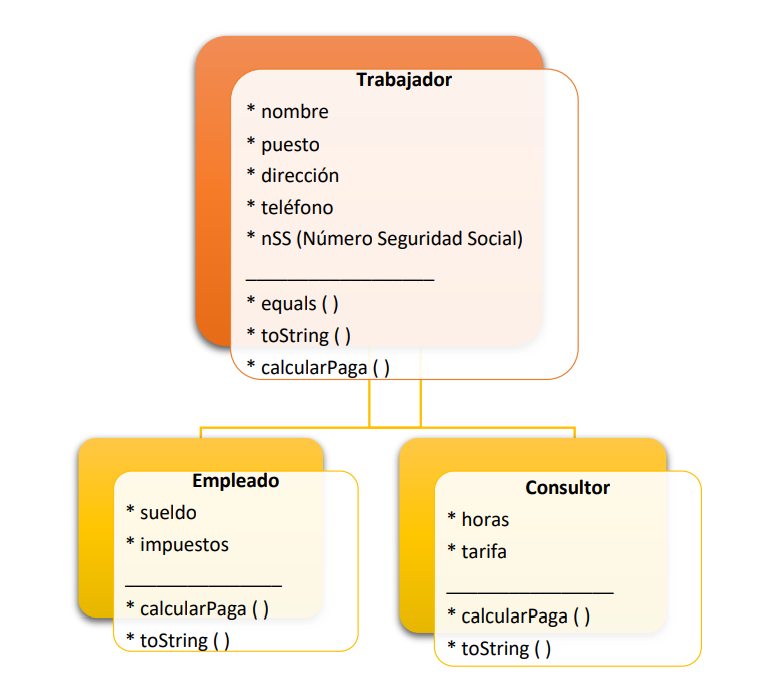
**APUNTES PROGRAMACIÓN UD4: Programación Orientada a Objetos II (HERENCIAS)**

**Superclase (Clase madre):** clase de la que se heredan sus atributos.

**Subclase (Clase hija):** clase que hereda los atributos de una clase superior.

**Una subclase deriva de una superclase.**

Este sería un ejemplo clásico de jerarquías de clases:



La clase Trabajador sería la superclase y las clases Empleado y Consultor serían subclases.

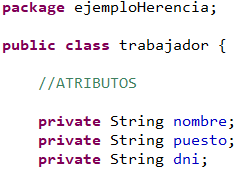
**ESTABLECER UNA HERENCIA**

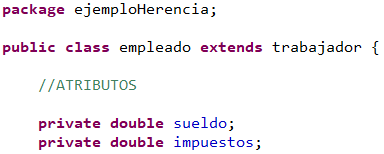
**Las subclases heredan atributos y métodos de la superclase PERO NO HEREDAN LOS CONSTRUCTORES.**

**La subclase también puede poseer atributos y métodos que no existan en la original.**

**UNA SUPERCLASE NO PUEDE HEREDAR NADA DE UNA SUBCLASE.**

Esta vez tenemos la superclase clase Trabajador y la subclase Empleado, cada una con los siguientes atributos:



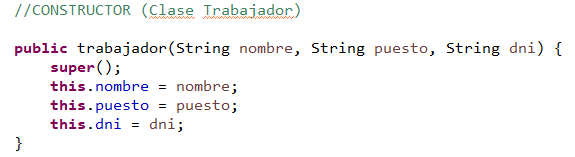


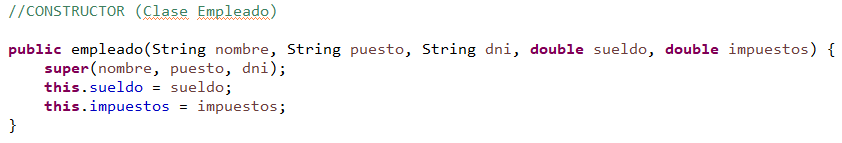
Para indicar herencia se usa la palabra **extends** en la sentencia de la mención de la subclase seguido del nombre de la superclase, esto indica que la clase Empleado deriva de la clase Trabajador.

**public class NombreSubclase extends NombreSuperclase {**

**public class Empleado extends Trabajador {**

A la hora de generar los constructores de ambas clases, se mostrarán de la siguiente forma:





Si bien en el constructor de la clase Trabajador no se observa nada raro, en el constructor de la clase Empleado podemos observar que están inscritos como parámetros los atributos de la clase Trabajador además de los suyos propios. Y también el **super** contiene los atributos de la clase Trabajador.

Esto ocurre porque la clase Empleado hereda de la clase Trabajador.

Con **super ()**, estamos haciendo una llamada al constructor de la clase madre, es decir, llamando al constructor de la clase Trabajador que se encarga de inicializar los atributos de esa clase.

Con todo ello, ya hemos establecido una herencia.

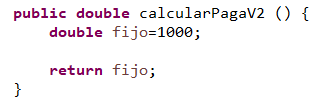
Seguido generamos los getters y setters y los “To String” de ambas clases.

**MÉTODOS CON EL MISMO NOMBRE**

**LLAMAR A UN MÉTODO DENTRO DE OTRO MÉTODO**

Vamos a calcular la paga de los trabajadores, empleados y consultores.

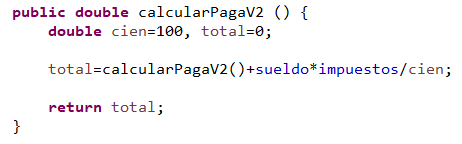
Para ello vamos a crear un método llamado “calcularPagaV2” en la clase Trabajadores y suponemos que ganan un sueldo fijo de 1000 €.



Y ahora vamos a crear el método “calcularPagaV2” en la clase Empleados.

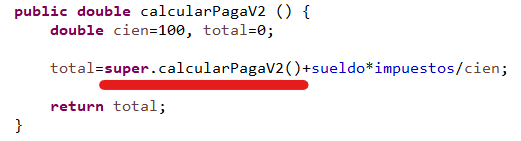
El sueldo de los empleados se calcula su…

Como sueldo fijo ya lo tenemos en el método “calcularPagaV2” de la clase Trabajador, lo llamamos en vez de instaurar otra variable para el sueldo fijo.



Bien, ya está hecho… Pero si los métodos se llaman igual ¿cómo hacemos para indicar que nos referimos al método de la clase madre?

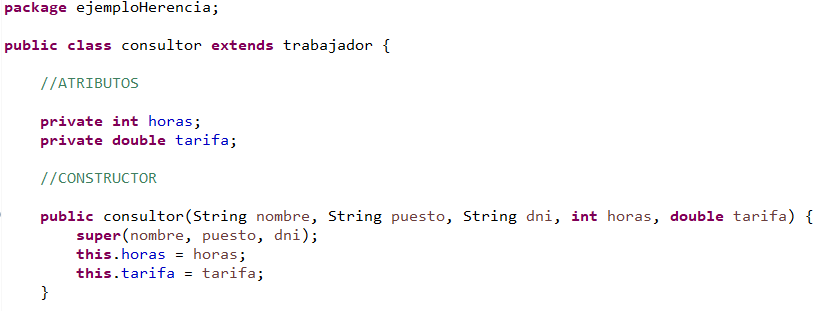
Pues con la expresión **super.** seguido del nombre del método que al que queremos referirnos.

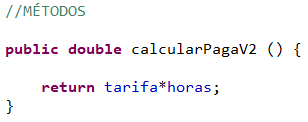


Ahora ese método se refiere al método de la clase Trabajador.

**PARÁMETROS HEREDITARIOS**

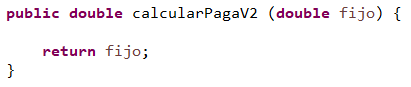
Primero, creamos la clase “Consultor” (otra subclase de la clase Trabajador) y le asignamos sus respectivos atributos y métodos.



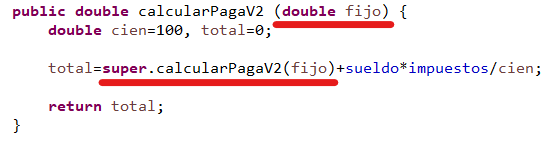


Bien, imaginemos que ahora el valor del sueldo fijo se nos tiene que dar por teclado, entonces tendríamos que crear la variable **fijo** en el “main” y tendríamos que asignarla como parámetro al método “calcularPagaV2” de la clase Trabajador.

Este sería el resultado:

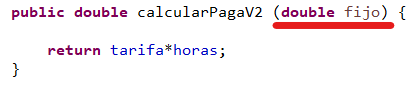


Pero claro, ahora que le hemos asignado un parámetro a una clase madre, la subclase Empleado que llama al método “calcularPagaV2” de su madre para realizar su método “calcularPagaV2” necesita ese parámetro para que funcione correctamente. **En otras palabras, HEREDA ESE PARÁMETRO.**



Pero eso no es todo, **TODAS LAS SUBCLASES DE UNA SUPERCLASE DEBEN HEREDAR SUS PARÁMETROS SI AL MISMO MÉTODO RESPECTA.**

Por ejemplo, el método “calcularPagaV2” de la clase Consultor no necesita de la variable **fijo** para funcionar, **PERO COMO ES SUBCLASE DE LA CLASE TRABAJADOR, TIENE QUE HEREDAR EL PARÁMETRO fijo PARA ESTE MÉTODO, AUNQUE NO SEA NECESARIO.**



**SI EN LA CLASE MADRE HAY UN PARÁMETRO QUE NO HAGA FALTA UTILIZAR EN LA HIJA, SE ESCRIBE DE IGUAL FORMA. ESTOS MÉTODOS TIENEN LA MISMA UTILIDAD EN AMBAS CLASES ¡¡¡MADRE E HIJA DEBEN TENER LA MISMA FIRMA!!!**

**MÉTODOS HEREDITARIOS**

**EL MODIFICADOR “final”**

Este modificador sirve para evitar que un método se pueda redefinir en una subclase:

**public class Consultor extends Trabajador {**

**public final double calcularPaga (){**

**return horas\*tarifa;**

**}**

**En este caso, la clase Consultor tiene el método final “calcularPaga”, pues aunque creemos subclases de Consultor, el dinero que se le pague siempre será en función de las horas que trabaje y de su tarifa horaria (y eso no lo podremos cambiar, aunque queramos).**

Además, el modificador **final** también sirve para evitar que se puedan crear subclases de una clase dada. Por ejemplo:

**public final class Empleado extends Trabajador{**

**Como la clase Empleado tiene el modificador final entre public y class, ya no se podrán crear subclases a partir de ella.**

**CLASES Y MÉTODOS ABSTRACTOS**

Una clase abstracta es una clase genérica creada con el propósito de proporcionar atributos y comportamientos que serán “compartidos” por todas las subclases.

**UNA CLASE SE VUELVE UNA CLASE ABSTRACTA SI CONTIENE MÍNIMO UN MÉTODO ABSTRACTO.**

Un método abstracto es un método calificado con la palabra **abstract** y con la particularidad de que no tiene cuerpo.

Se escribe añadiendo la palabra abstract entre **public** y el **TipoDelMétodo**:

**public abstract TipoDelMétodo NombreMétodo ();**

**public abstract double calcularArea ();**

Estos métodos no tienen cuerpo, porque la idea es proporcionar métodos que deban ser “redefinidos” en las subclases (clases hijas) de la clase abstracta, con la intención de adaptarlos a las necesidades particulares de estas.

**[\*] CONVERSIÓN DE TIPOS (CASTING)**

**Programación Orientada a Objetos II (POLIMORFISMO)**

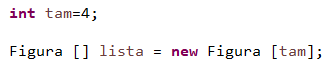
**La palabra “polimorfismo” significa “la facultad de asumir muchas formas”, refiriéndose en Java a la posibilidad de llamar a muchos métodos diferentes utilizando una única sentencia.**

Explicar polimorfismo.

**ARRAYS DE CLASES ABSTRACTAS**

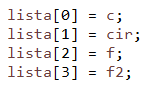
A la hora de declarar objetos, estos no pueden tener el comportamiento de una clase abstracta como se ha explicado anteriormente.

**Pero con los arrays es distinto, un array si puede ser de tipo Figura y además declararse con un “new Figura”.**



Esto ocurre porque un array puede ser de tipo Figura y comportarse como una.

Vamos a probar a rellenarlo con las ”Figuras” que anteriormente hemos creado (no rellenar así el array, ES FEO Y MUY CUTRE):



De esta forma, hemos conseguido diseñar un array de Figuras, que por supuesto, podemos utilizar para crear métodos.

Por ejemplo, vamos a crear un método que sume todas las áreas de las figuras que se encuentren dentro del array.

Como la suma va a realizarse con números con decimales, el método **“sumarAreas”** va a ser de tipo **double**, y obviamente va a necesitar como parámetro un array de objetos:

**public double sumarAreas (Figura [] listado){}**

**Programación Orientada a Objetos II (INTERFACES)**

**class:** No tiene métodos abstractos.

**abstract class:** Al menos 1 método abstracto.

**interface:** Todos sus métodos son abstractos.

**Para declarar una interfaz se utiliza la palabra reservada interface en lugar de class.**

**Todos los miembros de la interfaz (atributos y métodos) son public de manera implícita.**

**Todos los métodos son abstractos también de manera implícita**

**NombreClase Identificador = new NombreConstructor ();**

**Producto p = new Producto ();**

Una clase siempre se puede siempre se puede instanciar (aunque esta esté vacía), PERO NUNCA SE DEBE HACER, SE DEBE INSTANCIAR UNA CLASE LLENA.

Lo que ocurre al instanciar una clase es que se reserva un espacio en la memoria del ordenador al que se le llama como el identificador. Y en ese espacio se guarda la dirección de memoria de esa clase.