



Contenidos:

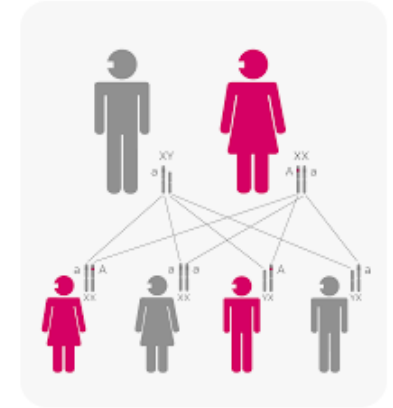
- Definición
- Tipos
- Declaración
- Qué se hereda
- Qué no se hereda
- Símil
- Override



Herencia familiar



Herencia de clases



Herencia genética

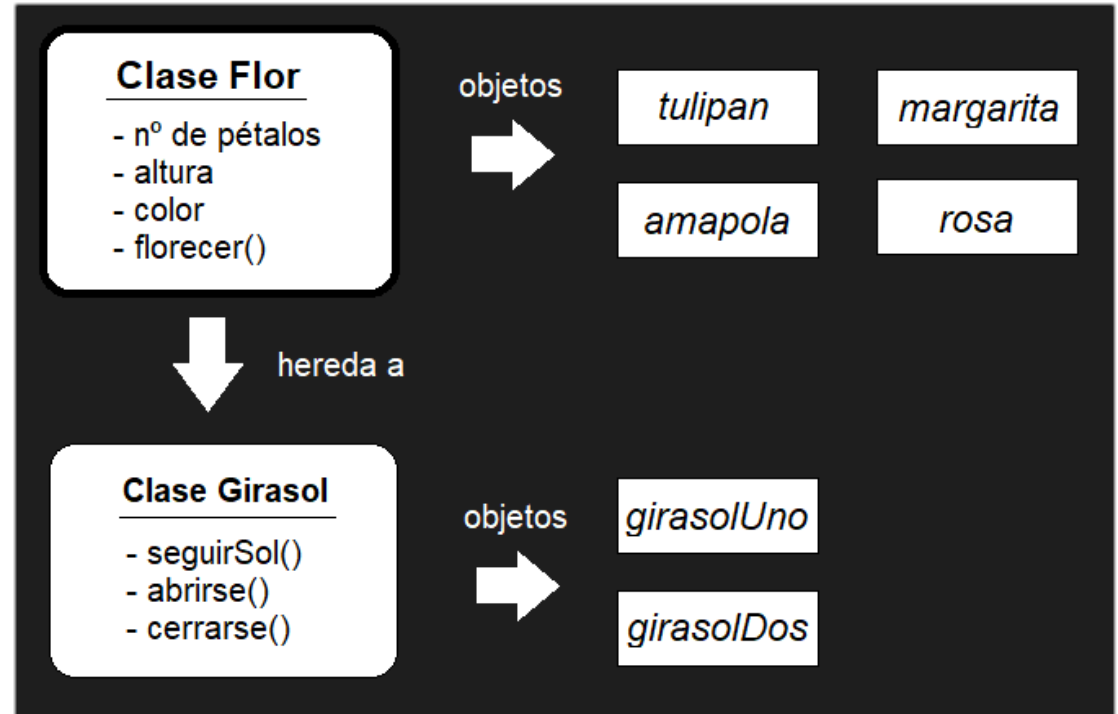


La herencia en Java implica que una **superclase**, o clase base, hereda sus funciones y atributos a una **subclase**, o clase derivada.

En otras palabras, consiste en que una clase padre **cede la definición de sus atributos y métodos** a una clase hija.

Esto es útil para **evitar repetir código**, reescribiendo los mismos atributos o métodos, en diferentes clases que pueden compartir **conceptos en común**.

Adicionalmente, la clase hija puede tener sus propios atributos y métodos adicionales.



Existen **cuatro tipos** de herencia en el mundo de la programación:

- **Simple:** Una clase hija hereda de una clase padre.
- **Multinivel:** Una clase hereda sus atributos a una clase hija, y esta también hereda sus atributos a otra clase.
- **Jerárquica:** Una clase padre hereda sus atributos a dos o más clases hijas.
- **Múltiple:** Una clase hija hereda de dos o más clases padre.

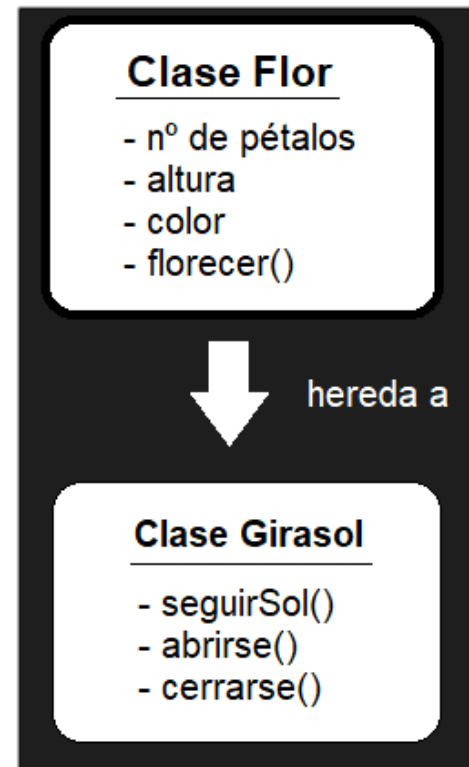




En la herencia de tipo **simple**, una clase hereda de una única clase.

En el mapa solo se tiene una clase padre y una clase hija.

Un ejemplo sencillo de herencia simple es el caso anterior de la **clase padre Flor** heredando sus atributos y métodos a la **clase hija Girasol**.

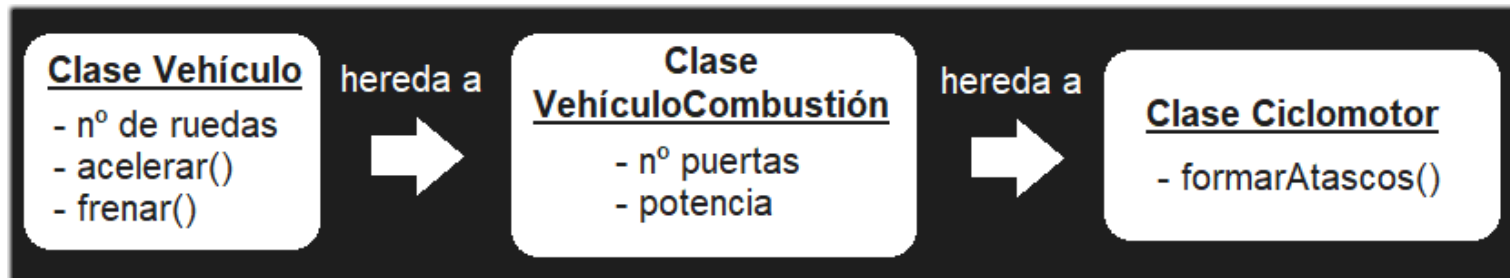




En la herencia de tipo **multinivel**, una clase hereda sus atributos a una clase hija, y esta también hereda sus atributos a otra clase.

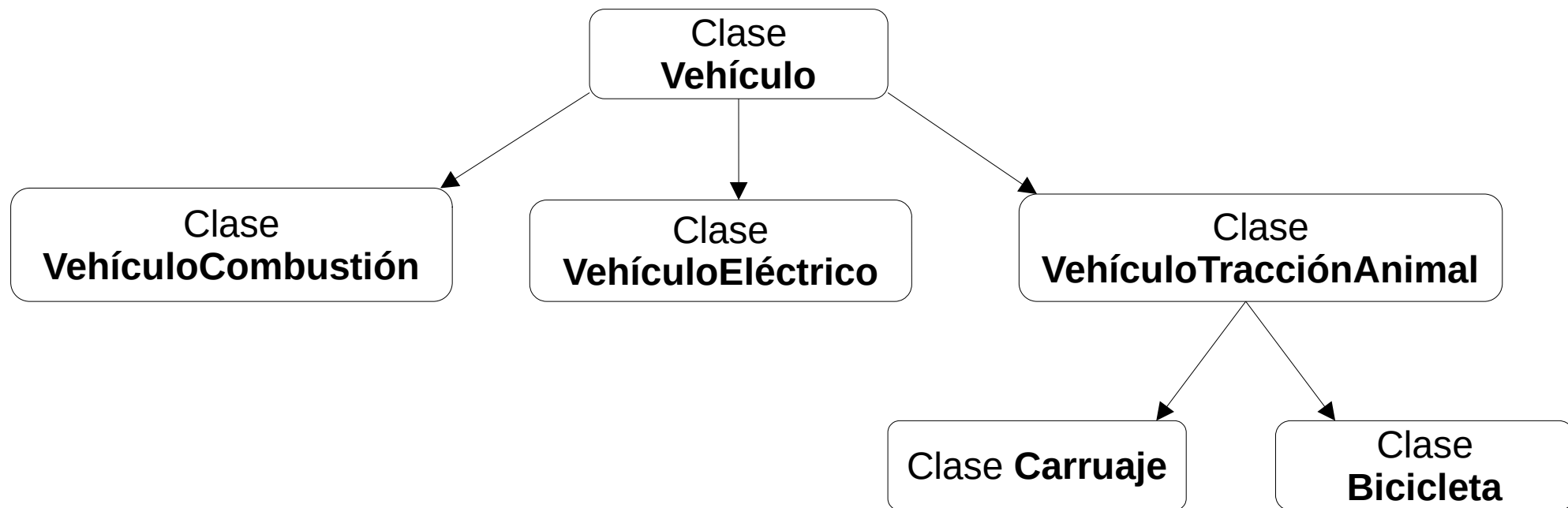
En el mapa tenemos una clase abuela, una clase padre y una clase hija.

Un ejemplo es el de una **clase base Vehículo**, que herede sus atributos y métodos a una clase **VehículoCombustión**, la cual a su vez hereda todos a la clase **Ciclomotor**.



En la herencia de tipo **jerárquica**, una clase padre hereda sus atributos a dos o más clases hijas, estableciendo así una jerarquía de clases.

En este caso el mapa se ramifica entre clases abuelas, padres e hijas.

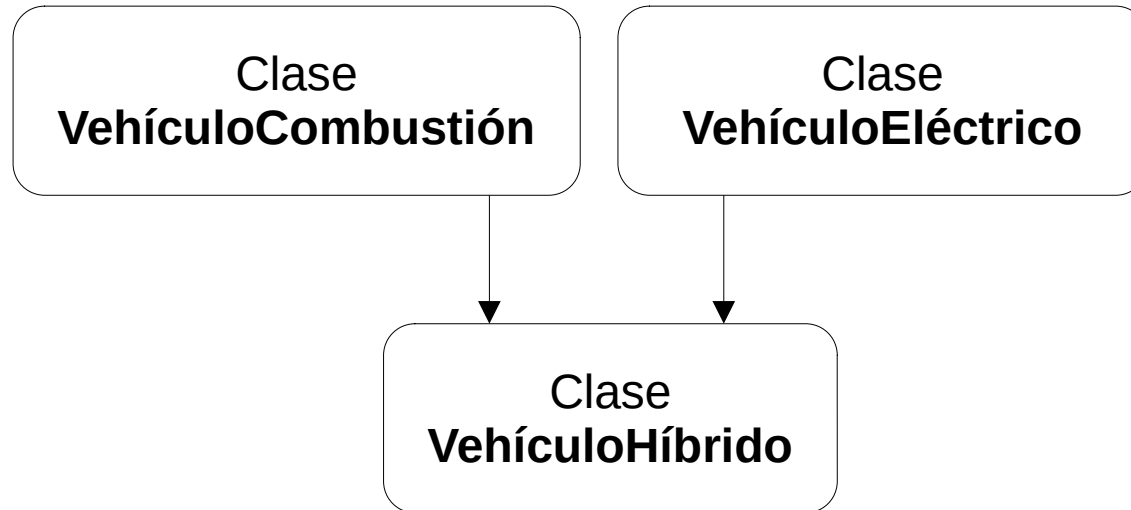




En la herencia de tipo **múltiple**, una clase hija hereda de dos o más clases padre.

La herencia múltiple no está permitida en Java, ya que una clase solo puede heredar directamente “extendiendo” de una sola clase.

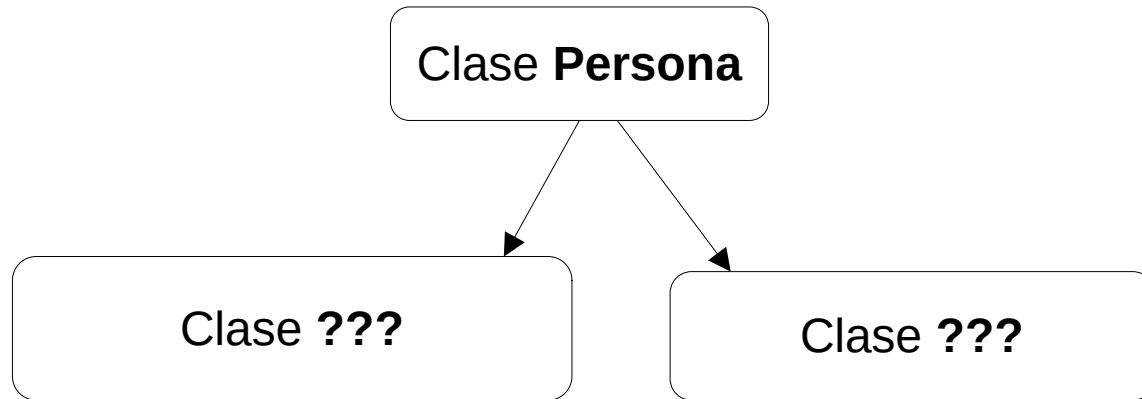
Sin embargo, en la siguiente unidad veremos formas de implementar características en común entre clases mediante el uso de interfaces.



Toca desarrollar un **organigrama**.


Empezando por una **clase Persona**, crea una **jerarquía de clases** con los distintos roles que se pueden encontrar entre las personas que forman la comunidad educativa del colegio: progenitores, estudiantes, profesores, personal de administración, personal de limpieza, ...

Asigna a cada uno de ellos los atributos o métodos que consideres necesario.





La herencia se realiza mediante la palabra clave **extends**.

```
src >  Sunflower.java > ...  
1  public class Sunflower extends Flower {  
2  
3      public void followSun() { }  
4      public void openUp() { }  
5      public void closeDown() { }  
6  
7  }  
8
```

¿Qué se hereda?



salesianos

COLEGIO SAN JUAN BOSCO LA CUESTA

Aquellos atributos o métodos declarados como **públicos** o **protegidos**.

```
public class Flower {  
  
    private int petalsNumber;  
    private float height;  
    private String color;  
  
    protected void flourish() { }  
  
    public int getPetalsNumber() {  
        return petalsNumber;  
    }  
}
```

No se
heredan

Sí se
heredan

¿Qué se hereda?



salesianos

COLEGIO SAN JUAN BOSCO LA CUESTA

Aquellos atributos o métodos declarados como **públicos** o **protegidos**.

```
public class Flower {  
  
    private int petalsNumber;  
    private float height;  
    private String color;  
  
    protected void flourish() { }  
  
    public int getPetalsNumber() {  
        return petalsNumber;  
    }  
}
```

```
public class Sunflower extends Flower {  
  
    public void followSun() { }  
    public void openUp() { }  
    public void closeDown() { }  
  
}
```

Nota: Los atributos o métodos protegidos actúan como si fueran privados.

Es decir, no se pueden llamar desde un objeto. Solo puede usarlos la clase hija.

¿Qué se hereda?



salesianos

COLEGIO SAN JUAN BOSCO LA CUESTA

Los atributos declarados como **constantes** con la palabra clave “final” pueden ser heredados, pero no pueden ser modificados en la clase hija.

```
public class Flower {  
  
    private int petalsNumber;  
    private String color;  
  
    protected final String BEST_SEASON = "Spring";  
  
    public Flower(int petals, String flowerColor) {  
        this.petalsNumber = petals;  
        this.color = flowerColor;  
    }  
}
```

```
public class Sunflower extends Flower {  
  
    public Sunflower(int petals, String flowerColor) {  
        super(petals, flowerColor);  
        System.out.println(this.BEST_SEASON);  
    }  
}
```

Nota: Como se aprecia en la imagen, puede accederse a los atributos o métodos protegidos que han sido heredados mediante **this**.



A partir de la jerarquía de clases de la actividad anterior, crea un proyecto Java llamado **HerenciaColegio** e implementa la clase **Persona** que diseñaste.

A continuación, sin crear constructores, crea algunas de las otras clases y haz que extiendan de la clase Persona para heredar sus atributos.

Recuerda mantener una estructura de paquetes adecuada.

Ve a la clase ejecutable y crea algunas instancias de cada una de las clases que has implementado.



¿Qué NO se hereda?



salesianos

COLEGIO SAN JUAN BOSCO LA CUESTA

Los **constructores** de la clase no se heredan.

Sin embargo, estos pueden ser llamados desde el constructor de la clase hija usando la palabra clave **super** y pasando como parámetros aquellos que tenga el constructor del padre.

```
public class Flower {  
  
    private int petalsNumber;  
    private String color;  
  
    public Flower(int petals, String flowerColor) {  
        this.petalsNumber = petals;  
        this.color = flowerColor;  
    }  
  
    protected void flourish() { }
```

```
public class Sunflower extends Flower {  
  
    public Sunflower(int petals) {  
        super(petals, "yellow");  
    }  
  
    public void followSun() { }  
    public void openUp() { }  
    public void closeDown() { }  
}
```

¿Qué NO se hereda?



salesianos

COLEGIO SAN JUAN BOSCO LA CUESTA

Con **super** estamos llamando al constructor de la clase Flower y, por lo tanto, guardando el valor “yellow” en el atributo “color” y el número de pétalos en el atributo “petalsNumber”.

Al acceder con el getter heredado, obtendremos “yellow”.

```
public class Flower {  
  
    private int petalsNumber;  
    private String color;  
  
    public Flower(int petals, String flowerColor) {  
        this.petalsNumber = petals;  
        this.color = flowerColor;  
    }  
  
    protected void flourish() { }
```

```
public class Sunflower extends Flower {  
  
    public Sunflower(int petals) {  
        super(petals, "yellow");  
    }  
  
    public void followSun() { }  
    public void openUp() { }  
    public void closeDown() { }  
}
```

¿Qué NO se hereda?



salesianos

COLEGIO SAN JUAN BOSCO LA CUESTA

Llamar al constructor de la clase antecesora será **obligatorio** siempre que declaremos un constructor en la clase que extiende.

```
public class Sunflower extends Flower {  
    public Sunflower(int petals, String flowerColor) {  
        Sunflower.Sunflower(int petals, String  
        flowerColor)  
    }  
    public  
    public  
    public  
}
```

Implicit super constructor Flower() is undefined.
Must explicitly invoke another
constructor Java(134217871)

[View Problem \(Alt+F8\)](#) No quick fixes available



Extender de **Flower**, sería similar a declarar la clase **Sunflower** como se muestra en la imagen.

Nótese que los atributos privados de la clase **Flower** no deberían estar en **SunFlower**.

Pero, para poder escribir el constructor y los métodos de acceso sin que el IDE marcara errores, había que declararlos.

Este ejemplo sería el resultado de tener los atributos definidos como **protected** en la clase padre.

```
1 public class Sunflower {
2
3     private int petalsNumber;
4     private String color;
5
6     public Sunflower(int petals) {
7         this.petalNumber = petals;
8         this.color = "yellow";
9     }
10
11     public void followSun() { }
12     public void openUp() { }
13     public void closeDown() { }
14
15     private void flourish() { }
16
17     public int getPetalNumber() {
18         return petalsNumber;
19     }
```



A partir de la jerarquía de clases implementadas en la actividad anterior, **añade un constructor a la clase Persona.**

A continuación, refactoriza el resto de clases que extienden de la clase Persona. Finalmente, lanza de nuevo la clase ejecutable y comprueba que todo esté bien.





Como vimos anteriormente, la **sobreescritura** o polimorfismo de inclusión es una característica que permite a las clases secundarias o subclasses hacer una implementación de un método que ha sido dado por una clase principal o superclase.

Es decir, una clase padre puede tener declarado un método con una funcionalidad. Sin embargo, la clase hija puede requerir algunas modificaciones para ir más acorde a lo que representa.

Es importante entender que todas las clases de Java extienden por defecto de la clase **Object**. Osea, que es lo mismo escribir:

```
public class Flower extends Object { }
```



Por ejemplo, como todas las clases de Java heredan por defecto de la clase **Object**. En esta clase, el método **toString** está implementado para devolver una referencia en memoria del objeto.

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

Sin embargo, a veces interesa **sobreescribir** ese método para que devuelva una información más legible para nosotros.

```
@Override  
public String toString() {  
    return "{ petalsNumber:" + petalsNumber + ", color," + color + ", BEST_SEASON," + BEST_SEASON + "}";  
}
```

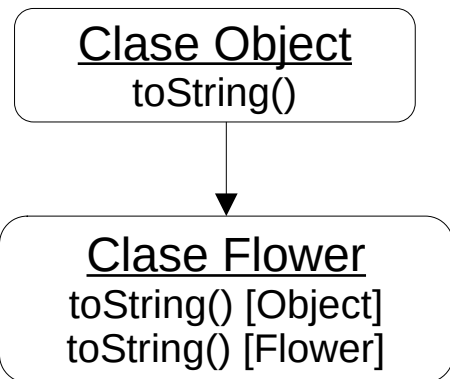
Para sobrescribir un método, la anotación **@Override** es importante.

Si la obvias, la definición de la clase hija oculta el toString de la clase **Object**, haciendo que coexistan los dos métodos en memoria.

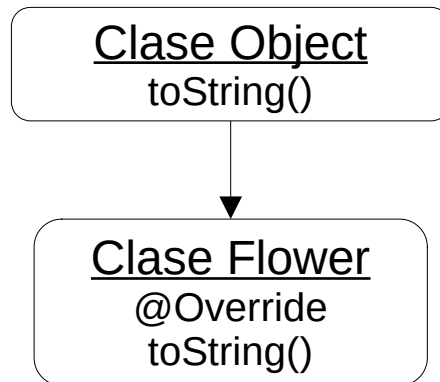
Aún así, sólo es accesible el que hemos declarado en la clase **Flower**.

Si añadimos la anotación, le indicamos a Java que se olvide de la definición del método heredado porque queremos definir una más apropiada para nuestro caso de uso.

Sin Override



Con Override





Busca la manera de entrar en la clase `Object` y localiza las definiciones de los métodos **`equals()`** y **`toString()`**.

Una vez hecho esto, ve a la clase `Persona` de las actividades anteriores y sobrescribe dichos métodos usando la anotación **`Override`**.

Ve a la clase ejecutable y utiliza estos dos métodos con la nueva definición que has indicado a la clase.

Finalmente, ejecuta el programa para comprobar que funciona correctamente.





También puedes **sobreescribir métodos propios definidos** en la clase antecesora.

```
public class Bird {  
    protected void sing() {  
        String song = "Chirp chirp chidori";  
        System.out.println(song);  
    }  
}
```

```
public class Chicken extends Bird {  
    @Override  
    protected void sing() {  
        System.out.println("Pooo po po pooo");  
    }  
}
```

```
public class Parrot extends Bird {  
    @Override  
    protected void sing() {  
        System.out.println("I'm a preeetty parrot, OARK!");  
    }  
}
```

Creando los objetos necesarios y ejecutando el programa de la **clase ejecutable**, se tiene:

```
public class App {  
    Run | Debug  
    public static void main(String[] args) {  
        Bird berd = new Bird();  
        berd.sing();  
  
        Chicken chickenLitel = new Chicken();  
        chickenLitel.sing();  
  
        Parrot jackEsParrot = new Parrot();  
        jackEsParrot.sing();  
    }  
}
```



Output:

```
Chirp chirp chidori  
Pooo po po pooo  
I'm a preeetty parrot, OARK!
```





Cabe destacar que Java **no** permite la **reducción de visibilidad** de un elemento heredado, siendo el orden de visibilidad: **public > protected > private**.

```
java Parrot.java 1
Parrot.java > ...
public class Parr
{
    @Override
    private void sing() {
        System.out.println("I'm a preeetty parrot, OARK!");
    }
}
```

void Parrot.sing()

Cannot reduce the visibility of the inherited method from Bird Java(67109273)

View Problem (Alt+F8) Quick Fix... (Ctrl+.)



Sin embargo, **sí** permite **aumentar el grado de visibilidad** de un método heredado, siendo el orden de visibilidad: **public > protected > private**.

```
public class Parrot extends Bird {  
    @Override  
    public void sing() {  
        System.out.println("I'm a preeetty parrot, OARK!");  
    }  
}
```



Crea una clase **Factura** que tenga dos atributos protegidos **id** e **importe**, y un método público **calcularTotal** que retorne el valor del importe.

Añade un constructor que inicialice sendos atributos.

Crea dos clases **FacturaPeninsular** y **FacturaCanaria**, y haz que hereden de la clase **Factura**.

Sabiendo que el **IGIC** es de un 7% y que el **IVA** es de un 21%, sobrescribe el método **calcularTotal** en cada clase hija para que retorne el importe final con su impuesto correspondiente.

Crea una nueva clase ejecutable, instancia un objeto de cada tipo de factura con el mismo importe y muestra el valor retornado en cada caso por el método **calcularTotal()**.





¿Sabías que...?

Godot es un motor de videojuegos 2D y 3D multiplataforma, libre y de código abierto.

Este contiene numerosos elementos llamados Nodos.

Cada uno de estos nodos hereda propiedades de un nodo padre, como Node2D o Node3D, estableciendo una jerarquía de clases.

