

# WUOLAH



alvaromontorio

[www.wuolah.com/student/alvaromontorio](http://www.wuolah.com/student/alvaromontorio)



## HerenciaPolimorfismo.pdf

*Apuntes Tema 5: Herencia y Polimorfismo*



1º Programación II



Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenieros Informáticos  
Universidad Politécnica de Madrid



## ¿Harto de chapar algo que **no te renta?**

¿Cuál es tu trabajo ideal?

Haz el test aquí

<http://bit.ly/necesitouncambio>





Grado en Ingeniería Informática  
Grado en Matemáticas e Informática



Asignatura: PROGRAMACIÓN II

# Herencia y Polimorfismo

Profesores de Programación II

DLSIIS - E.T.S. de Ingenieros Informáticos  
Universidad Politécnica de Madrid

Diciembre 2016

A person is seen from behind, standing at a concert or festival. Their arms are raised in the air, and they are wearing a wristband. The background is filled with bright, warm yellow and orange light, suggesting stage lighting or a sunset. Other people's hands are visible in the background, also raised.

# ENCENDER TU LLAMA CUESTA MUY POCO



BURN.COM

**BURN**  
ENERGY DRINK

#StudyOnFire

# Herencia simple


# Contenido

- Motivación: similitud entre clases
- Conceptos de herencia
  - ➔ Especialización
  - ➔ Principio de sustitución
- Herencia en Java
  - ➔ Constructores, atributos, métodos
- Referencias a objetos especializados
  - ➔ *Upcasting, downcasting*
- Nivel de acceso *protected*





¿Cuál es tu trabajo ideal?

Haz el test aquí 

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Ejemplo animales sin herencia

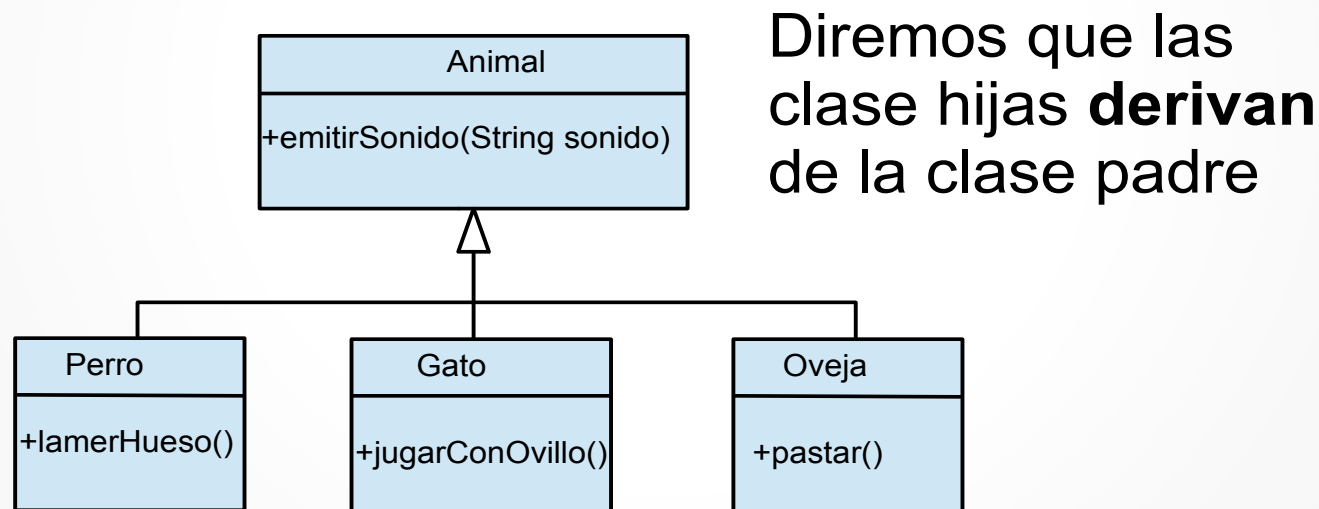
- Crear un proyecto java con el nombre Animales
- Crear una clase Perro con un método emitirSonido() que imprime por consola “guau” y un método lamerHueso() que imprime por consola “lamiendo hueso”
- Crear una clase Gato con un método emitirSonido() que imprime por consola “miau” y un método jugarConOvillo() 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()

# Introducción a la herencia

- El método emitirSonido() es común a las clases Perro, Gato y Oveja
- Cada clase tiene métodos que son específicos de esa clase, por ejemplo, el método lamerHueso() es específico de la clase Perro
- El mecanismo de **herencia** permite implementar una clase (la **clase hija**) a partir de otra clase (la **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

# Conceptos de la herencia

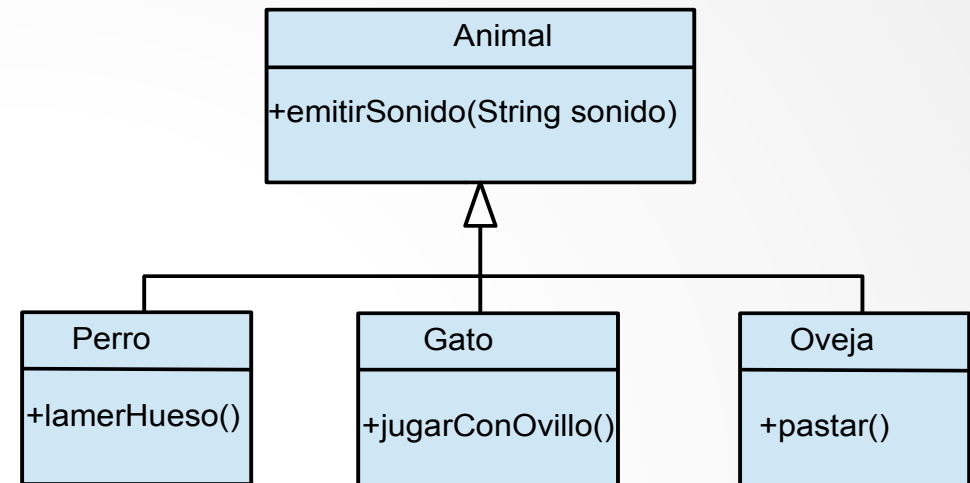
- Con el mecanismo de **herencia** diremos que:
  - Animal es la **clase padre** (o **superclase** o **clase base**) de las clases Perro, Gato y Oveja
  - La clase Perro es una **clase hija** (o **subclase** o **clase derivada**) de la clase Animal





# Conceptos de la herencia

- Una clase hija **es una correcta subclase** de una clase padre si cumple las siguientes reglas:
  - ➡ **Regla Es-un:** un objeto de la clase hija es también un objeto de la clase *padre*.
  - ➡ **Principio de sustitución:** los objetos de la clase hija pueden sustituir a los de la clase padre cuando se les pida realizar una operación.





¿Cuál es tu trabajo ideal?

Haz el test aquí

<http://bit.ly/necesitouncambio>

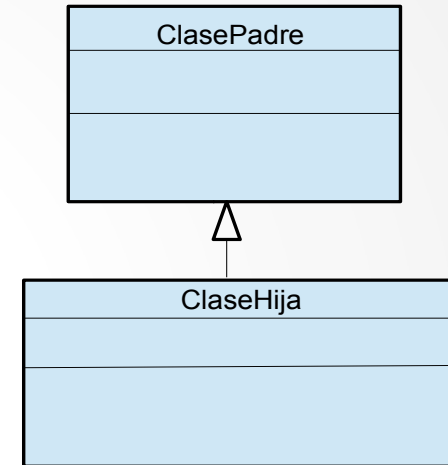
# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Herencia de clases en Java

- Java permite definir una clase como clase derivada o subclase de una clase padre.

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



- No permite la herencia múltiple (**sólo un padre**)

# Ventajas de usar herencia

- Reutilizar código
  - ➔ No hay que escribir el método emitirSonido() en cada clase. Escribimos menos código.
  - ➔ Al tener el método emitirSonido(String sonido) sólo en la clase Animal si hacemos algún cambio en ese método sólo hay que hacerlo en la clase Animal en vez de en todas las clases hijas Perro, Gato, Vaca...
  - ➔ A menos líneas de código, menos errores
- La herencia tiene otras ventajas que iremos viendo

# Ejemplo de 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 de la T7.

# Herencia de atributos

- Ahora queremos que los objetos de tipo Perro, Gato y Oveja tengan como atributos nombre y edad.
- Sin el mecanismo de herencia tendríamos que definir esos dos atributos en cada una de las clases Perro, Gato y Oveja
- Sin embargo, gracias a la herencia, sólo necesitamos definirlos en la clase Animal.





¿Cuál es tu trabajo ideal?  
Haz el test aquí

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Constructores y herencia

- En el ejemplo, creamos constructores en las clases Perro, Gato y Oveja que inicialicen los atributos nombre y edad, ¿Qué observamos?
  - ➔ Solución: creamos un constructor en la clase Animal que inicialice los atributos nombre y edad
- Necesitamos pasar los argumentos desde los constructores de las clases hijas al constructor de la clase Animal.
  - ➔ Solución: **super**

# Constructores y herencia

- Recordatorio: en una clase si no hay declarado ningún constructor siempre está el constructor por defecto pero si hay declarado un constructor, ya no se puede llamar al constructor por defecto a menos que lo creamos explícitamente
- 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**

# Constructores y herencia


- En la clase Animal modificamos el método emitirSonido(String sonido) para que además del sonido imprima también el nombre del animal y probamos este código en la clase de prueba
- Se añaden los siguientes atributos:
  - A la clase Oveja le vamos a añadir un atributo pastor de tipo String.
  - A la clase Perro le vamos a añadir un atributo amo de tipo String.

# Constructores y Herencia

```
public class Animal {  
    private String nombre; // En este ejemplo no hace falta declararlos  
    private int edad;  
    public Animal (String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}  
  
public class Perro extends Animal {  
    private String amo;  
    public Perro (String nombre, int edad, String amo) {  
        super(nombre, edad);  
        this.amo = amo;  
    }  
}  
  
public class Oveja extends Animal {  
    private String pastor;  
    public Oveja (String nombre, int edad, String pastor) {  
        super(nombre, edad);  
        this.pastor = pastor;  
    }  
}
```



¿Cuál es tu trabajo ideal?

Haz el test aquí 

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Herencia de atributos

- 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.



# Referencias a objetos: *upcasting*

```
public class PruebaHerencia {  
    public static void main(String[] args) {  
        Animal flora = new Animal("Flora",3,"beeee");  
        flora.emitirSonido();  
        // En una variable de tipo Animal se puede guardar la referencia  
        // a un objeto de tipo Oveja. Esto se llama UPCASTING  
        Animal perlita = new Oveja("perlita",2, "Juan");  
        perlita.emitirSonido();  
        Perro toby = new Perro("Toby",5, "Pepe");  
        toby.emitirSonido();  
  
        // En una variable de tipo Animal se puede guardar la referencia  
        // a un objeto de tipo Perro. Esto se llama UPCASTING  
        Animal tobyDuplicado = toby;  
        tobyDuplicado.emitirSonido();  
  
        // continua en la siguiente transparencia
```

# Referencias a objetos: *downcasting*

```
// la siguiente línea da error porque un objeto de tipo Oveja no  
// puede contener una referencia de tipo Animal  
// Oveja perlitaDuplicada = perlita;  
// pero se puede hacer DOWNCASTING  
Oveja perlitaDuplicada = (Oveja) perlita;
```

```
// Otro ejemplo de DOWNCASTING  
((Oveja) perlita).pastar();
```

```
// Este código da error y no se puede hacer DOWNCASTING  
// Oveja floraDuplicada = (Oveja) flora;
```

```
// continua en la siguiente transparencia
```

# Referencias a objetos: *downcasting*

// En este caso también se puede hacer **DOWNCASTING**

Perro tobyTriplicado = (Perro) tobyDuplicado


// Otro ejemplo de **DOWNCASTING**

((Perro) tobyDuplicado).lamerHueso();

// continua en la siguiente transparencia



¿Cuál es tu trabajo ideal?

Haz el test aquí 

<http://bit.ly/necesitouncambio>

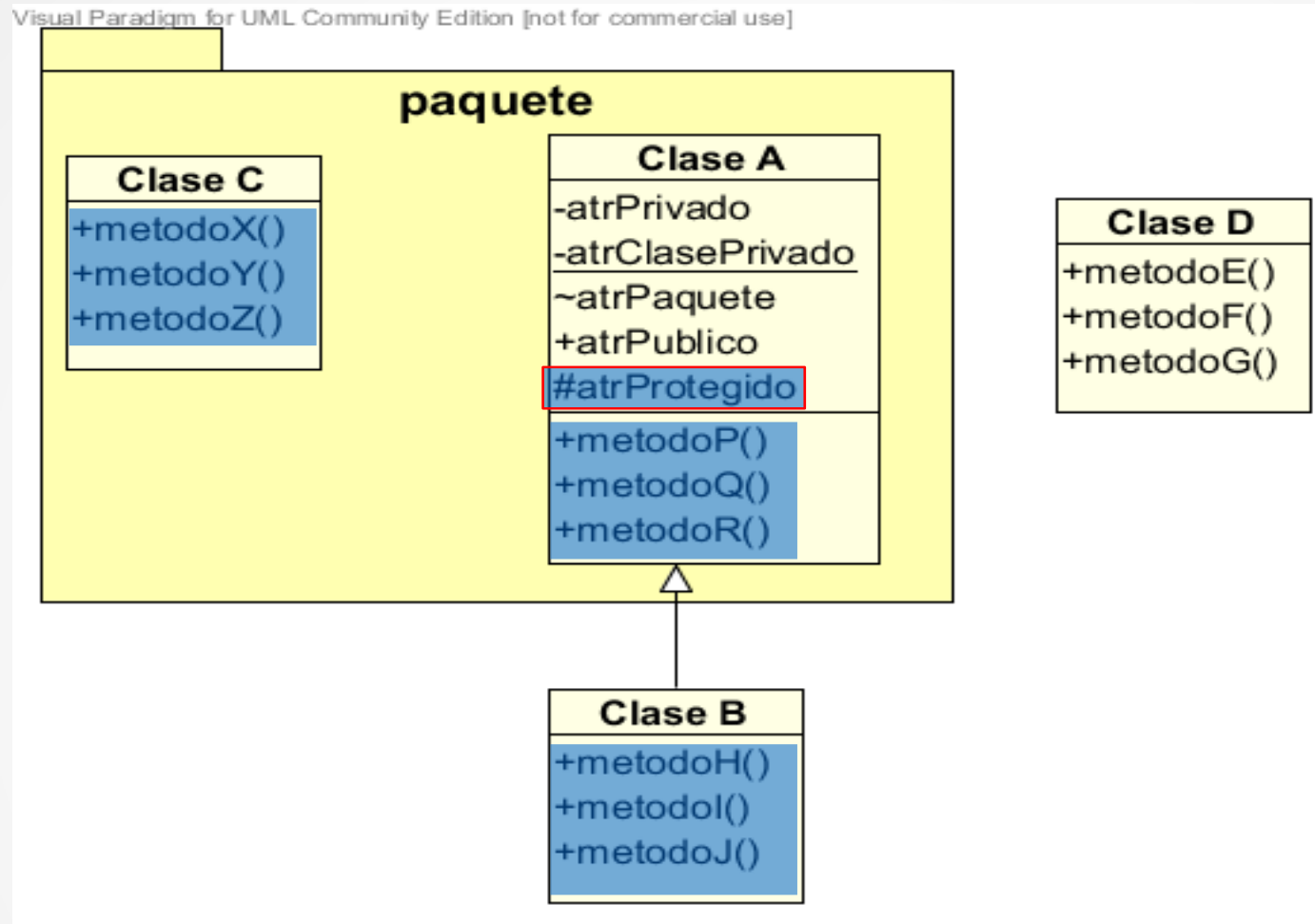
# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Referencias a objetos: *downcasting* controlado

```
// DOWNCASTING controlado
if (perlita instanceof Oveja){
    Oveja perlitaTriplicada = (Oveja) perlita;
} else {
    System.out.println("Este objeto no es de tipo Oveja");
}
if (flora instanceof Oveja){
    Oveja floraDuplicada = (Oveja) flora;
} else {
    System.out.println("Este objeto no es de tipo Oveja");
}
}
```

# Nivel de acceso *protected*

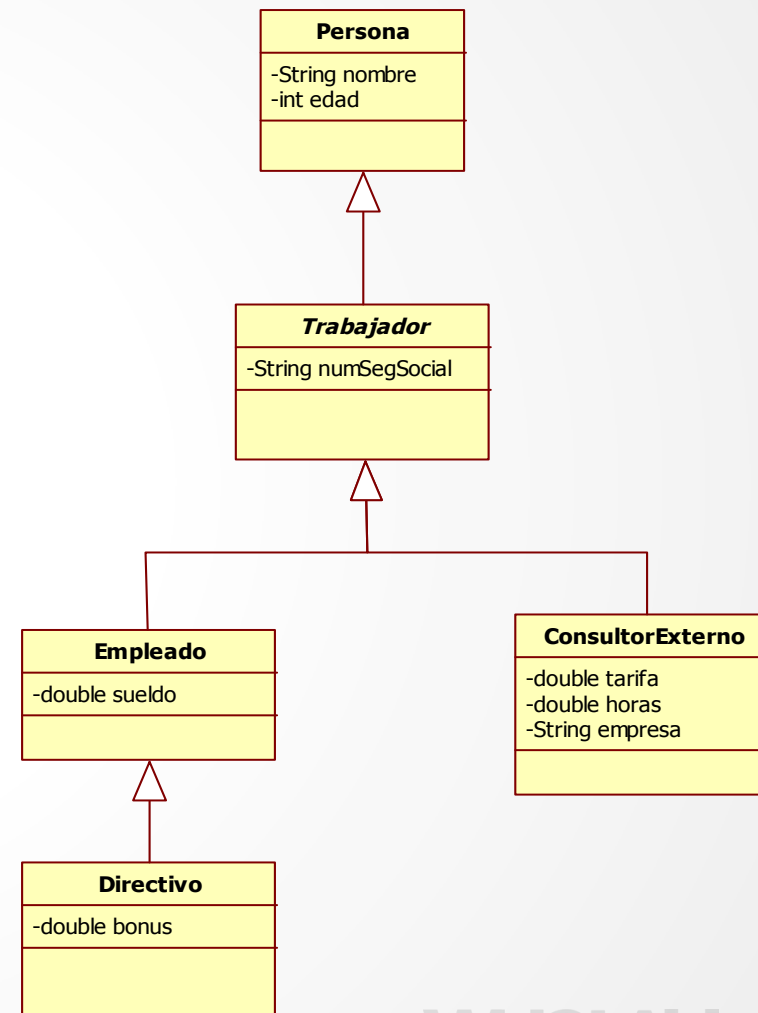




# Ejercicio

- 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

No ponemos los constructores



# Herencia con sobrescritura

**Herencia simple**


WUOLAH

23

¿Harto de chapar algo que no te renta? Descubre tu trabajo ideal. ¡Haz el test!



¿Cuál es tu trabajo ideal?

Haz el test aquí 

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

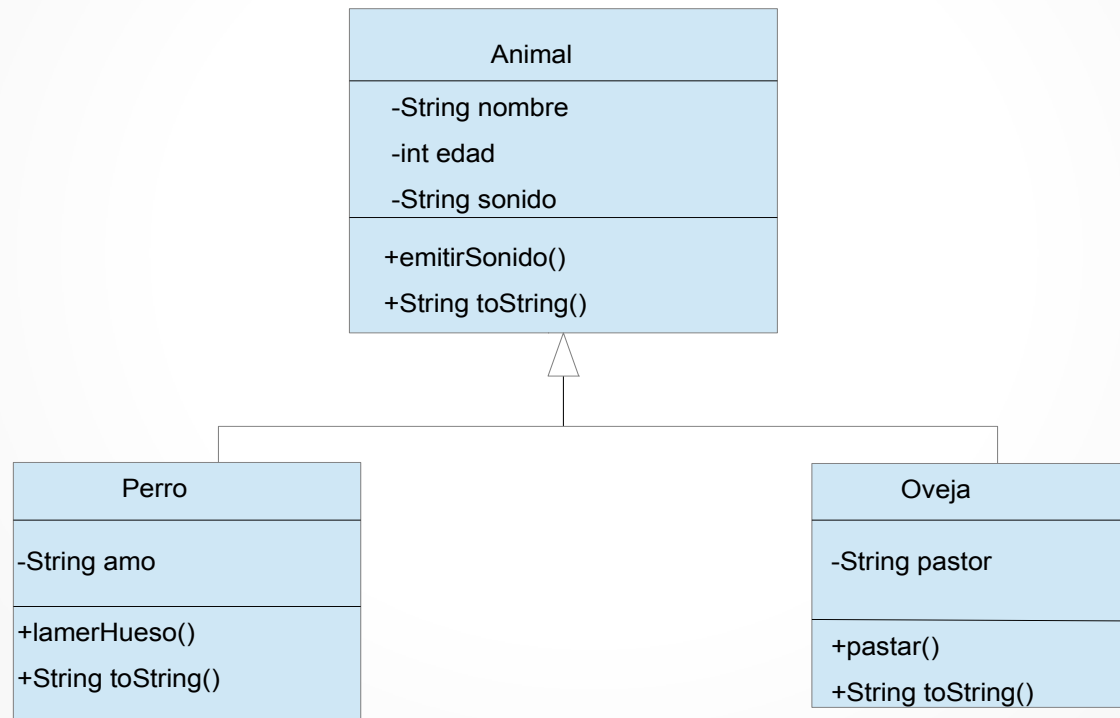
IRON  
HACK

## Contenido

- Concepto de sobrescritura (*override*)
- Polimorfismo
- Restringir la sobrescritura
  - ➔ Modificador **final**
- Algunos métodos universales
  - ➔ Heredados de **Object**
  - ➔ **equals**, **toString**, ...

# Ejemplo animales

- Queremos implementar un método toString() que devuelva un String con los valores de todos los atributos de los objetos de cada clase.



# Conceptos sobrescritura

- A veces se quiere que las clases hijas completen o cambien el comportamiento de un método de la clase padre. Esto se llama **sobrescribir** (o **redefinir**) un método
- El método redefinido (de la clase hija) puede usar la implementación dada por el padre (para ello en Java se usa **super**)
- O usar cualquier otro método accesible, tanto del padre como de sí mismo



# Redefinir métodos de la clase padre

```
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() {  
        return super.toString() + " Amo: " + this.amo;  
    }  
}
```

Se usa la implementación del padre



¿Cuál es tu trabajo ideal?

Haz el test aquí

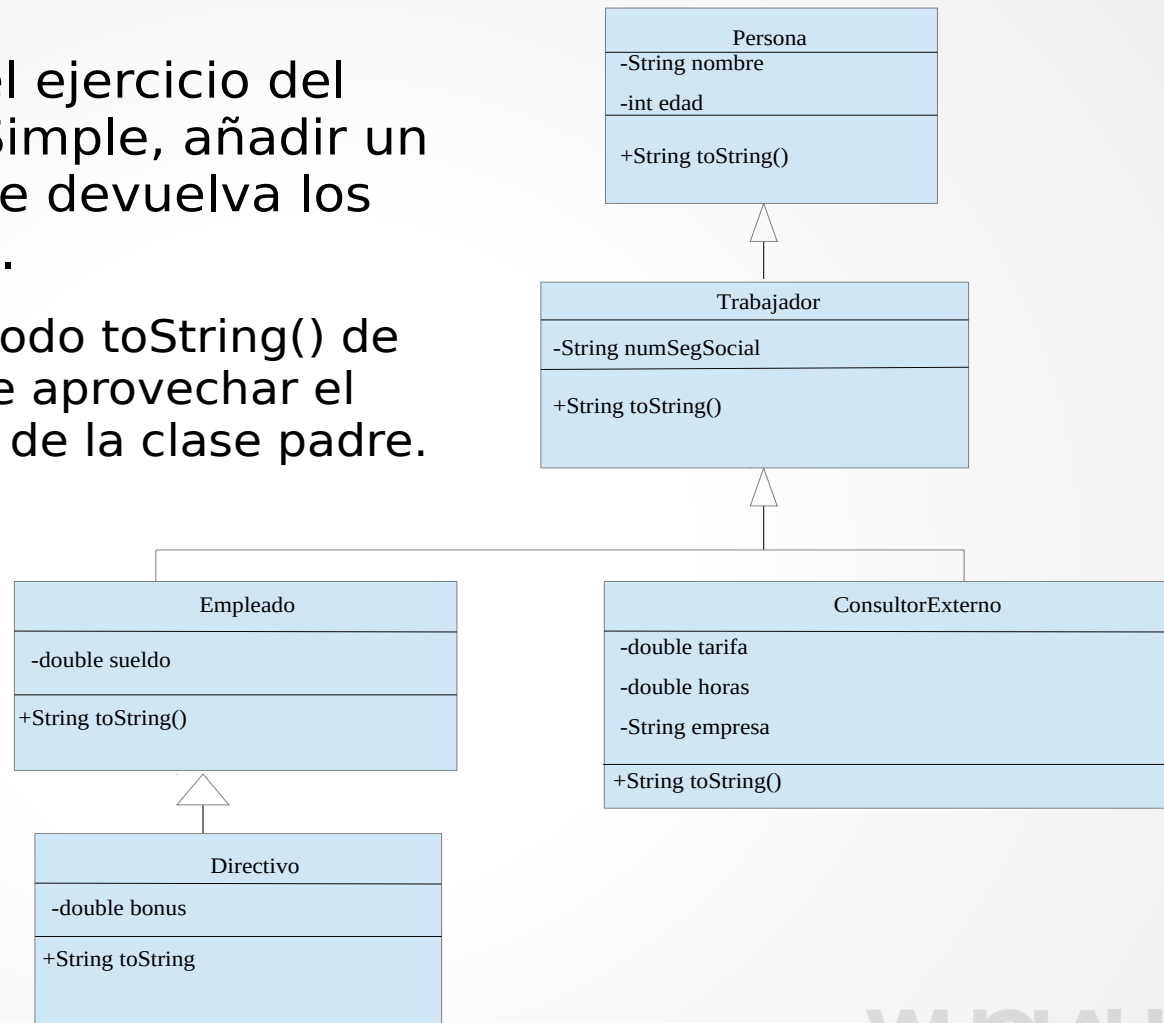
<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que no te renta?

IRON  
HACK

## Ejercicio

- Dadas las clases del ejercicio del tema de Herencia Simple, añadir un método toString que devuelva los atributos del objeto.
  - ➡ Al redefinir el método toString() de una clase, se debe aprovechar el método toString() de la clase padre.



# Polimorfismo en el ejemplo de animales

- 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.
- Creamos un vector que contenga todos 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.

# Polimorfismo en el ejemplo de los animales

- En el ejemplo anterior tenemos un vector que declaramos que contiene objetos de tipo Animal pero realmente puede contener también objetos de tipo Perro y de tipo Oveja.
  - ➔ Esto se llama **POLIMORFISMO** porque la misma variable, por ejemplo, animales[0], puede contener una referencia a un objeto de tipo Animal o de tipo Perro o de tipo Oveja.
- Durante la ejecución del programa, al llegar a la llamada al método toString() el sistema de ejecución decide si se ejecuta el código del método toString() de la clase Animal o el de la clase Perro o el de la clase Oveja.
  - Esto se llama **DYNAMIC BINDING** o enlazado dinámico.

# Polimorfismo

- El polimorfismo en POO se da por el uso de la herencia
- Se produce por distintas implementaciones de los métodos definidos en la clase padre (**sobrescribir**):
  - ➔ Distinta implementación entre clase hija y padre
  - ➔ Distinta implementación 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**



# ¿Harto de chapar algo que no te renta?

IRON  
HACK



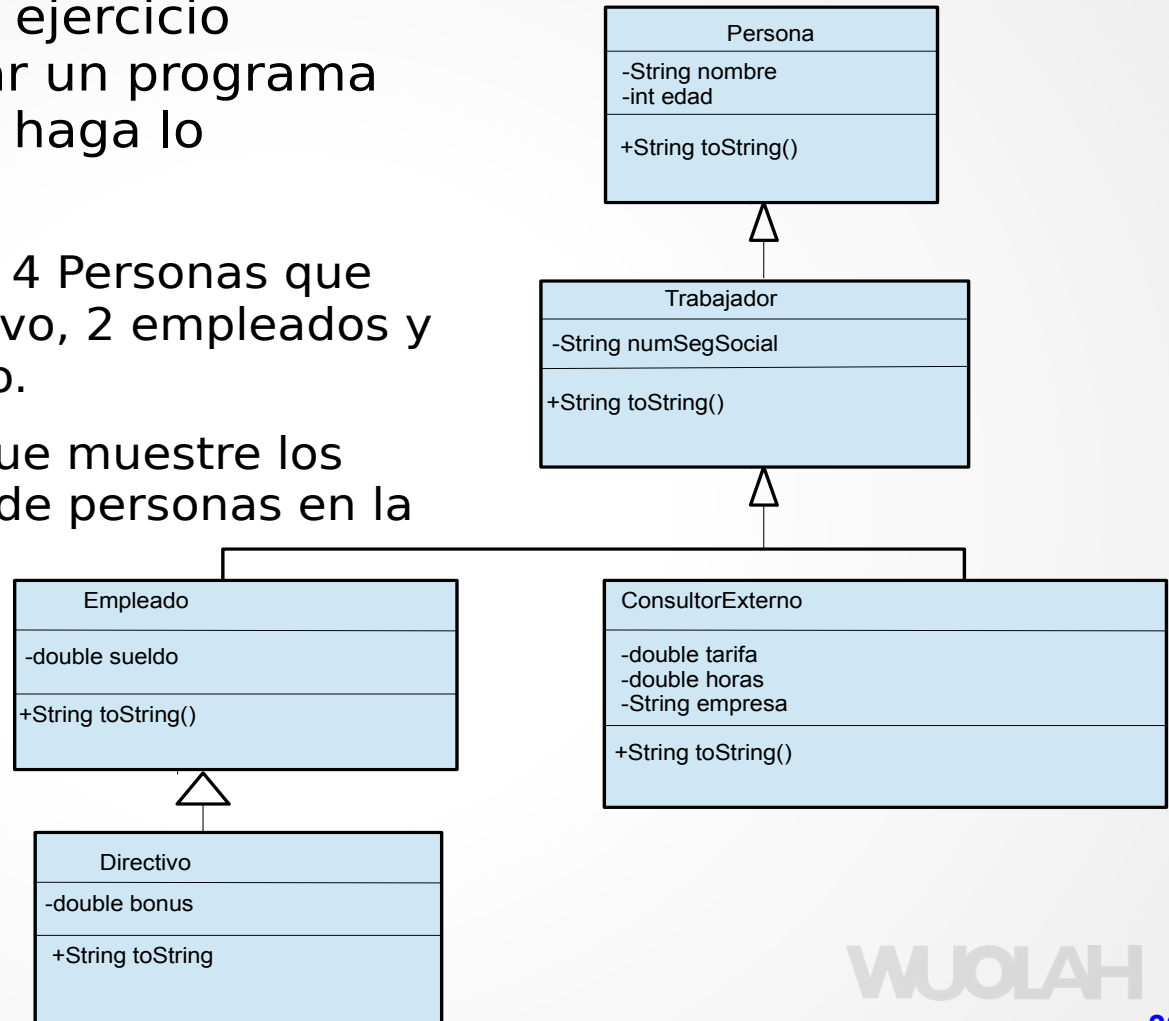
¿Cuál es tu trabajo ideal?

Haz el test aquí

<http://bit.ly/necesitouncambio>

## Ejercicio

- Dadas las clases del ejercicio anterior implementar un programa principal (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.



# Modificador final

- En Java

- ➔ Si se pone **final** delante de un atributo, ese atributo no se puede modificar después de ser inicializado.
- ➔ Si se pone **final** delante de un método significa que ese método no se puede sobrescribir.
- ➔ Si se pone **final** delante de una clase, esa clase no puede tener clases hijas.



# La clase Object

- En Java todas las clases heredan de la clase **Object** de manera implícita (no hace falta poner “extends Object”).
- Métodos como equals() y toString() están en la clase Object y se pueden sobrescribir en cualquier clase.

# Redefinición equals de la clase Object

```
// Método implementado dentro de la clase Fecha
// el parámetro no puede ser de tipo Fecha!!
public boolean equals (Object o) {
    Fecha fecha = (Fecha) o; // downcasting: Object → Fecha
    return (day == fecha.day) &&
        (month == fecha.month) &&
        (year == fecha.year);
}

//Programa principal en un fichero .java distinto de
//Fecha.java
public static void main(String[] args) {
    Fecha ob1, ob2;
    ob1 = new Fecha(12, 4, 96);
    ob2 = new Fecha("12/4/96");
    System.out.println("La primera fecha es " + ob1);
    System.out.println("La segunda fecha es " + ob2);
    System.out.println(ob1 == ob2); System.out.println(ob1.equals(ob2));
}
```



¿Cuál es tu trabajo ideal?

Haz el test aquí 👉

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Herencia abstracta

Herencia simple

WUOLAH

# Contenido

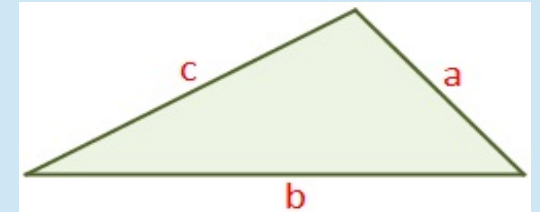
- Concepto de método abstracto
  - ➔ Implementación no definida
  - ➔ Necesidad de definirlo en las subclases
- Concepto de clase abstracta
  - ➔ Se puede usar como tipo de variable o argumento
  - ➔ Imposibilidad de crear objetos

# Ejemplo

- Crear la clase padre Poligono y las clases hijas Triangulo y Rectangulo
- Crear un método que calcule el área de un polígono

# Ejemplo

- La implementación de dicho método es distinta dependiendo de si el objeto es un triángulo o un rectángulo
- En resumen, tendríamos:
  - ➔ Un método calcularArea() aplicable a todos los objetos de la clase Poligono.
  - ➔ La implementación del método es completamente diferente en cada subclase de Poligono.
    - ★ Área de un triángulo.
    - ★ Área de un rectángulo.
    - ★ Etc.




Fórmula de Herón:

$$area = \sqrt{s * (s - a) * (s - b) * (s - c)}$$

$$s = (a + b + c) / 2$$



¿Cuál es tu trabajo ideal?

Haz el test aquí 

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Métodos abstractos

- 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 nombreMétodo(....);

- Luego en cada subclase se define un método con la misma cabecera y distinto cuerpo.



# Clases Abstractas

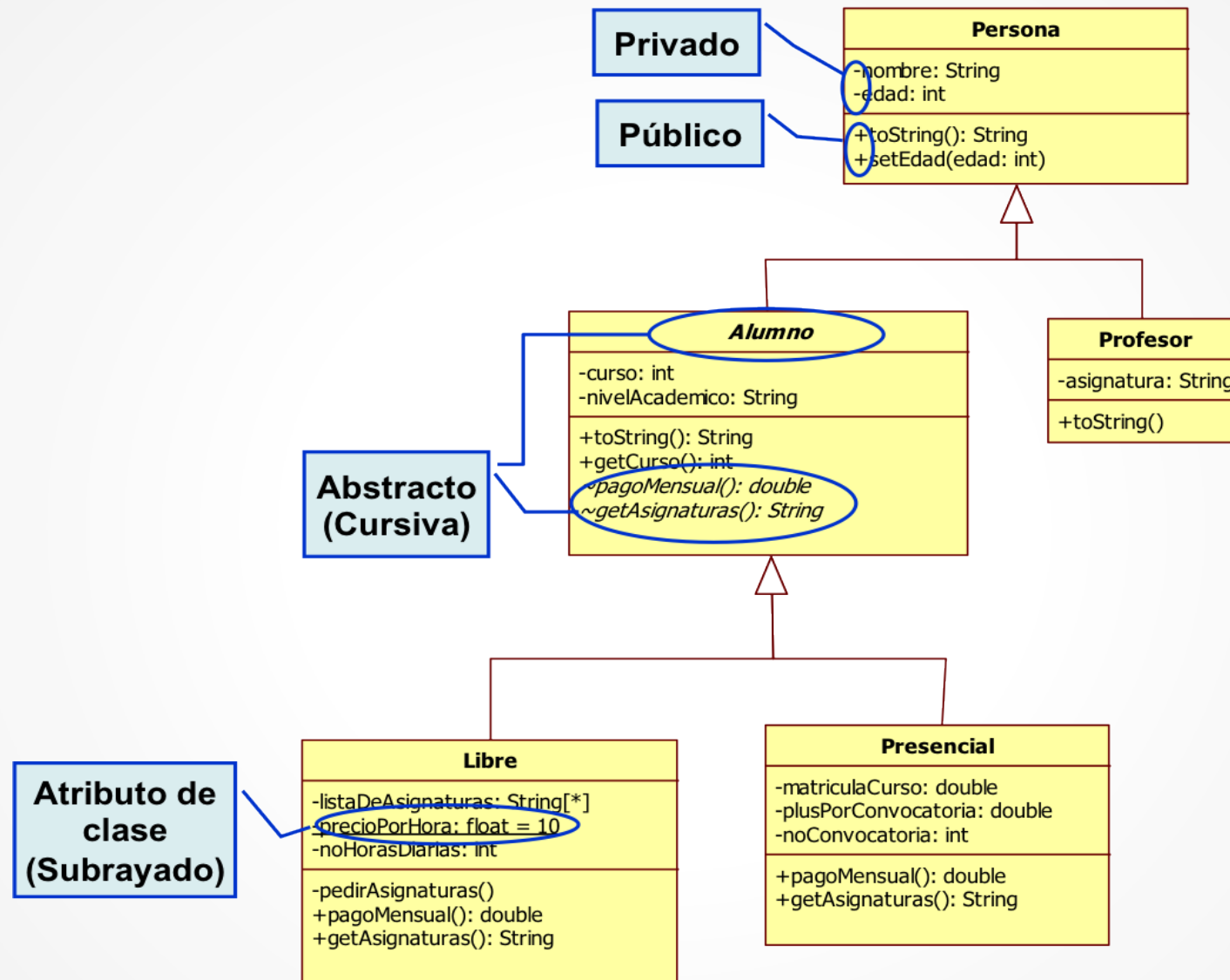
- Si una clase contiene al menos un método abstracto, entonces es una clase abstracta.
- Una **clase abstracta** es una clase de la que no se pueden crear objetos, pero puede ser utilizada como clase padre para otras clases.
- Declaración:

```
abstract class NombreClase {  
    .....  
}
```

# Ejemplo Alumnos y Profesor

- Crear las clases Persona, Alumno, Profesor, Libre y Presencial con los atributos y métodos de la figura siguiente.
- La clase Alumno tiene dos métodos, pagoMensual y getAsignaturas que son comunes a todos los objetos de tipo Alumno y cuya implementación depende de si es de tipo Libre o Presencial.
- El alumno Libre paga mensualmente por las horas de clase diarias de las asignaturas de las que está matriculado, mientras que el alumno Presencial paga el precio de la matrícula más un precio fijo por convocatoria (plusConvocatoria) multiplicado por el número de la convocatoria, todo ello dividido en los 12 meses del año.

# Ejemplo de clase abstracta





¿Cuál es tu trabajo ideal?

Haz el test aquí

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Ejemplo de clase abstracta

```
public abstract class Alumno extends Persona {  
    private int curso;  
    private String nivelAcademico;  
    public Alumno(String n, int e, int c, String nivel) {  
        super(n, e);  
        curso = c;  
        nivelAcademico = nivel;  
    }  
    public String toString() {  
        return super.toString() + curso + nivelAcademico;  
    }  
    public int getCurso() {  
        return curso;  
    }  
    public abstract double pagoMensual();  
    public abstract String getAsignaturas();  
}
```

# Ejemplo de clase abstracta

```
public class Libre extends Alumno {
    private String[] listaDeAsignaturas;
    private static float precioPorHora = 10;
    private int noHorasDiarias;
    public Libre(String n, int e, int c, String nivel,
        int horas){
        super(n, e, c, nivel); noHorasDiarias = horas;
        pedirAsignaturas();
    }
    //se inicializa listaDeAsignaturas
    private void pedirAsignaturas(){...}
    public double pagoMensual() {
        return precioPorHora * noHorasDiarias * 30;
    }
    public String getAsignaturas() {
        String asignaturas = "";
        for (int i = 0; i < listaDeAsignaturas.length; i++)
            asignaturas += listaDeAsignaturas[i] + ' ';
        return asignaturas;
    }
}
```

# Ejemplo de clase abstracta

```
public class Presencial extends Alumno {
    private double matriculaCurso;
    private double plusPorConvocatoria;
    private int noConvocatoria;

    public Presencial(String n, int e, int c, String nivel,
        double mc, double pc, int nc) {
        super(n, e, c, nivel);
        matriculaCurso = mc;
        plusPorConvocatoria = pc;
        noConvocatoria = nc;
    }

    public double pagoMensual() {
        return (matriculaCurso +
            plusPorConvocatoria * noConvocatoria) / 12;
    }

    public String getAsignaturas(){
        return "todas las del curso " + getCurso();
    }
}
```

# Ejemplo de clase abstracta

```
// FUNCIONES EN EL PROGRAMA DE PRUEBA
void mostrarAsignaturas(Alumno alumnos[]) {
    for (int i=0; i<alumnos.length; i++) {
        System.out.println(alumnos[i].getAsignaturas()); // enlace dinámico
    }
}

double totalPagos(Alumno alumnos[]) {
    double total=0;
    for (int i=0; i<alumnos.length; i++) {
        total += alumnos[i].pagoMensual(); // enlace dinámico
    }
    return total;
}
```



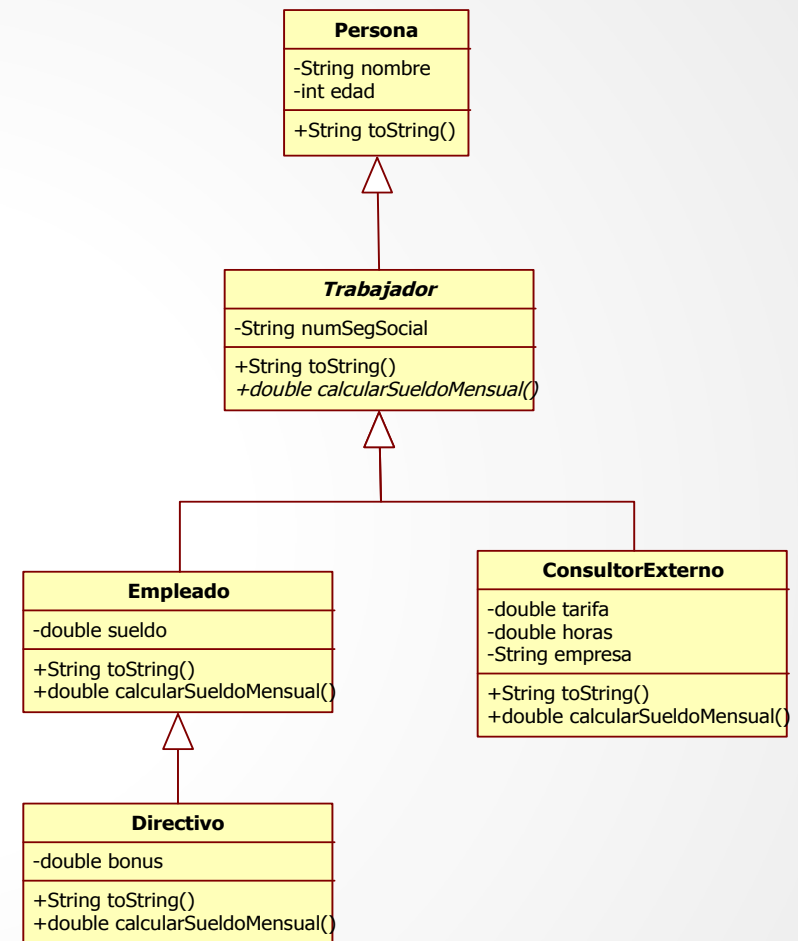


# ¿Harto de chapar algo que no te renta?

IRON  
HACK

## Ejercicio

- Añadir a la clase **Trabajador** un método abstracto *double calcularSueldoMensual()*
- Añadir a la clase **Empleado** un método *double calcularSueldoMensual()* que devuelve el sueldo dividido entre 14.
- Añadir a la clase **Directivo** un método *double calcularSueldoMensual()* que devuelve el sueldo dividido entre 14 y le suma su bonus.
- Añadir a la clase **ConsultorExterno** un método *double calcularSueldoMensual()* que devuelve *tarifa\*horas*.
- Implementar un programa principal (**main**) que haga lo siguiente:
  - ➔ Crear un vector de 4 trabajadores que contenga: 1 directivo, 2 empleados y 1 consultor externo.
  - ➔ Calcular el sueldo mensual total de las personas del vector y mostrarlo en pantalla.



¿Cuál es tu trabajo ideal?

Haz el test aquí

<http://bit.ly/necesitouncambio>

# Interfaces

WUOLAH

¿Harto de chapar algo que no te renta? Descubre tu trabajo ideal. ¡Haz el test!

# Contenido

- Introducción
- Definición
- Ejemplos de interfaces
- Interfaces vs clases abstractas

# Introducción

- Podría suceder que varias clases no necesariamente relacionadas compartan un mismo conjunto de operaciones
  - ➔ Una interfaz nos permite especificar un conjunto de operaciones
- Y dependiendo de la clase, cada operación se realice de diferente manera
- Ejemplo:
  - ➔ Clases: Parcela, Foto, Cuadro, EspejoCircular,...
  - ➔ Todas esas clases incluyen los métodos: calcularArea, calcularPerímetro, etc.
- Esto permite que podamos escribir métodos que sirvan para todas las clases que implementen una misma interfaz.
  - ➔ Ejemplo: Interfaz *Comparable*<T> y Arrays.Sort()



¿Cuál es tu trabajo ideal?  
Haz el test aquí



<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Introducción

- Podríamos definir una interfaz que agrupe todos los métodos comunes
  - Los métodos no se implementan completamente, únicamente se definen sus cabeceras
- Y luego definir varias clases que implementen los métodos del interfaz.

# Definición

En Java se hace a través de la definición de la interfaz

➔ **public interface** *Nombre* {.....}

- Suele incluir un conjunto de cabeceras de métodos (métodos abstractos).
  - ➔ La clase que implemente la interfaz tiene que implementar todos los métodos
- Una interfaz **no tiene atributos**
- Una interfaz puede incluir también definiciones de constantes públicas (static final)

# Definición

- Una misma clase puede implementar más de una interfaz ⇒ **Herencia múltiple de interfaces**
- Se pueden crear jerarquías de interfaces (con **extends**).
- A partir de la versión 1.8 de Java:
  - ➔ Se pueden definir **métodos por defecto** (**default**) que luego se pueden sobrescribir o no en las clases que implementen la interfaz.
  - ➔ Una interfaz **pueden poseer métodos de clase** (**static**)



# Ejemplo 1 de Interfaz

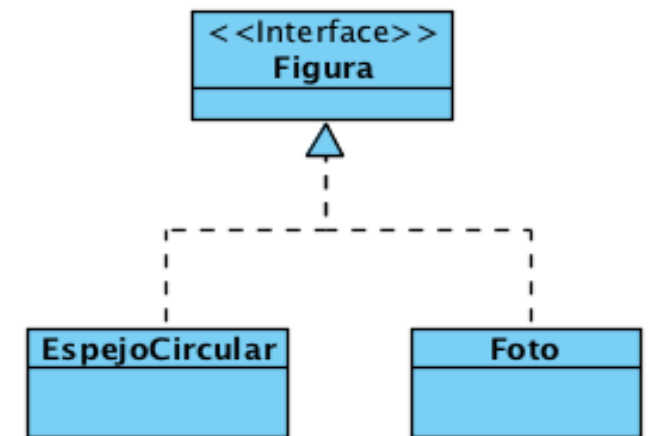
```
public interface Figura {  
    double area();  
    double perimetro();  
}
```

Define una  
interfaz

```
public class EspejoCircular implements Figura {  
    private double radio;  
    .....  
    public double area() {  
        return Math.PI*radio*radio; }  
    public double perimetro() {  
        return 2*Math.PI*radio; }  
}
```


Implementa una interfaz

```
public class Foto implements Figura {  
    private double lado1, lado2;  
    .....  
    public double area() { return lado1*lado2; }  
    public double perimetro() { return 2*(lado1+ lado2); }  
}
```





¿Cuál es tu trabajo ideal?

Haz el test aquí 

<http://bit.ly/necesitouncambio>

# ¿Harto de chapar algo que **no te renta?**

IRON  
HACK

## Ejemplo 1 de Interfaz

- Se pueden declarar referencias a objetos que implementen una cierta interfaz.
- Esto permite definir un método que sea aplicable a todos los objetos de clases que implementen una cierta interfaz.

```
double totalArea( Figura v[] ){  
    double t=0;  
    for (int i=0; i<v.length; i++)  
        t += v[i].area(); // enlace dinámico  
    return t;  
}
```

Array de instancias de clases  
que implementan la interfaz  
Figura

# Ejemplo 2: Ordenar → Comparar

- Para poder ordenar un array es necesario saber como comparar 2 elementos para averiguar cuál es mayor o si son iguales.

- ➔ Java proporciona la interfaz *Comparable<T>*

- ➔ Es necesario que una clase A que quiera poder comparar objetos de la propia clase A implemente esa interfaz así:

```
public class A implements Comparable<A> { ... }
```

- ➔ Esto obliga a tener implementado el método

- ★ **public int compareTo(A parametro)**

- **<0** si this es menor que parametro

- **>0** si this es mayor que parametro

- **=0** si son iguales

- El método que proporciona la API de Java que permite ordenar un array es:

- ➔ **Arrays.sort (Array de objetos)**

# Ejercicio

- Dada una clase Persona y una clase FechaComparable que implementa la interfaz `Comparable<FechaComparable>`
- Modificar la clase Persona de forma que implemente la interfaz `Comparable<Persona>`
  - ➔ La comparación se hará teniendo en cuenta sólo la edad.
- Escribir un programa principal que ejecute:
  - ➔ Crear un array de 5 personas y ordenar el array (con `Arrays.sort`)

# Interfaces vs clases abstractas

Interfaz	Clase abstracta
Una clase puede implementar múltiples interfaces	Una clase sólo puede extender a otra clase
Las clases que lo implementan no tiene por qué estar relacionadas entre sí	La clase que extiende a la abstracta tiene una relación <i>es-un</i> con ella y <i>es-hermano</i> con otras subclases de la abstracta
Sólo permite constantes públicas ( <b>static final</b> )	Permite atributos de clase y de instancia
Todos los métodos son públicos	Sus métodos no tienen por qué ser públicos