TEMA 5: HERENCIA SIMPLE

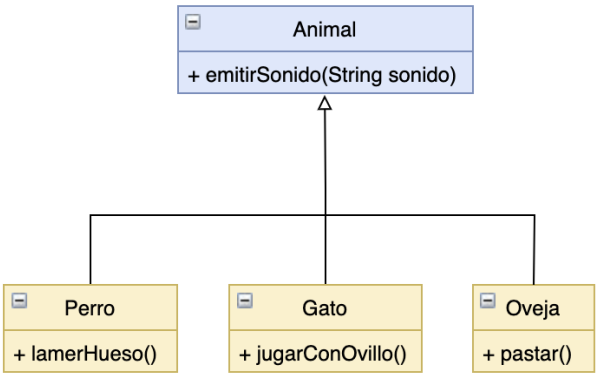
Ejemplo animales sin herencia:

- ➢ Crear un proyecto java con el nombre Animales
- ➢ Crear una clase Perro con un método emitirSonido() que imprime por consola "guau" y un método lamerHueso() que imprime por consola "lamiendo hueso"
- ➢ Crear una clase Gato con un método emitirSonido() que imprime por consola "miau" y un método jugarConOvillo() que imprime por consola "juego con el ovillo "
- ➢ Crear una clase Oveja con un método emitirSonido() que imprime por consola "bee" y un método pastar() que imprime por consola "pastando hierba"
- ➢ Crear una clase de prueba. En el main de dicha clase crear un objeto de tipo Perro, otro de tipo Gato y otro de tipo Oveja. Hacer que cada objeto invoque al método emitirSonido()

Observamos que el método emitirSonido() es común a las clases Perro, Gato y Oveja; y que luego cada clase tiene un método específico (lamerHueso() de la clase Perro, jugarConOvillo() de la clase Gato y pastar() de la clase Oveja).

La herencia nos permite implementar una clase (clase hija) a partir de otra clase (clase padre) de forma que:

→ La clase hija incluya el código de la clase padre
→ La clase padre es conceptualmente una clase más general y la clase hija es más especializada



Animal es la clase padre (o **superclase**) de las clases Perro, Gato y Oveja
La clase Perro es una clase hija (o **subclase** o clase derivada) de la clase Animal

Una clase hija es una correcta subclase de una clase padre si cumple:
(1) **Regla Es-Un**: un objeto de la clase hija es también un objeto de la clase padre
(2) **Principio de sustitución**: los objetos de la clase hija pueden sustituir a los de la clase padre cuando se les pide realizar una operación

Podemos definir una clase como clase derivada o subclase de una clase padre de la siguiente forma:

```
class ClaseHija extends ClasePadre{
     ..........
}
```

NOTA: cada clase hija solo puede tener UNA única clase padre (no existe herencia múltiple)

Para pasar los argumentos desde los constructores de las clases hijas al constructor de la clase padre utilizamos `super(atributo1, atributo2, ...)`

Cuando se crea un objeto de una clase hija, se ejecutan los constructores siguiendo el orden de derivación, es decir, primero se ejecuta el constructor de la clase padre, y después los de las clases derivadas (hijas). Hay dos opciones:

→ O tiene que estar el constructor por defecto
→ O se usa `super`

Si el constructor de la clase padre tiene argumentos para dar valor a los atributos del padre, el constructor de la clase hija le pasará dichos valores mediante `super`

Para comprobar desde la clase padre si un objeto pasado como argumento es del tipo de alguna de sus subclases debemos usar `instanceof`

La principal ventaja de usar herencia es que nos permite reutilizar código. De esta forma…
(1) escribimos menos código,
(2) si queremos modificar el método, basta con hacerlo en la clase padre en lugar de ir modificándolo en todas las clases hijas, y
(3) al haber menos líneas de código, habrá menos errores.

Ejemplo animales con herencia:

- Creamos otro proyecto HerenciaAnimales
- Modificamos las clases del proyecto anterior de manera que aprovechen el mecanismo de la herencia.
- Implementamos la jerarquía de clases anterior (*figura anterior*).
- Queremos que los objetos de tipo Perro, Gato y Oveja tengan como atributos nombre y edad.
- Queremos crear un constructor que nos permita inicializar los objetos de tipo Perro, Gato y Oveja.
- Queremos añadir los siguientes atributos: (1) a la clase Oveja le vamos a añadir un atributo pastor de tipo String y (2) a la clase Perro le vamos a añadir un atributo amo de tipo String.
- Queremos modificar el método emitirSonido (String sonido) para que además del sonido, imprima también el nombre del animal.

- **Problema**: con un objeto de tipo Perro se puede llamar al método emitirSonido("bee")
- **Ejercicio**: modificar el código del ejemplo para que esto no ocurra
- **Idea**: Todos los animales emiten algún sonido, pero el sonido es distinto para cada animal
  Por defecto, todos los animales del mismo tipo van a emitir el mismo sonido: guau, bee, etc., pero se podría cambiar con un setter para el atributo sonido.

Se conoce como **upcasting** a guardar en una variable de tipo Padre la referencia a un objeto de tipo Hija; mientras que se conoce como **downcasting** a asignar a una variable de tipo Hija la referencia casteada (Hija) de una variable tipo Padre.

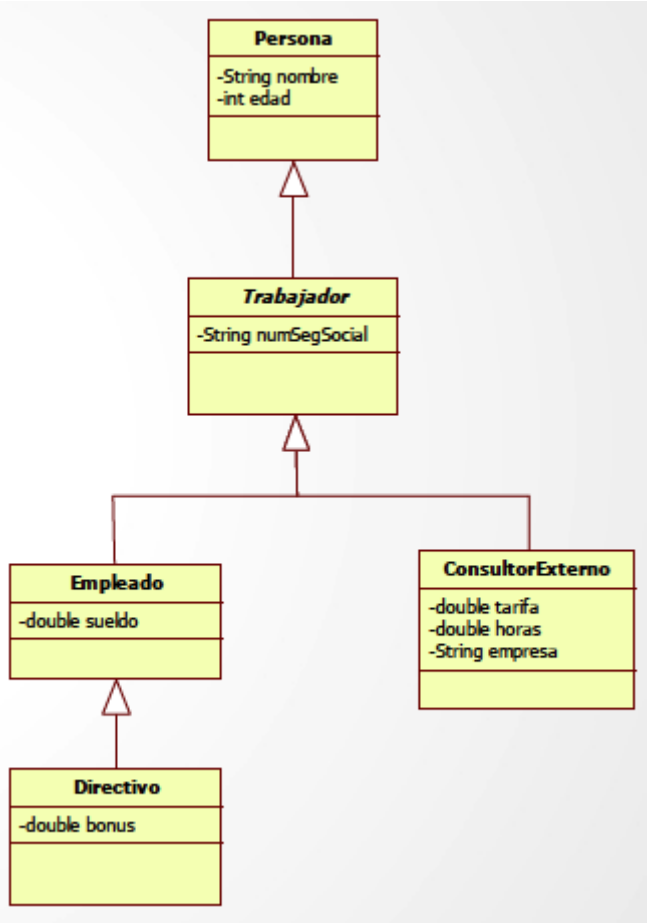
Ejemplos *upcasting* y *downcasting*:

```
Padre p1 = new Hija();          // UPCASTING
Hija h1 = (Hija) p1;            // DOWNCASTING → sí permitido

Padre p2 = new Padre();
Hija h2 = (Hija) p2;            // DOWNCASTING → ¡¡NO permitido!!
```

Ejercicio:

- Crear una clase Persona con nombre(String) y edad (int)
  - Implementar el constructor que inicialice el nombre y la edad con parámetros de entrada
- Crear una clase Trabajador que herede de Persona y tenga un número de la seguridad social (String)
  - Implementar el constructor con todos los atributos de la clase con parámetros de entrada
- Crear una clase Empleado que herede de Trabajador y tenga un atributo sueldo (double)
  - Implementar el constructor con todos los atributos de la clase con parámetros de entrada
- Crear una clase Directivo que herede de Empleado y tenga el atributo (double) bonus
  - Implementar el constructor con todos los atributos de la clase con parámetros de entrada
- Crear una clase ConsultorExterno que herede de Trabajador y tenga tarifa (double) y horas (double) y una empresa (String)
  - Implementar el constructor con todos los atributos de la clase con parámetros de entrada



TEMA 5: HERENCIA CON SOBRESCRITURA (POLIMORFISMO)

1. SOBRESCRITURA

Cuando queremos cambiar o completar el comportamiento de un método de la clase padre en una de sus clases hijas, estamos **sobrescribiendo** (o **redefiniendo**) dicho método. El método de la clase hija puede usar la implementación dada por el padre (super) o cualquier otro método accesible, tanto del padre como de sí mismo.

Ejemplo:

```
public class Animal {
    private String nombre;
    private int edad;
    private String sonido;
    ..................
    public String toString() {
      return "Nombre: "+ this.nombre+" Edad: "+ this.edad+
            " Sonido: " +this.sonido;
    }
}

public class Perro extends Animal {
    private String amo;
    ..................
    public String toString() {    // Sobrescribimos el método toString() de la clase padre
      return super.toString() + " Amo: " + this.amo; // Usamos la clase padre
    }
}
```

Ejercicio continuación:

Dadas las clases del ejercicio del tema anterior (Persona, Trabajador, Empleado, ConsultorExterno, Directivo), añadir un método toString que devuelva los atributos del objeto. Al redefinir el método toString de una clase hija, se debe aprovechar el método toString() de la clase padre.

2. POLIMORFISMO

El **polimorfismo** en POO se da por el uso de la herencia y se produce por distintas implementaciones de los métodos definidos en la clase padre (sobrescritura), es decir, por distintas implementaciones entre clase padre e hija y/o por distintas implementaciones entre clases hija.

Una misma llamada ejecuta distintas sentencias dependiendo de la clase a la que pertenezca el objeto. El código a ejecutar se determina en tiempo de ejecución → **Enlace dinámico**

Ejemplo:
Supongamos que tenemos una granja donde hay perros, ovejas y otros animales y que queremos un programa que nos imprima su nombre, edad, sonido y, en el caso de los perros su dueño y en el de las ovejas el nombre de su pastor. Para ello creamos un vector que contenga todos los objetos de tipo Animal, de tipo Perro y de tipo Oveja, e imprimimos los valores de los atributos de esos objetos llamando al método toString(). → Necesitamos un vector de objetos de tipo Animal (**clase Padre**)

Ejercicio:
Dadas las clases del ejercicio anterior, implementar un *main* que haga lo siguiente:
→ Crear un vector de 4 Personas que contenga 1 directivo, 2 empleados y 1 consultor externo
→ Escribir un bucle que muestre los objetos del vector de personas en la pantalla

En Java si ponemos el modificador **final** delante de un **atributo**	→	no puede modificarse una vez inicializado (es decir, se convierte en una *constante*)
En Java si ponemos el modificador **final** delante de un **método**	→	no puede *sobrescribirse*
En Java si ponemos el modificador **final** delante de una **clase**	→	no puede tener *clases hijas*

En Java todas las clases heredan de la **clase Object** de manera implícita (no hace falta poner "extends Object")

Algunos métodos como **equals**() y **toString**() están en la **clase Object** y se pueden sobrescribir en cualquier clase.

TEMA 5: HERENCIA ABSTRACTA

Los **métodos abstractos** sirven para declarar métodos que son aplicables a una clase padre pero cuya implementación depende de cada clase hija. Para declarar un método como abstracto, se pone delante la palabra reservada *abstract* y no se define un cuerpo: ***abstract*** *tipo nombreMetodo(...);* Luego, en cada subclase se define un método con la misma cabecera y distinto cuerpo.

Si una clase contiene al menos un método abstracto, entonces es una clase abstracta. Una clase abstracta es una clase de la que no se pueden crear objetos, pero puede ser utilizada como clase padre para otras clases. Se declara como ***abstract class*** *NombreClase {...}*

Ejercicio1:
Crear la clase Poligono y las clases hijas Triangulo y Rectangulo. Crear un método que clasifique el polígono, es decir, si es un triángulo, debe indicar si es equilátero, isósceles o escaleno; mientras que si es un rectángulo, debe indicar si es un cuadrado o no.

La implementación de dicho método es distinta dependiendo de si el objeto es un triángulo o un rectángulo; por lo que tendríamos un método getTipo() aplicable a todos los objetos de la clase Poligono, cuya implementación será diferente en cada subclase.