos exclusivos del ser humano como, por ejemplo, su profesión. Hay que escribir los métodos necesarios para manejar sus datos exclusivos (como persona).

En este ejemplo la clase Persona hereda de la clase Mamífero y la clase Mamífero hereda de la clase Animal.

3 Constructores y herencia

Cuando se crea un objeto de una subclase, primero se ejecuta la función constructora de la superclase y, cuando esta finaliza, se ejecuta la función constructora de la subclase.

Vamos a observar un ejemplo muy sencillo para ver cómo se va llamando a las funciones constructoras de las clases que hereda.

Ejemplo3

Ejercicio 2

Instrucción super

Si en el constructor de la clase hija no llamamos explícitamente al constructor de la clase padre, mediante *super* en Java, entonces el compilador llama automáticamente al constructor por defecto (es decir, sin argumentos) de la clase padre. En este caso, si la clase padre no tiene constructor por defecto, obtendremos un error de compilación. En Java, además del método especial *super*, se puede utilizar la función ***this*** para llamar desde un constructor a otro constructor de la propia clase.

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

      //Constructor por defecto que usa el método especial this() para llamar
      // al constructor con 3 argumentos de la propia clase Persona
      public Persona () {
            this("Fulanito","Lopez Perez",22);
    }

      //Constructor con 3 parámetros
      public Persona (String nombreNuevo, String apellidosNuevos, int edadNueva) {
            nombre = nombreNuevo;
            apellidos = apellidosNuevos;
            edad = edadNueva;
     }
}
```

La instrucción "**super**" dentro de un constructor de una subclase llama al constructor de la superclase de manera explícita. Si no hay que pasarle parámetros no es necesario llamar al constructor de la superclase con **super** ya que se ejecuta automáticamente; hay que invocarlo de forma explícita cuando el constructor de la clase padre espera que le pasemos argumentos.

La sentencia que llama al constructor de la clase padre tiene que ser la primera dentro del constructor de la subclase. Esta llamada explícita al constructor de la clase heredada no conlleva el que se vaya ejecutar dichas funciones dos veces. La llamada se hace del siguiente modo:

```
super(y dentro de estos paréntesis ponemos los datos que le queremos
enviar a la función constructora de la superclase)
```

Ejemplo 4 y Ejemplo 5

4 Modificadores en clases, atributos y métodos

4.1 Modificadores de la clase

Los modificadores de la clase son:

- **public**: Como sabemos, es un modificador de acceso. Si a una clase le ponemos dicho modificador, estamos haciendo que dicha clase pueda ser usada fuera del paquete donde está definida. Si no se pone el modificador public, por defecto es ***package***; esto significa que la clase solo puede ser utilizada por las clases que están en el mismo paquete donde está definida.
- **abstract**: Se usa para indicar que una clase es abstracta. Esto significa que la clase puede contener métodos sin implementación (abstractos). Una clase abstracta sirve, exclusivamente, para que sea heredada y no se puede instanciar objetos con este tipo de clases.
- **final**: Cuando una clase tiene el modificador final, de dicha clase no pueden heredar otras clases.

4.2 Modificadores de los atributos y métodos

Modificadores de visibilidad

Todos los atributos y métodos de una clase son siempre accesibles para los métodos de la propia clase.

Existen una serie de modificadores de visibilidad con el fin de poder controlar el acceso de los miembros de una clase desde otras clases y desde las propias clases que heredan de ella:

- **public**: A los miembros de una clase, declarados como "public" se puede acceder a ellos desde cualquier otra clase, siempre que se tenga acceso a dicha clase.
- **private**: Los miembros declarados private son accesibles solo dentro de la propia clase.
- **protected**: Los miembros declarados protected son accesibles dentro de la propia clase, dentro de las clases del propio paquete donde está definida y dentro de la subclase, aunque no estén en el mismo paquete.
- **No se pone nada (package)**: Si no se pone ningún modificador de los anteriores, esto significa que solo pueden acceder a dichos miembros aquellas clases que estén en el mismo paquete que la clase donde están definidos dichos miembros. Se les conoce con el nombre de ***package*** o friendly. Si una clase hereda miembros de tipo package, podrá hacer uso de ellos siempre que las subclases y la superclase estén en el mismo paquete.

Ejercicio 3

Además de los modificadores de visibilidad, vamos a ver otros modificadores que pueden tener los atributos y métodos.

4.3 Modificadores exclusivos de los atributos

- **static**: Si un atributo de una clase es static, esto implica que el valor que contenga dicho atributo sea el mismo para todos los objetos que pertenezcan a esa clase. Si se cambia ese valor, se cambia para todos los objetos. Esto se debe a que dicho valor no está repetido para cada objeto, sino que existe una sola copia. Son ***atributos de la clase*** y no del objeto.
- **final**: Si un atributo es final, esto obliga a darle un valor inicial. Esta inicialización solo puede hacerse en dos sitios: (1) en el constructor de la clase o (2) en el momento de la declaración dentro de la clase. Si optamos por la vía del constructor, el compilador nos obligará a inicializar el atributo final en todos los constructores de la clase. Así pues, un atributo final actúa como una constante una vez se le ha asignado un valor. Dicho valor ya no se podrá cambiar a lo largo de la ejecución del programa. Es lo que se conoce como ***constante***.

4.4 Modificadores exclusivos de los métodos

- **abstract**: Es un método que ***no tiene cuerpo***.
- **static**: Son métodos a los que se puede acceder con el nombre de la clase, sin tener que usar ningún objeto de dicha clase. Suelen utilizarse para acceder a atributos estáticos. Se les conoce como ***métodos de la clase***.
- **final**: Si un método es final, implica que la clase que lo herede no podrá sobrescribirlo.
- **native**: Su cuerpo está codificado en otro lenguaje de programación.

Ejercicio 4

5 Sobrescritura de métodos

La sobrescritura de métodos existe cuando hay herencia de clases.

Cuando una clase hereda de otra, puede redefinir los métodos de la clase base solo con definir dentro de la subclase dichos métodos.

Las reglas para sobrescribir dichos métodos son:

- El tipo de dato que devuelven debe de ser el mismo para los dos métodos.
- El nombre del método debe ser el mismo.
- Los argumentos deben de ser los mismos y del mismo tipo. Si no coinciden, ya no hacen referencia al método de la clase base, sino que sería otro método distinto y estaríamos hablando de sobrecarga del método.
- El modificador de visibilidad del método de la subclase no puede ser más restrictivo que el de la clase base (por ejemplo, *protected* en la clase padre y *private* en la clase hija).
- Las modificaciones hechas en la clase hija afectan solo a la clase hija y a las clases que hereden de esta, en ningún caso afectan a la clase padre.
- En el caso de los métodos *static*, estos solo pueden ser sobrescritos si en la clase hija siguen siendo *static*.
- Las excepciones que lance el método de la subclase no pueden ser más que las que lanza el método de la clase base.

Hay que recordar que si un método tiene el modificador final, la clase que lo herede no podrá sobrescribir dicho método.

Por defecto, si dentro de la clase derivada llamamos al método sobrescrito, se ejecuta el método de la clase derivada. Si queremos ejecutar el método sobrescrito de la clase base, hay que usar la instrucción super:

```
super.metodoSobrescrito();
```

Ejemplo 6

Ejercicio 5

6 Clases y métodos abstractos y finales

6.1 Clases y métodos abstractos

Una clase puede heredar de otra clase o de una clase abstracta, o puede implementar una o varias interfaces.

Una clase abstracta tiene este tipo de cabecera:

```
abstract class ClaseAbstracta
```

Una clase abstracta se utiliza como superclase (con el fin de que otras clases hereden de ella); de esta manera, la clase que derive de ella está obligada a implementar los métodos abstractos que tenga dicha clase abstracta. Con esto se consigue fijar el comportamiento mínimo que tiene que tener una clase.

Un **método abstracto** es aquel que tiene el modificador abstract y **no tiene un cuerpo definido**. Es decir, solo tiene indicada la cabecera de dicho método.

La utilidad de un método abstracto es indicar cuál es la cabecera del método que va a tener que implementar cualquier clase que herede de la clase abstracta a la que pertenece.

Las propiedades de una clase abstracta son:

- Se pueden declarar objetos (referencias a ellos) con este tipo de clases, pero no se pueden instanciar.
- Una clase abstracta debería de tener, al menos, un método abstracto. Si no es así, no tendría sentido que dicha clase fuera abstracta.
- Una clase abstracta puede tener métodos que no sean abstractos. La clase que hereda dicha clase no está obligada a implementar los métodos que no sean abstractos.

Las propiedades de un método abstracto son:

- No pueden tener cuerpo.
- No pueden estar definidos en una clase que no sea abstracta.

6.2 Clases y métodos finales

Si queremos que de una clase no puedan heredar otras clases, hay que poner el modificador "final".

Si una clase tiene métodos con el modificador "final", implica que si una clase deriva de ella, dicha subclase no podrá sobrescribir dichos métodos.

7 Interfaces. Clases abstractas e interfaces.

Una interfaz y una clase abstracta son muy parecidas. La diferencia está en que **en una interfaz, todos los métodos tienen que ser abstractos**, es decir, estarán formados por las cabeceras de diferentes métodos (prototipos).

Con lo cual, una **interfaz**, al igual que las clases abstractas, sirve para definir el comportamiento **mínimo que tiene que tener una clase**.

Las **interfaces se declaran** usando la palabra clave `interface` en vez de `class`.

```
public interface Obligatorio {
    public void pedirTodosLosDatos();
    public void visualTodosLosDatos();
}
```

Las interfaces no son heredadas por una clase, sino que **una clase implementa una interfaz**. Esto se indica de esta forma:

```
class NombreClase implements NombreInterface
```

Al poner esto, se está obligando a que la clase "`NombreClase`" implemente todos los métodos que tenga dicha interfaz.

Por ejemplo:

Si una clase implementa la interfaz `Obligatorio` (definida anteriormente), estará obligada a implementar los métodos: `pedirTodosLosDatos()` y `visualTodosLosDatos()`.

```
class MiClase implements Obligatorio {
    void pedirTodosLosDatos() {
      <código>
    }
    void visualTodosLosDatos(){
      <código>
    }
}
```

Las interfaces siguen los principios de la herencia múltiple, ya que permite que una clase pueda implementar varias interfaces. Es decir, podemos escribir esto:

```
Class NombreClase implements NombreInterface1,
      NombreInterfae2, NombreInterface3,...
```

Con esto, estamos obligando a que la clase "NombreClase" implemente todos los métodos que tengan todas las interfaces indicadas.

Las propiedades de las interfaces son:

- Una interfaz no puede tener atributos salvo que sean estáticos (*static*) y constantes (*final*). Además, **en una interfaz los atributos deben ser públicos e inicializados con un valor en el momento de la declaración**.
- A los métodos de una interfaz no se les puede poner los modificadores **private** ni **protected**. Implícitamente, son **public**.
- Una interfaz no puede tener constructor.
- A los métodos de una interfaz no hace falta ponerles el modificador **abstract**; porque implícitamente lo son.
- Dentro de la clase que implementa una interfaz, todos los métodos que sobrescribe de dicha interfaz tienen que ser públicos.
- Se puede declarar un objeto con el nombre de una interfaz, pero no se puede instanciar indicando el nombre de la interfaz. De este modo, si se define una atributo/variable cuyo tipo es una interfaz, se le puede asignar un objeto que sea una instancia de una clase que implementa la interfaz. Esto significa que, utilizando interfaces como tipos, se puede aplicar el mecanismo de polimorfismo a clases que no están relacionadas por el mecanismo de herencia, pero sí por el mecanismo de implementación de una interfaz.
- Una interfaz puede heredar de otra interfaz. En Java y C#, debido a la restricción de la herencia simple, solo podrá heredar de una interfaz:

```
public interface Interfaz2 extends Interfaz1 {
    //Firmas de los métodos
}
```

8 Polimorfismo

8.1 Concepto

Es el mecanismo que permite que un objeto de una clase se comporte como un objeto de cualquiera de sus subclases (o subclases de las subclases y así sucesivamente) porque una variable que apunta a un objeto de una clase puede apuntar a objetos de sus clases derivadas.

El polimorfismo se da en tiempo de ejecución (ligadura dinámica). Esto sucede cuando una subclase sobrescribe algún método de la clase padre; esto hará que una misma llamada a un método conlleve la ejecución de métodos diferentes, dependiendo del objeto que llame a dicho método.

Ejemplo

La clase **Empleado** y la clase **Jefe** (que es hija de la clase Empleado), si dentro de la clase Jefe hemos sobrescrito métodos de la clase Empleado, cuando hagamos, por ejemplo, estas instanciaciones de objetos:

```
Empleado emp1 =new Jefe();
```

```
Empleado emp2=new Empleado();
```

Al hacer estas llamadas al mismo método:

```
emp1.metodosobrescrito();
emp2.metodosobrescrito();
```

El método que se va a ejecutar va a ser diferente. Esto se debe a que se ejecuta el método asociado a la clase que se indicó en la instanciación y no el método asociado a la clase que se indicó en la declaración de la referencia al objeto. Esto es lo que se le llama **ligadura dinámica**.

Por ello, en el ejemplo anterior, el objeto **emp1** ejecutará el método que hay en la clase Jefe y el objeto **emp2** ejecutará el método que hay en la clase Empleado.

Con lo cual, para que exista polimorfismo se tienen que dar tres condiciones:

- Que haya herencia.
- Que haya sobrescritura de métodos.
- Definir un objeto con una superclase e instanciarlo con una subclase:
  `Empleado emp=new Jefe();`

Ejemplo 7

8.2 Comprobación estática y dinámica de tipos

Las comprobaciones estáticas son las que realiza el propio compilador al analizar el código fuente.

Ejemplo:

Si tenemos un objeto como este: `Empleado emp= new Empleado();`

Y ponemos: `emp.visualizar();`

El compilador comprueba que haya dentro de la clase Empleado un método con el nombre: `visualizar()`. Si no es así, dará un error de compilación.

Las comprobaciones dinámicas son las que se realizan en tiempo de ejecución.

Por ejemplo:

Si tenemos esta función:

```
public static void visualizarTabla(Object empresa[]) {
      Empleado emp;
      for(int pos=0; pos<empresa.length; pos++) {
            emp=(Empleado)empresa[pos];
            emp.visualTodo();
      }
}
```

El compilador no puede comprobar en el momento de la compilación si los objetos que va a recibir este método serán de tipo Empleado. Esta comprobación la hace en el momento de la ejecución del programa, de tal forma que si este método recibiera una tabla con información de

alumnos, se produciría un error de ejecución al hacer la comprobación dinámica y ver que no puede convertir un objeto de tipo Alumno a Empleado. Este sería el mensaje que lanzaría el programa al producirse ese error de ejecución:

```
Exception in thread "main" java.lang.ClassCastException: Alumno cannot
be cast to Empleado
```

Para que estos errores-no se produzcan, habría que añadir a dicha función la siguiente comprobación:

```
public static void visualizarTabla(Object empresa[]) {
    Empleado emp;
    boolean continuar=true;
    for(int pos=0;pos<empresa.length && continuar;pos++) {
          if(empresa[pos] instanceof Empleado) {
                emp=(Empleado)empresa[pos];
                emp.visualTodo();
          } else
                continuar=false;
    }
    if(continuar==false) {
          System.out.println("\n\nSe ha suspendido la visualización de datos.");
          System.out.println("\nEsto se debe a que la tabla tiene datos, que
                                no son de tipo Empleado");
    }
}
```

Ejercicio

9 Conversiones de tipos entre objetos (casting)

El concepto de *casting, también llamado cast* o *narrowing*, está íntimamente relacionado con el polimorfismo y, por tanto, también con la herencia.

Se puede definir como el proceso en el que el compilador permite hacer una conversión de un objeto polimórfico a uno de sus posibles tipos dinámicos o, dicho de otra manera: si se desea acceder a los métodos de la clase hija teniendo una referencia de una clase padre, hay que convertir explícitamente la referencia de clase padre a clase hija. Esta conversión es lo que se denomina *casting*.

Por ejemplo:

Si la clase Jefe es hija de la clase Empleado, podemos poner:

```
Empleado e=new Jefe();
```

Ya que todo `jefe` es un `empleado`.

Con lo cual, cuando instanciamos un objeto, se admite la referencia de una clase del mismo tipo (de la que es el objeto) o de una clase superior.

Pero esto no sucede al revés.

```
Jefe j=new Empleado();
```

Ya que `Empleado` no engloba la clase `Jefe`. Dicho de otra forma: "no todos los empleados son jefes".

Por esta misma razón, si hay un parámetro que es de un tipo determinado, permite que el argumento que reciba sea del mismo tipo o superior. Por ejemplo, si la clase **Jefe** hereda de la clase **Empleado** y también la clase **Becario** hereda de la clase **Empleado**, y tenemos un método con esta cabecera:

```
void metodo(Empleado e)
```

Dicho método admite las siguientes llamadas:

- `metodo(objetoTipoJefe)`
- `metodo(objetoTipoBecario)`
- `metodo(objetoTipoEmpleado)`

Ya que dicho método quiere un empleado; un jefe o un becario lo son.

Luego, dentro del método, para saber de qué tipo es el objeto que ha recibido, podemos usar el operador **instanceof**.

Por ejemplo:

```
void metodo(Empleado e) {
    if(e instanceof Becario)
          System.out.println("Se ha enviado un objeto de tipo Becario");
    else
          if(e instanceof Jefe)
                System.out.println("Se ha enviado un objeto de tipo Jefe");
          else
                System.out.println("Se ha enviado un objeto de tipo Empleado");
 }
```

En el método anterior, independientemente del tipo que sea el objeto que se recibe, solo se podrán usar los métodos o propiedades de la clase Empleado. Para poder usar todas las propiedades o métodos del objeto que se envía, habría que hacer una conversión de tipo (casting). Es decir, habría que hacer esto:

```
void metodo(Empleado e) {
  if(e instanceof Becario) {
        System.out.println("Se ha enviado un objeto de tipo Becario");
        Becario eb;
        eb= (Becario)e;
  } else {
        if(e instanceof Jefe) {
```

```
                System.out.println("Se ha enviado un objeto de tipo Jefe");
                Jefe ej;
                ej=(Jefe)e;
            } else {
                System.out.println("Se ha enviado un objeto de tipo Empleado");
            }
    }
```

Ten en cuenta que todo becario y jefe es un empleado, por lo que si en el primer if ponemos:

```
    if(e instanceof Empleado)
```

siempre pasaría por ahí.

10 Clases y tipos genéricos o parametrizados

Un tipo genérico es como un tipo comodín, ya que se puede convertir en cualquier tipo.

Por ejemplo, si tenemos este método:

```
public static <T> boolean tamañoArrayPar(T array[]) {
      int tam=array.length;
      if(tam%2==0)
            return true;
      else
            return false;
}
```

Al poner "**<T>**" estamos indicando que dicho método va a utilizar un tipo genérico. Esto consiste en que dicha "T" se va a convertir en el tipo que se envíe como argumento.

Así, si tenemos este código:

```
  public static void main(String args[]) {
      boolean cierto;
      Alumno clase[]=new Alumno[4];
      Empleado vecEmp[]=new Empleado[5];
      cierto=tamañoArrayPar(clase);
      cierto=tamañoArrayPar(vecEmp);
  }
```

En la primera llamada al método, trabajaría con una tabla de tipo Alumno y en la segunda, con una tabla de tipo Empleado. Hemos conseguido trabajar con un método que admite cualquier tipo.

Si queremos usar los genéricos en una clase con el fin de conseguir que admita cualquier tipo de datos, habría que hacerlo de esta forma:

```
      public class ClaseGenerica<T> {
            T tabla[];
            public void recibirTabla(T tab[]) {
                  tabla=tab;
            }
            public T devolDatPosTabla(int pos) {
                  return tabla[pos];
```

```
        }
    }
```

Para hacer uso de esta clase habría que hacerlo así:

```
ClaseGenerica<Empleado> empresa=new ClaseGenerica<Empleado>();
```

Así, dentro de la clase donde se puso T se convertiría automáticamente en el tipo Empleado.

También podemos limitar los tipos con los que se puede parametrizar nuestra clase. Por ejemplo, si queremos una clase que sume dos números, no tendría sentido poder parametrizar nuestra clase con el tipo *String* (que es texto). Para hacerlo usaremos la palabra reservada ***extends***:

```
public class Suma<T1 extends Number, T2 extends Number>{ ... }
```

y desde el programa principal podríamos hacer:

```
Suma suma1=new Suma<Integer, Double>(2, 6.5);
```

pero no:

```
Suma suma1=new Suma<Double, String>(6.5, "8");
```