



# UT4 DESARROLLO DE CLASES: HERENCIA Y POLIMORFISMO

# HERENCIA

- Es un mecanismo que permite definir unas clases a partir de otras heredando todas sus propiedades.
- El resultado es una nueva clase que extiende la funcionalidad de una clase existente sin tener que reescribir el código asociado a esta última.

# HERENCIA

- La nueva clase, a la que se conoce como **subclase**, puede poseer atributos y métodos que no existan en la clase original
- Los objetos de la nueva clase heredan los atributos y los métodos de la clase original, que se denomina **superclase**

# HERENCIA: LA CLASE OBJECT

- Es la raíz de toda la jerarquía de Java. Todas las clases de Java derivan de Object
- Tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase.

# HERENCIA: CLASE OBJECT

- Los más importantes son los siguientes:
  - `clone()` crea un objeto a partir de otro de la misma clase. No debería llamar al operador `new` ni a los constructores
  - `equals()` Indica si dos objetos son o no iguales. Devuelve `true` si son iguales
  - `toString()` Devuelve un `String` que contiene la representación del objeto como cadena de caracteres, por ejemplo para imprimirlo o exportarlo.
  - `finalize()` se llama automáticamente cuando se vaya a destruir el objeto

# EJEMPLO DE HERENCIA

- Vamos a simular el comportamiento que tendrían los diferentes integrantes de la selección española de futbol; tanto los futbolistas como el cuerpo técnico.
- De cada uno de ellos vamos a necesitar algunos datos que reflejaremos en los **atributos** y una serie de acciones que reflejaremos en sus **métodos**. Estos atributos y métodos los mostramos en el siguiente diagrama de clases:

# EJEMPLO DE HERENCIA

## Futbolista

 id: Integer  
 Nombre: String  
 Apellidos: String  
 Edad: Integer  
 dorsal: Integer  
 demarcacion: String


 Concentrarse(): void  
 Viajar(): void  
 jugarPartido(): void  
 entrenar(): void


## Entrenador

 id: Integer  
 Nombre: String  
 Apellidos: String  
 Edad: Integer  
 idFederacion: String

 Concentrarse(): void  
 Viajar(): void  
 dirigirPartido(): void  
 dirigirEntrenamiento(): void

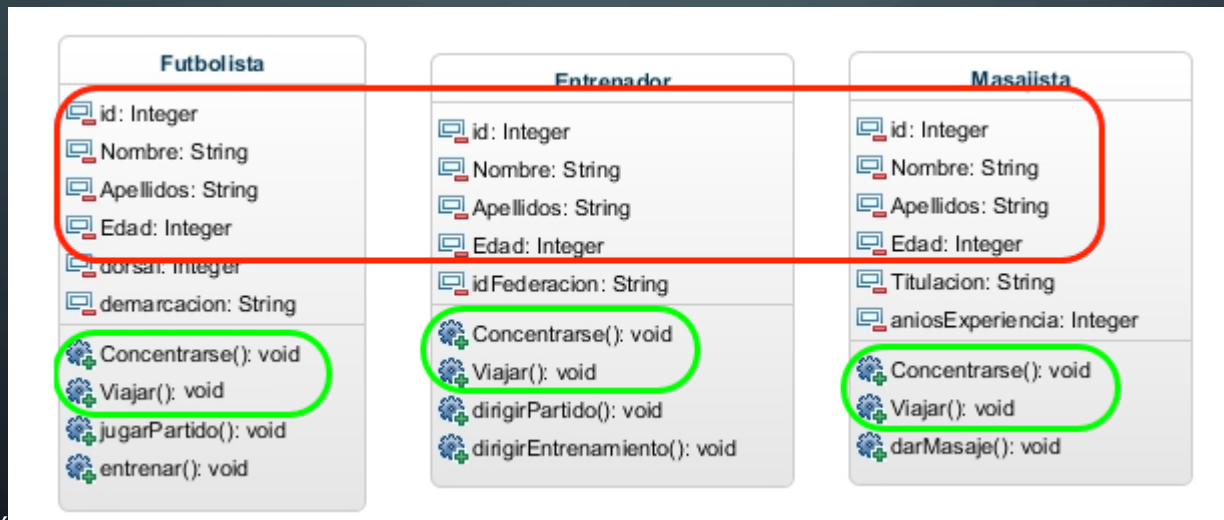
## Masajista

 id: Integer  
 Nombre: String  
 Apellidos: String  
 Edad: Integer  
 Titulacion: String  
 aniosExperiencia: Integer

 Concentrarse(): void  
 Viajar(): void  
 darMasaje(): void

# EJEMPLO DE HERENCIA

- Como se puede observar, vemos que en las tres clases tenemos atributos y métodos que son iguales ya que los tres tienen los atributos *id*, *Nombre*, *Apellidos* y *Edad*; y los tres tienen los métodos de *Viajar* y *Concentrarse*:





# EJEMPLO HERENCIA

```
public class Futbolista
{

    private int id;
    private String Nombre;
    private String Apellidos;
    private int Edad;
    private int dorsal;
    private String demarcacion;

    // constructor, getter y setter

    public void Concentrarse() {
        ...
    }

    public void Viajar() {
        ...
    }

    public void jugarPartido() {
        ...
    }

    public void entrenar() {
        ...
    }

}
```

```
public class Entrenador
{

    private int id;
    private String Nombre;
    private String Apellidos;
    private int Edad;
    private String idFederacion;

    // constructor, getter y setter

    public void Concentrarse() {
        ...
    }

    public void Viajar() {
        ...
    }

    public void dirigirPartido() {
        ...
    }

    public void dirigirEntreno() {
        ...
    }

}
```

```
public class Masajista
{

    private int id;
    private String Nombre;
    private String Apellidos;
    private int Edad;
    private String Titulacion;
    private int aniosExperiencia;

    // constructor, getter y setter

    public void Concentrarse() {
        ...
    }

    public void Viajar() {
        ...
    }

    public void darMasaje() {
        ...
    }

}
```

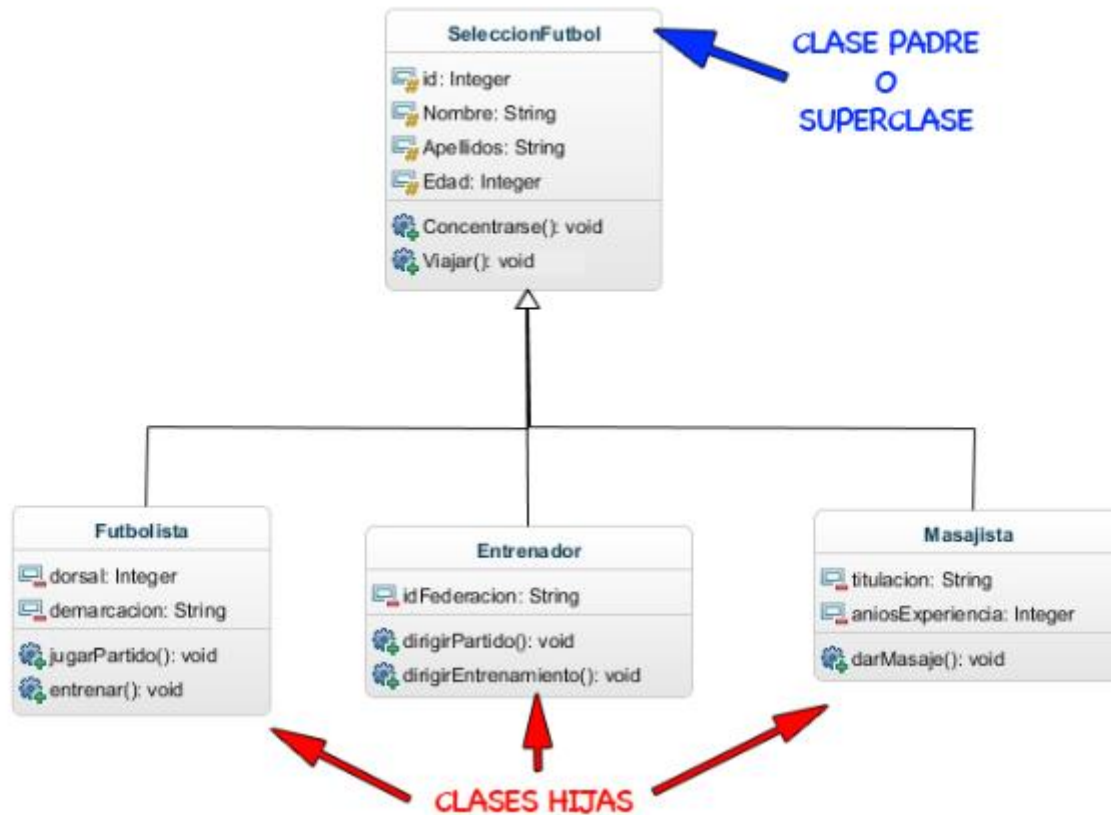
# HERENCIA

- *El objetivo será:*
- *Crear una clase con el código que es común a las tres clases que será la Clase Padre o SuperClase*
- *Con el código que es específico de cada clase, lo dejaremos en ella, siendo denominadas estas clases como Clases Hijas o subclases*

# HERENCIA

- Las clases hijas o subclases *heredan de la clase padre todos los atributos y métodos públicos o protegidos.*
- Es muy importante decir que las clases hijas *no van a heredar nunca los atributos y métodos privados de la clase padre*

# EJEMPLO DE HERENCIA



```

public class SeleccionFutbol
{

    protected int id;
    protected String Nombre;
    protected String Apellidos;
    protected int Edad;

    // constructor, getter y setter

    public void Concentrarse() {
        ...
    }

    public void Viajar() {
        ...
    }

}

```

```

public class Futbolista extends SeleccionFutbol
{
    private int dorsal;
    private String demarcacion;

    public Futbolista() {
        super();
    }

    // getter y setter

    public void jugarPartido() {
        ...
    }

    public void entrenar() {
        ...
    }

}

```

```

public class Entrenador extends SeleccionFutbol
{
    private String idFederacion;

    public Entrenador() {
        super();
    }

    // getter y setter

    public void dirigirPartido() {
        ...
    }

    public void dirigirEntreno() {
        ...
    }

}

```

```

public class Masajista extends SeleccionFutbol
{
    private String Titulacion;
    private int aniosExperiencia;

    public Masajista() {
        super();
    }

    // getter y setter

    public void darMasaje() {
        ...
    }

}

```

# OPERADOR INSTANCEOF

- El operador **instanceof** nos permite comprobar si un objeto es de una **clase concreta**.

# OPERADOR INTANCEOF

Por ejemplo, tenemos 3 clases:

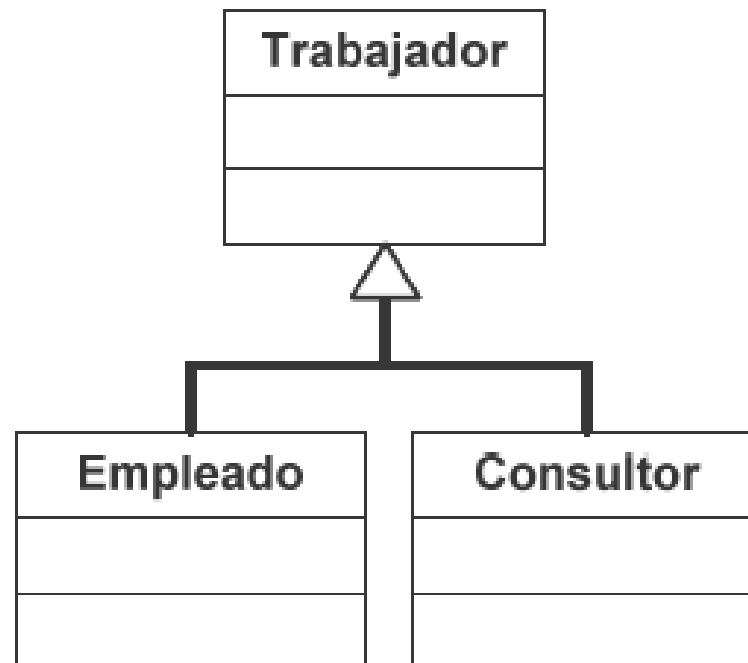
- Empleado (clase padre),
- Comercial (clase hija de Empleado)
- Repartidor (clase hija Empleado).

# OPERADOR INSTANCEOF

```
public class EmpleadoApp {  
  
    public static void main(String[] args) {  
  
        Empleado empleados[]=new Empleado[3];  
        empleados[0]=new Empleado();  
        empleados[1]=new Comercial();  
        empleados[2]=new Repartidor();  
  
        for(int i=0;i<empleados.length;i++){  
            if(empleados[i] instanceof Empleado){  
                System.out.println("El objeto en el indice "+i+" es de la clase Empleado");  
            }  
            if(empleados[i] instanceof Comercial){  
                System.out.println("El objeto en el indice "+i+" es de la clase Comercial");  
            }  
            if(empleados[i] instanceof Repartidor){  
                System.out.println("El objeto en el indice "+i+" es de la clase Repartidor");  
            }  
        }  
    }  
}
```



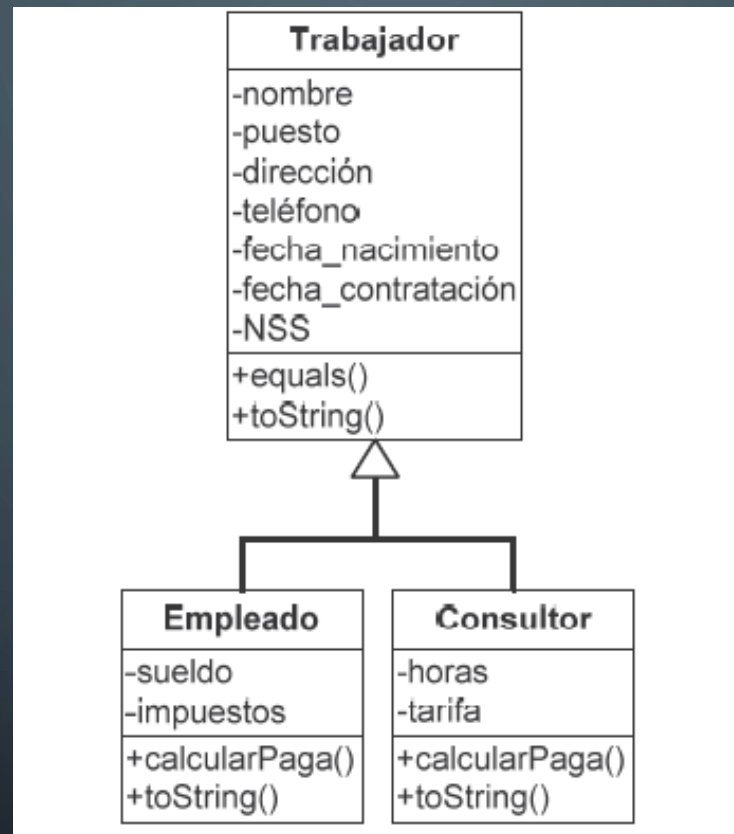
# HERENCIA



# HERENCIA

- § Trabajador es una clase genérica que sirve para almacenar datos como el nombre, la dirección, el número de teléfono o el número de la seguridad social de un trabajador.
- § Empleado es una clase especializada para representar los empleados que tienen una nómina mensual (encapsula datos como su salario anual o las retenciones del IRPF).
- § Consultor es una clase especializada para representar a aquellos trabajadores que cobran por horas (por ejemplo, registra el número de horas que ha trabajado un consultor y su tarifa horaria).

# HERENCIA



# HERENCIA

Las clases Empleado y Consultor además de los atributos y operaciones que definen, heredan de Trabajador todos sus atributos y operaciones.

| <u>unEmpleadoConcreto</u>  |
|--|
| -nombre<br>-puesto<br>-dirección<br>-teléfono<br>-fecha_nacimiento<br>-fecha_contratación<br>-NSS<br>-sueldo<br>-impuestos |
| +equals()<br>+toString()<br>+calcularPaga()  |

# EJERCICIO

- Crear la clase Trabajador con todos sus atributos, los métodos de acceso y modificación y definiendo los métodos toString() y equals() (siempre deberemos definirlos).
- El constructor tiene sólo 2 parámetros que son el nombre y el NSS.
- El método toString() debe imprimir el nombre del Trabajador seguido de su NSS entre paréntesis. P.e. *Marta (NSS 123)*

# EJERCICIO

- Crear la clase Empleado que hereda de trabajador.

```
public class Empleado extends Trabajador
```

- Con la palabra reservada **super** accedemos a los miembros de la superclase desde la subclase
  - En el constructor lo primero que encontramos es una llamada al constructor de la clase padre con `super(...)`. Si no ponemos nada, se llama al constructor por defecto de la superclase antes de ejecutar el constructor de la subclase. Debe tener los siguientes parámetros.

```
public Empleado(String nombre, String NSS, double sueldo)
```

```
La variable impuestos=0.3*suelo;
```

- El método `calcularPaga()` realiza la siguiente operación:

**(sueldo-impuestos)/PAGAS**

donde PAGAS es una variable privada de la clase Empleado que tiene un valor constante de 14.

- Redefinir el método `toString` para que imprima lo siguiente:

***Empleado Marta (NSS 123)***

# EJERCICIO

- Crea la clase Consultor que hereda de Trabajador.

*public class Consultor extends Trabajador*

- Crea un constructor con los siguientes parámetros:

*public Consultor(String nombre, String NSS, int horas, double tarifa)*

- El método calcularPaga() realiza la siguiente operación:

*horas\*tarifa*

- Redefinir el método toString para que imprima lo siguiente:

***Consultor Marta (NSS 123)***

# EJERCICIO

```
System.out.println(t);  
System.out.println(e);  
System.out.println(c);
```

- Crea en la clase principal los siguientes objetos:

*Trabajador t=new Trabajador ("Marta","1 2 3");*

*Empleado e=new Empleado("Juan", "456",24000.0);*

*Consultor c= new Consultor("Marta", "1 2 3", 10, 50.0);*

- Imprímelos por pantalla
- Comprueba si el trabajador es un empleado o un consultor utilizando el método equals()

```
if (t.equals(e))  
    System.out.println("El trabajador "+t.GetNombre() +" es un empleado");
```



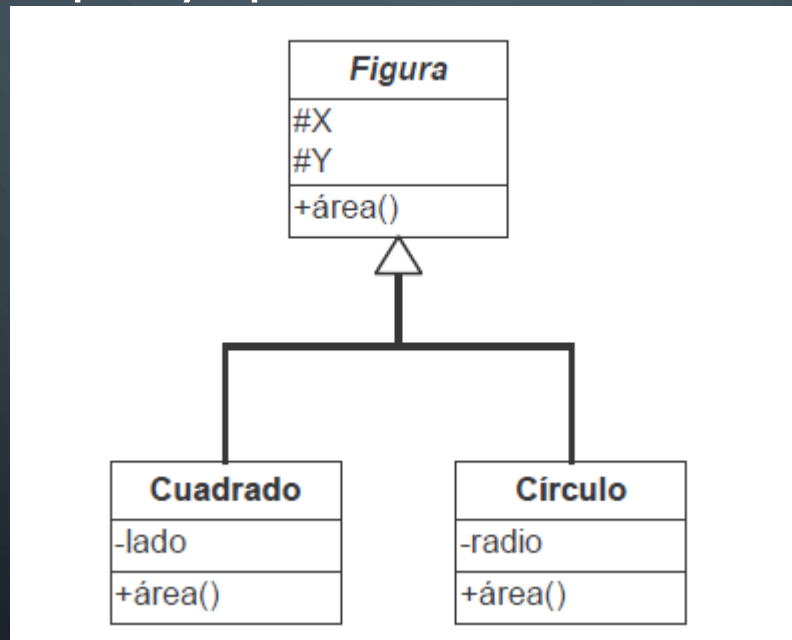
# CLASES ABSTRACTAS

- Es una clase de la que no se pueden crear objetos.
- Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general

```
public abstract class Miclase {}
```

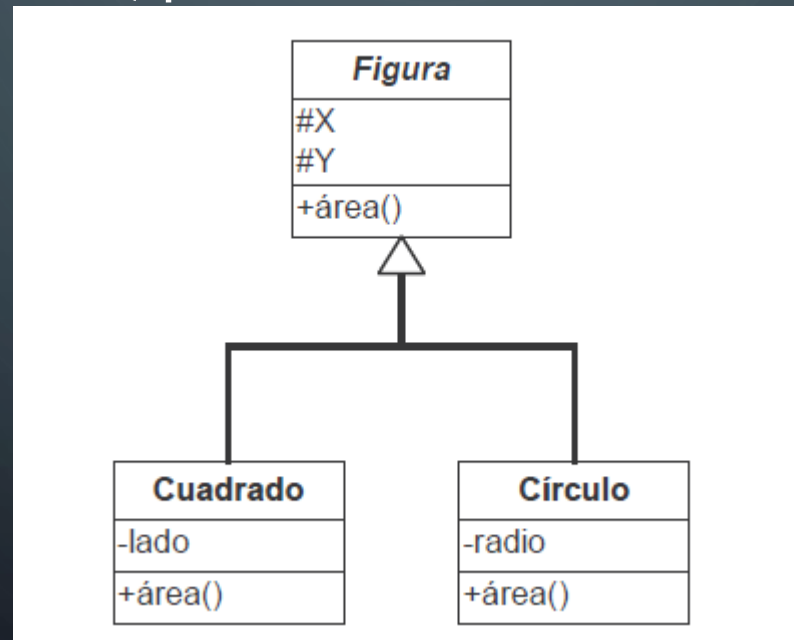
# ¿CUÁNDO SE UTILIZAN LAS CLASES ABSTRACTAS?

Cuando deseamos definir una clase que englobe objetos de distintos tipos y queremos hacer uso del polimorfismo



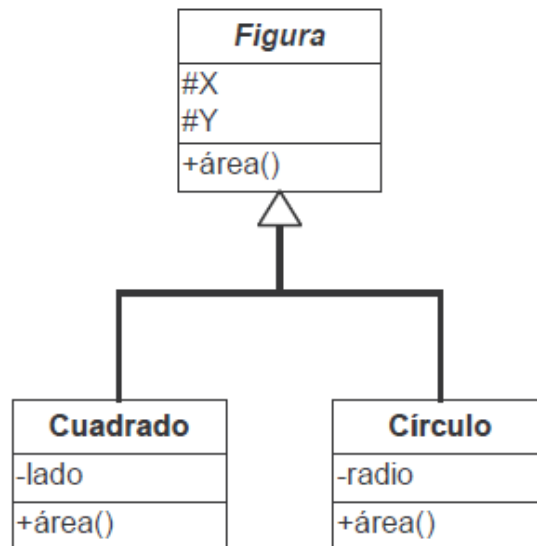
# CLASE ABSTRACTA

Figura es una clase abstracta porque no tiene sentido calcular su área, pero sí la de un cuadrado o un círculo.



# EJERCICIO

- Implementa en Java estas 3 clases, sabiendo que el área del cuadrado es  $\text{lado} * \text{lado}$  y la del círculo es  $\text{PI} * \text{radio} * \text{radio}$
- Las variables X e Y indican la posición y sólo pueden acceder a ellas las clases hijas de Figura.



# POLIMORFISMO

Al redefinir métodos, objetos de diferentes tipos pueden responder de forma distinta a la misma llamada (y podemos escribir código de forma general sin preocuparnos del método concreto que se ejecutará en cada momento).

# POLIMORFISMO

- Ejemplo:

Podemos añadirle a la clase Trabajador un método calcularPaga genérico (que no haga nada):

```
public class Trabajador...  
    public double calcularPaga ()  
    {  
        return 0.0;                // Nada por defecto  
    }
```

# POLIMORFISMO

En las subclases de Trabajador, no obstante, sí que definimos el método calcularPaga() para que calcule el importe del pago que hay que efectuarle al trabajador (en función de su tipo).

```
public class Empleado extends Trabajador...  
    public double calcularPaga ()          // Nómina  
    {  
        return (sueldo-impuestos)/PAGAS;  
    }
```

```
class Consultor extends Trabajador...  
    public double calcularPaga ()          // Por horas  
    {  
        return horas*tarifa;  
    }
```

# POLIMORFISMO

Como los consultores y los empleados son trabajadores, podemos crear un array de trabajadores con consultores y empleados:

```
Trabajador trabajadores[]=new Trabajador[2];  
trabajadores[0]=new Empleado("Jose","135", 24000.0);  
trabajadores[1]=new Consultor("Juan","246",10,50.0);
```



# POLIMORFISMO

- Una vez que tengamos un vector con todos los trabajadores de una empresa, podríamos crear un programa que realizase los pagos correspondientes a cada trabajador.

```
Trabajador trabajadores[]=new Trabajador[2];  
trabajadores[0]=new Empleado("Jose","135", 24000.0);  
trabajadores[1]=new Consultor("Juan","246",10,50.0);
```

# POLIMORFISMO

La salida por pantalla sería la siguiente:

```
El trabajador Jose tiene una paga de 1200.0  
El trabajador Juan tiene una paga de 500.0  
BUILD SUCCESSFUL (total time: 1 second)
```

# POLIMORFISMO

- Cada vez que se invoca el método `calcularPaga()`, se busca automáticamente el código que en cada momento se ha de ejecutar en función del tipo de trabajador (enlace dinámico)
- La búsqueda del método que se ha de invocar como respuesta a un mensaje dado se inicia con la clase receptor. Si no se encuentra un método apropiado en esta clase, se busca en su clase padre y así sucesivamente hasta encontrar la implementación adecuada del método que se ha de ejecutar como respuesta a la invocación original

# POLIMORFISMO

No hay que confundir el polimorfismo con la sobrecarga de métodos (distintos métodos con el mismo nombre pero con diferentes parámetros) p.e. la sobrecarga de constructores dentro de una misma clase

```
public Punto (){  
    this.x = 0.0;  
    this.y = 0.0;  
}
```

```
public Punto ( double coordenadaX, double coordenadaY){  
    x = coordenadaX;  
    y = coordenadaY;  
}
```

# LA PALABRA RESERVADA FINAL

En Java, usando la palabra reservada final, podemos:

- Evitar que un método se pueda redefinir en una subclase

```
class Consultor extends Trabajador
{
    ...
    public final double calcularPaga ()
    {
        return horas*tarifa;
    }
    ...
}
```

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 podremos cambiarlo aunque queramos)

# LA PALABRA RESERVADA FINAL

- Evitar que se puedan crear subclases de una clase dada:

```
public final class Circulo extends Figura
...

public final class Cuadrado extends Figura
...
```

Al usar “final”, tanto Circulo como Cuadrado son ahora clases de las que no se puede crear subclases.

En ocasiones será “final” porque no tenga sentido crear subclases o, simplemente porque no deseemos que la clase se pueda extender.

RECORDATORIO: final también sirve para definir constantes