

# Programación orientada a objetos

## Herencia y polimorfismo

Elizabeth León Guzmán, Ph.D.

[eleonguz@unal.edu.co](mailto:eleonguz@unal.edu.co)

Jonatan Gómez Perdomo, Ph. D.

[jgomezpe@unal.edu.co](mailto:jgomezpe@unal.edu.co)

Arles Rodríguez, Ph.D.

[aerodriguezp@unal.edu.co](mailto:aerodriguezp@unal.edu.co)

Camilo Cubides, Ph.D. (c)

[eccubidesg@unal.edu.co](mailto:eccubidesg@unal.edu.co)

Carlos Andres Sierra, M.Sc.

[casierrav@unal.edu.co](mailto:casierrav@unal.edu.co)

Research Group on Data Mining – Grupo de Investigación en Minería de Datos – (Midas)

Research Group on Artificial Life – Grupo de Investigación en Vida Artificial – (Alife)

Computer and System Department

Engineering School

Universidad Nacional de Colombia

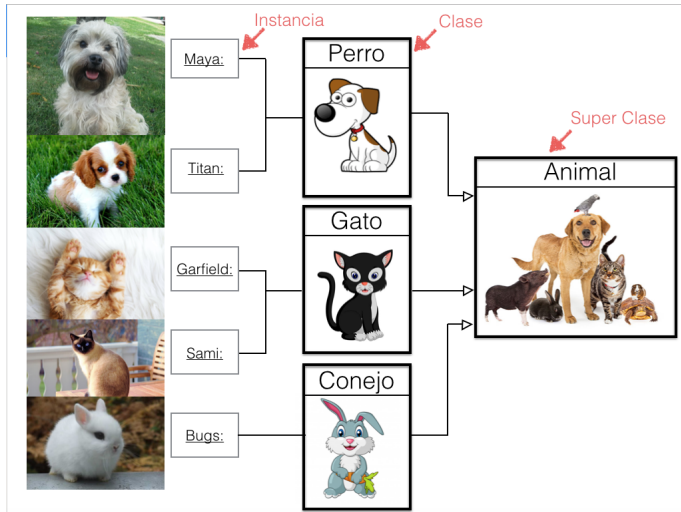
# Agenda

- 1 Clases
- 2 Herencia
- 3 Modificadores
- 4 Polimorfismo
- 5 Interfaces



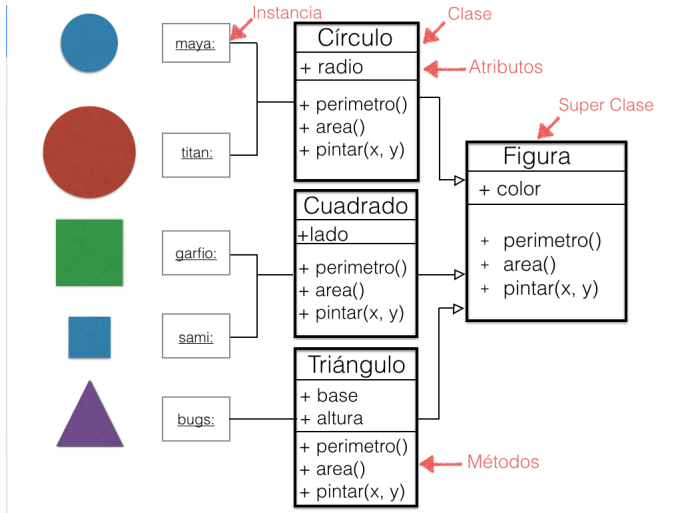
# Clases

## Ejemplo Los Animales



# Clases

## Figuras Geométricas



# Agenda

- 1 Clases
- 2 Herencia
- 3 Modificadores
- 4 Polimorfismo
- 5 Interfaces



# Herencia

La **herencia** es una de las características principales de la programación orientada a objetos.

Permite definir nuevas clases de objetos (**subclases**) que heredan los atributos y métodos que han sido definidos en la clase (**super-clase**).

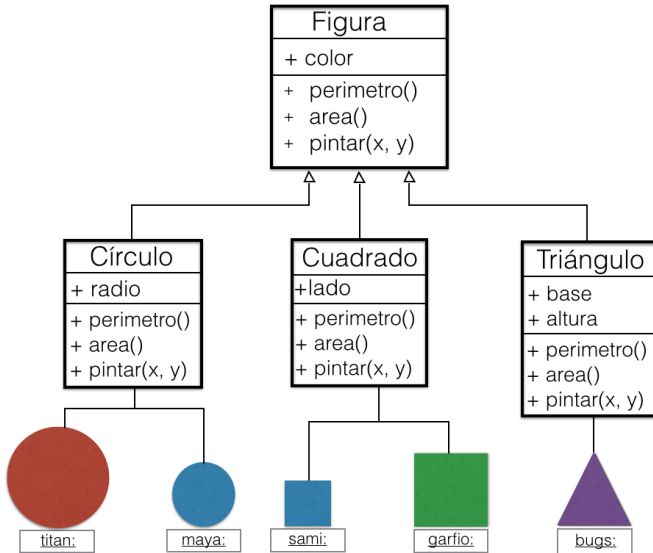
El término herencia se refiere a la capacidad que tienen los objetos de las subclases de utilizar como propios (dependiendo del nivel de acceso) los atributos y métodos definidos en su linaje, esto es, en su super-clase, en la super-clase de su super-clase, ...

La herencia permite a los programadores la reutilización de código desarrollado con anterioridad.



# Clases

## Figuras Geométricas



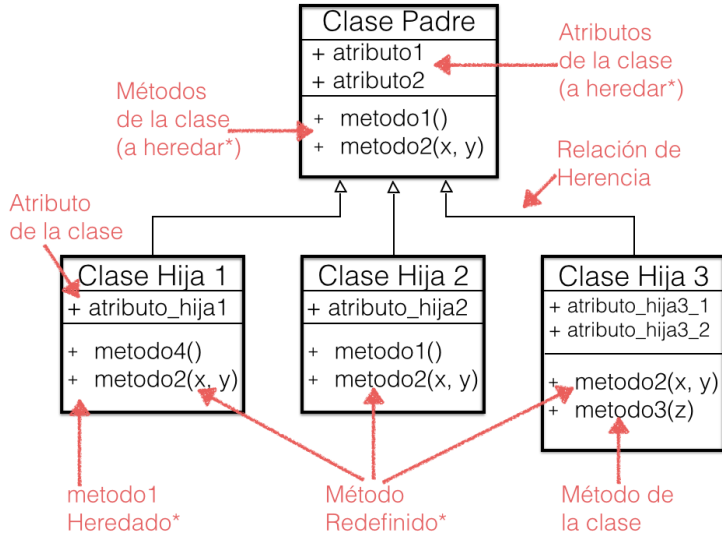
# Herencia (I)

- La clase principal (super clase) también se conoce como **padre** o **base**.
- La clase que hereda (subclase) se conoce como **hija** o **derivada**.
- Al conjunto de relaciones de herencia se le conoce como **Jerarquía de Clases** el cual puede verse como un árbol de relaciones con una flecha "hueca" ( $\rightarrow$ ) de la clase hija a la padre.
- Las clases hijas pueden heredar de la clase padre todas sus características tal como estén definidas (dependiendo del nivel de acceso), sin embargo en caso de ser necesario se pueden modificar aquellas características que lo requieran.





# Herencia (II)



# Ejemplo

## Vehículos (I)

Tomemos como ejemplo la clase “Vehículo” la cual cuenta, entre otros, con los atributos “Motor” y “Ruedas” además de los métodos “Moverse” y “Frenar”. Podemos entonces crear dos clases adicionales, la clase “Motocicleta” y la clase “Transmilenio”. Estas subclases heredan los atributos y métodos de la clase “Vehículo” por lo que no es necesario volver a definirlos.



# Ejemplo

## Vehículos (II)

Hasta el momento las clases “Motocicleta” y “Transmilenio” contienen exactamente los mismos atributos y métodos, sin embargo, el que hereden de una misma clase no les impide tener atributos y métodos propios. Por ejemplo se puede añadir el atributo “ruta” a la clase “Transmilenio” (este atributo es específico de la clase Transmilenio y no afecta ni a la clase “Vehículo” ni a la clase “Motocicleta”)



# Herencia (III)

Como podemos notar, cada jerarquía de clases puede llegar a ser única y se define dependiendo de las necesidades del programa.

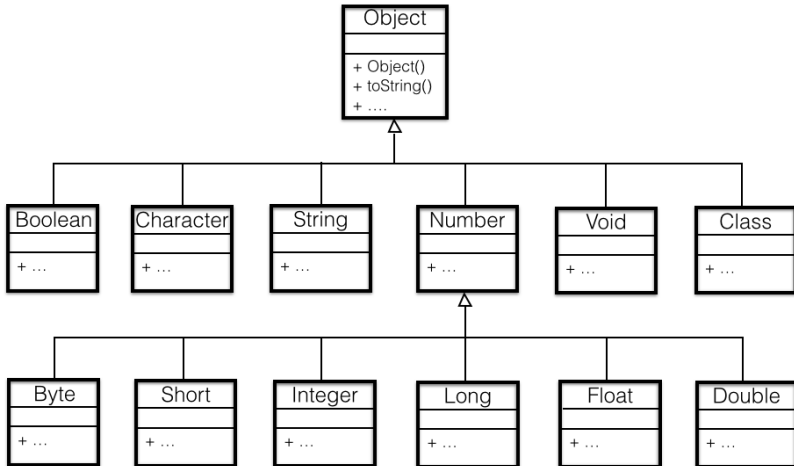
Sin embargo, algunas clases de objetos son estándar pues están muy bien definidas, lo que permite agruparlas en librerías.

En Java todas las clases heredan o derivan de la superclase `Object`.



# Herencia (IV)

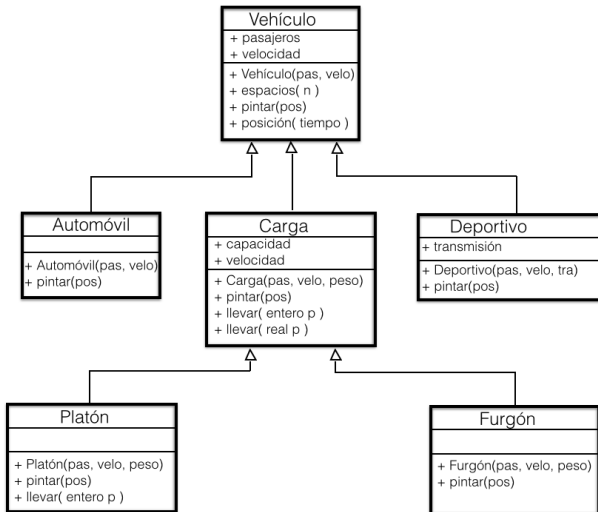
## Classes Java



# Vehículos (I)

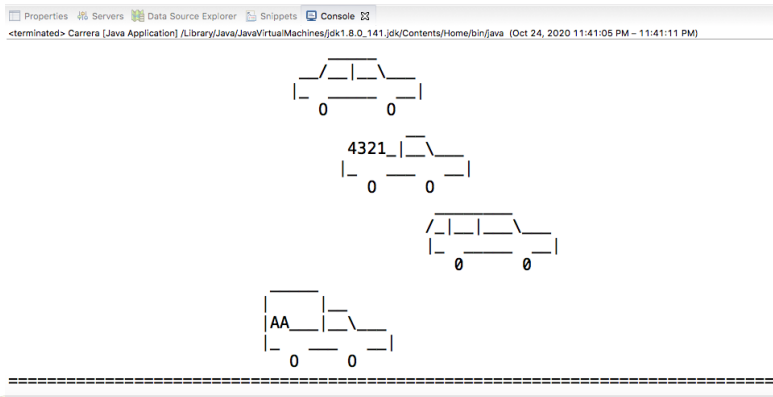
## Versión super-simplificada

Código disponible en <https://github.com/jgomezpe/carros>



# Vehículos (II)

Al ejecutarlo trate que la consola de su ambiente quede del alto mostrado en la imagen, verá una animación!!



```
<terminated> Carrera [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_141.jdk/Contents/Home/bin/java (Oct 24, 2020 11:41:05 PM - 11:41:11 PM)
```

0

4321

0

AA



# Vehículos (III) - Clase Vehículo

```
public abstract class Vehiculo {  
    protected double velocidad;  
    protected int pasajeros;  
    public Vehiculo(int pasajeros, double velocidad) {  
        this.velocidad = velocidad;  
        this.pasajeros = pasajeros;  
    }  
    public int posicion( int tiempo ) {  
        return (int)(tiempo*velocidad);  
    }  
    public void espacios(int espacios) {  
        for( int i=0;i<espacios;i++)  
            System.out.print(' ');  
    }  
    public abstract void pintar(int posicion);  
}
```





# Vehículos (IV) - Clase Automovil

```
public class Automovil extends Vehiculo {  
    public Automovil(int pasajeros, double velocidad) {  
        super(pasajeros, velocidad);  
    }  
    @Override  
    public void pintar(int posicion) {  
        espacios(posicion+4);  
        System.out.println("____");  
        espacios(posicion+1);  
        System.out.println("__/_|_\\\\_");  
        espacios(posicion);  
        System.out.println("|_  ____  _|");  
        espacios(posicion);  
        System.out.println("    0      0");  
    }  
}
```



# Herencia (V)

Se define que una clase es hija de otra usando la palabra reservada `extends`.

Una subclase normalmente requiere se ejecute el constructor de la superclase antes que su propio constructor para inicializar las variables de instancia heredadas. Para esto se utiliza la palabra reservada `super` seguida entre paréntesis de los parámetros correspondientes en el cuerpo del constructor de la clase padre.

```
class Hija extends Padre {  
    ...  
    public Hija(arg1, arg2, .... ) {  
        super(arg1, arg2, ... );  
        ...  
    }  
    ...  
}
```



# Herencia (super)

## Pregunta

¿Usa el constructor de la clase `Automovil` el constructor de la clase `Vehiculo`?



# Agenda

- 1 Clases
- 2 Herencia
- 3 Modificadores**
- 4 Polimorfismo
- 5 Interfaces



# Modificadores

Existen cinco tipos de modificadores en Java, no todos usables al mismo tiempo, ni para todo elemento. El orden de algunos de los modificadores se puede cambiar.

- De acceso: para atributos, metodos y clases.
- De concreción: para clases y métodos en clases.
- De alcance: para atributos y métodos en clases.
- De sellamiento: para clases y atributos y métodos en clases.
- De omisión: para métodos en interfaces (a ver en la sección de interfaces).

<acceso> <alcance> <concreción> <sellamiento> <omisión>



# Modificadores de Acceso (I)

Los modificadores de acceso definen qué clases pueden acceder a un atributo, método o clase, es decir, definen la forma en que el programa accede a la información de cada clase. En particular, los modificadores de acceso afectan a los métodos y los atributos a los que se puede acceder dentro de una jerarquía de herencia.

```
<acceso> definición_clase  
<acceso> definición_atributo  
<acceso> definición_método
```



# Modificadores de Acceso (II)

## Tipos

Existen cuatro formas diferentes para modificar el acceso:

**private:** Sólo serán accesibles dentro de la misma clase, es decir, no se permite el acceso a estos elementos desde ninguna otra clase.

**protected:** Visibles dentro de la misma clase, por todas las clases dentro del mismo paquete y por todas las subclases o clases derivadas.

**public:** Accesibles desde cualquier otra clase.

**Sin modificador:** Cuando no se usa ningún modificador. El elemento sólo será visible dentro de la misma clase y por todas las demás clases dentro del mismo paquete.



# Modificadores de Acceso (III)

## Resumen Tipos

Modificador	Clase	Subclase	Paquete	Todos
public	si	si	si	si
protected	si	si	si	no
private	si	no	no	no
Ninguno	si	no	si	no

Table: Resumen Tipo Modificador Acceso.





# Modificador de Concreción (abstract)

Usado para determinar cuando una clase está diseñada o creada sólo para ser clase padre, es decir, de la cual no se pueden crear instancias (clases **no concretas** o **abstractas**). Aunque puede tener definidos métodos y atributos **concretos**, es decir, totalmente definidos, se distingue por tener métodos abstractos.

```
<acceso> abstract class Clase {  
    ...  
    <acceso> abstract definicion_método;  
    ...  
}
```



# Modificador de concreción (static)

## Preguntas

¿Es la clase Vehiculo abstracta?

¿Es la clase Automovil abstracta?

¿Existe un método abstracto en la clase Vehiculo?

¿Existe un método abstracto en la clase Automovil?

¿Cuáles métodos de la clase Vehiculo son concretos?

¿Cuáles métodos de la clase Automovil son concretos?



# Vehículos (V) - Clase Carrera

```
public class Carrera {  
    public static void pausar() {  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args) {  
        Automovil automovil = new Automovil(5, 1);  
        ...  
    }  
}
```



# Modificador de Alcance (static)

Determina cuando un atributo y/o método es de la clase (¡Si las clases también son objetos en Java y son instancias de la clase `Class`, genial!) o de las instancias. Un atributo y/o método **estático** puede ser accedido preguntándole a la clase sin necesidad de crear una instancia. Toda instancia de la clase lo puede usar. No se puede definir un método `abstract` y `static` a la vez.

```
class Clase {  
    <alcance> definicion_atributo  
    ...  
    <alcance> definicion_método  
    ...  
}
```



# Modificador de alcance (static)

## Preguntas

- ¿Cuál es el modificador de alcance de los métodos de la clase Carrera?
- ¿Es el atributo out en la clase System estático?
- ¿Es el método sleep en la clase Thread estático?
- ¿Es el método espacios en la clase Vehiculo estático?



# Modificador de Sellamiento (final)

Tiene tres usos principales: en definición de clases, en definición de métodos y en definición de atributos y argumentos.

- **Clases Selladas:** Son aquellas clases que no pueden ser utilizadas como clase padre, es decir no se puede heredar de ellas. Debido a esto no pueden ser abstractas (son modificadores excluyentes).
- **Métodos sellados:** Son métodos que no pueden ser sobre-escritos por clases hijas.
- **Atributos y argumentos sellados:** Son atributos y/o argumentos cuyos valores no pueden ser cambiados.



# Modificador de Sellamiento (final)

```
class <sello> Clase {  
    <sello> definicion_atributo  
    ...  
    <sello> definicion_método  
    ...  
    método(<sello> argumento, ... ) {  
        ...  
    }  
    ...  
}
```



# Agenda

- 1 Clases
- 2 Herencia
- 3 Modificadores
- 4 Polimorfismo**
- 5 Interfaces





# Polimorfismo

Según la Real Academia Española (<https://dle.rae.es/polimorfo>),

**Polimorfo** es que tiene o puede tener distintas formas.

En programación significa que un método puede tener distintas formas.

Esto debido a la capacidad de sobre-escribir los métodos cuando se realiza herencia (una clase hija puede redefinir los comportamientos) o por el tipo de los argumentos que recibe. Al primer tipo de polimorfismo se le llama **Dinámico** o por herencia y al segundo **estático** o por mensaje. A un método polimórfico también se le dice **sobre-cargado**.



# Vehículos (VI) - Clase Carga

```
public class Carga extends Vehiculo {  
    protected int capacidad;  
    protected Object carga;  
    public Carga(int pasa, double vel, int peso){ ... }  
    public void pintar(int posicion){ ... }  
    public boolean llevar(int p) {  
        int k = 1;  
        for(int i=0; i<this.capacidad; i++) { k *=10; }  
        if(p < k) {  
            this.carga = p;  
            return true;  
        } else {  
            return false;  
        }  
    }  
    public boolean llevar( String p){ ... }  
}
```



# Vehículos (VII) - Clase Platon

```
public class Platon extends Carga {  
    public Platon(int pasa, double vel, int peso){ ... }  
    public void pintar(int posicion){ ... }  
    public boolean llevar(int p) {  
        boolean lolleva = super.llevar(p);  
        if(lolleva) {  
            String laCarga = this.carga.toString();  
            String cargaInvertida = "";  
            for(int i=laCarga.length()-1; i>=0; i--) {  
                cargaInvertida += laCarga.charAt(i);  
            }  
            this.carga = cargaInvertida;  
        }  
        return lolleva;  
    }  
}
```



# Vehículos (VIII) - Clases Furgon y Deportivo

```
public class Furgon extends Carga {
    public Furgon(int pasa, double vel, int peso) {
        ...
    }
    public void pintar(int posicion) {
        ...
    }
}

public class Deportivo extends Vehiculo {
    public Deportivo(int pasajeros, double velocidad) {
        ...
    }
    public void pintar(int posicion) {
        ...
    }
}
```



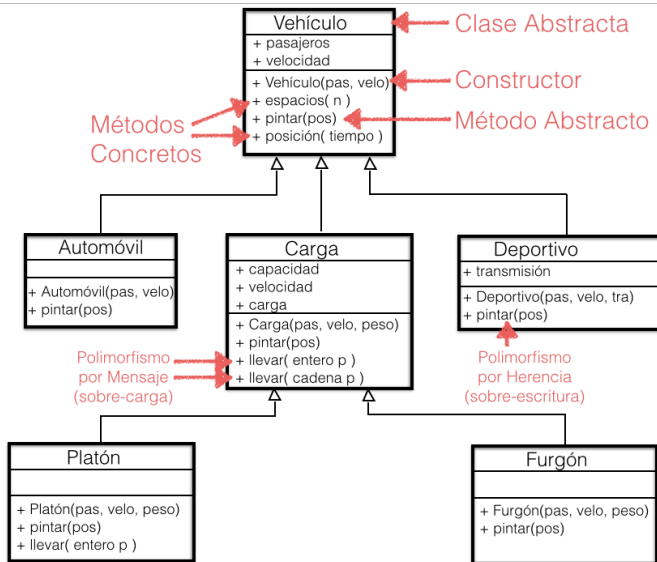
# Polimorfismo

## Preguntas

- ¿Cuáles métodos sobre-carga la clase Automovil?
- ¿Cuáles métodos sobre-carga la clase Furgon?
- ¿Cuáles métodos sobre-carga la clase Platon?
- ¿Cuáles métodos sobre-carga la clase Carga?
- ¿Cuáles métodos sobre-carga la clase Deportivo?



# Vehículos (IX)



# Agenda

- 1 Clases
- 2 Herencia
- 3 Modificadores
- 4 Polimorfismo
- 5 Interfaces**



# Interfaces I

Otra forma de lograr abstracción en Java es mediante el uso de interfaces. Una **interfaz** es una lista de acciones que puede llevar a cabo un determinado objeto, es decir, una interfaz es un grupo de métodos relacionados entre sí, que en general sólo contienen el **prototipo del método**, no su código.





# Interfaces II

En Java, una interfaz se define con la palabra reservada `interface` como se muestra a continuación:

## Ejemplo

```
// interfaz
interface Animal {
    public void sonidoAnimal(); /* método de interfaz
                                (prototipo, no tiene cuerpo)*/
    public void dormir(); /* método de interfaz
                           (prototipo, no tiene cuerpo)*/
}
```



# Interfaces III

Para acceder a los métodos de una interfaz, ésta debe ser "implementada" por una clase. Para ello se usa la palabra reservada `implements` (similar a como se hace con `extends` en herencia). El cuerpo de los métodos se define dentro de la clase que lo implementa:

## Ejemplo

```
// la clase Perro "implementa" la interfaz Animal
class Perro implements Animal {
    public void sonidoAnimal() { // Cuerpo de sonidoAnimal()
        System.out.println("El perro hace: wow wow");
    }
    public void dormir() { // Cuerpo de dormir()
        System.out.println("Zzz");
    }
}
```



# Características de las interfaces

- Las interfaces no pueden ser usadas para crear objetos (en el ejemplo anterior no es posible crear un objeto Animal).
- Los métodos de las interfaces usualmente no poseen cuerpo, éste se define en las clases que lo implementen.
- Cuando se implementa una interfaz, todos sus métodos (excepto aquellos definidos por omisión, deben ser definidos.
- Una interfaz no puede contener un constructor ya que no se pueden usar para crear objetos.



# Variables en interfaces

Las variables se pueden declarar en una interfaz, pero son implícitamente públicas, estáticas y finales (`public`, `static`, y `final`). Su uso se limita principalmente a definir constantes (se acostumbra a escribir las constantes en mayúsculas):

## Ejemplo

```
//Una interfaz que contiene constantes
interface Constante {

    //Definiendo 3 constantes
    int MIN = 0;
    int MAX = 10;
    String MSJERROR = "LIMITE ERROR";
}
```



# Modificador de Omisión (default)

Permite a las interfaces definir métodos concretos que pueden ser usados por interfaces y/o clases hijas, cuando ellas no los definan.

## Ejemplo

```
interface Interfaz{  
    ...  
    default método(...){  
        ...  
    };  
    ...  
}
```



# Interfaces multiples I

Java no soporta herencia multiple (una clase sólo puede heredar de una única superclase) sin embargo esta restricción no está presente en las interfaces. Una clase puede implementar multiples interfaces separándolas con coma:

## Ejemplo

```
interface Interface1 {  
    public void miMetodo(); // método de interface  
}  
  
interface Interface2 {  
    public void miOtroMetodo(); // método de interface  
}
```



# Interfaces multiples II

## Ejemplo

```
class DemoClass implements Interface1, Interface2 {  
  
    public void miMetodo() {  
        System.out.println("Texto 1 ...");  
    }  
  
    public void miOtroMetodo() {  
        System.out.println("Texto 2 ...");  
    }  
  
}
```



# Herencia e interfaces

Una interfaz puede heredar otra mediante el uso de la palabra reservada `extends`. La sintaxis es la misma que para heredar clases.

## Ejemplo

```
//Una interface puede extender de otra
interface A{
    void metodo1();
    void metodo2();
}
```

```
//B ahora incluye metodo1() y metodo2() - y añade metodo3()
interface B extends A{
    void metodo3();
}
```





# Problemas varios I

## Problemas

- 1 Modele mediante el uso de clases y herencia la organización al interior de una universidad (profesores, alumnos, directivos, empleados, ...) Utilice la siguiente clase como base para las demás:

```
public class Persona {  
    private String nombre;  
    private String apellidos;  
    private int edad;  
  
    //Constructor  
    public Persona (String nombre, String apellidos,  
                    int edad) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad;  
    }  
}
```

# Problemas varios II

## Problemas

```
//Métodos
public String getNombre () { return nombre; }
public String getApellidos () { return apellidos; }
public int getEdad () { return edad; }
} //Cierre de la clase
```

- ② Usando el concepto de herencia, diseñe una superclase que sirva como base para los elementos que contiene una biblioteca. **Pista:** Piense en que características tienen en comun los libros, revistas, articulos y demas elementos presentes en cualquier biblioteca.



# Problemas varios III

## Problemas

- 3 Desarrolle la jerarquía de figuras geométricas de la diapositiva 7 en Java. Cambie el método pintar por toString e imprima en el información relevante de los objetos.
- 4 Modele una clase "Calculadora" que realice las cuatro operaciones básicas (suma, resta, multiplicación y división) Para ello haga uso del concepto de interfaces y diseñe dos interfaces, una que contenga las operaciones y otra que contenga constantes de utilidad para cálculos, como pi o e.

