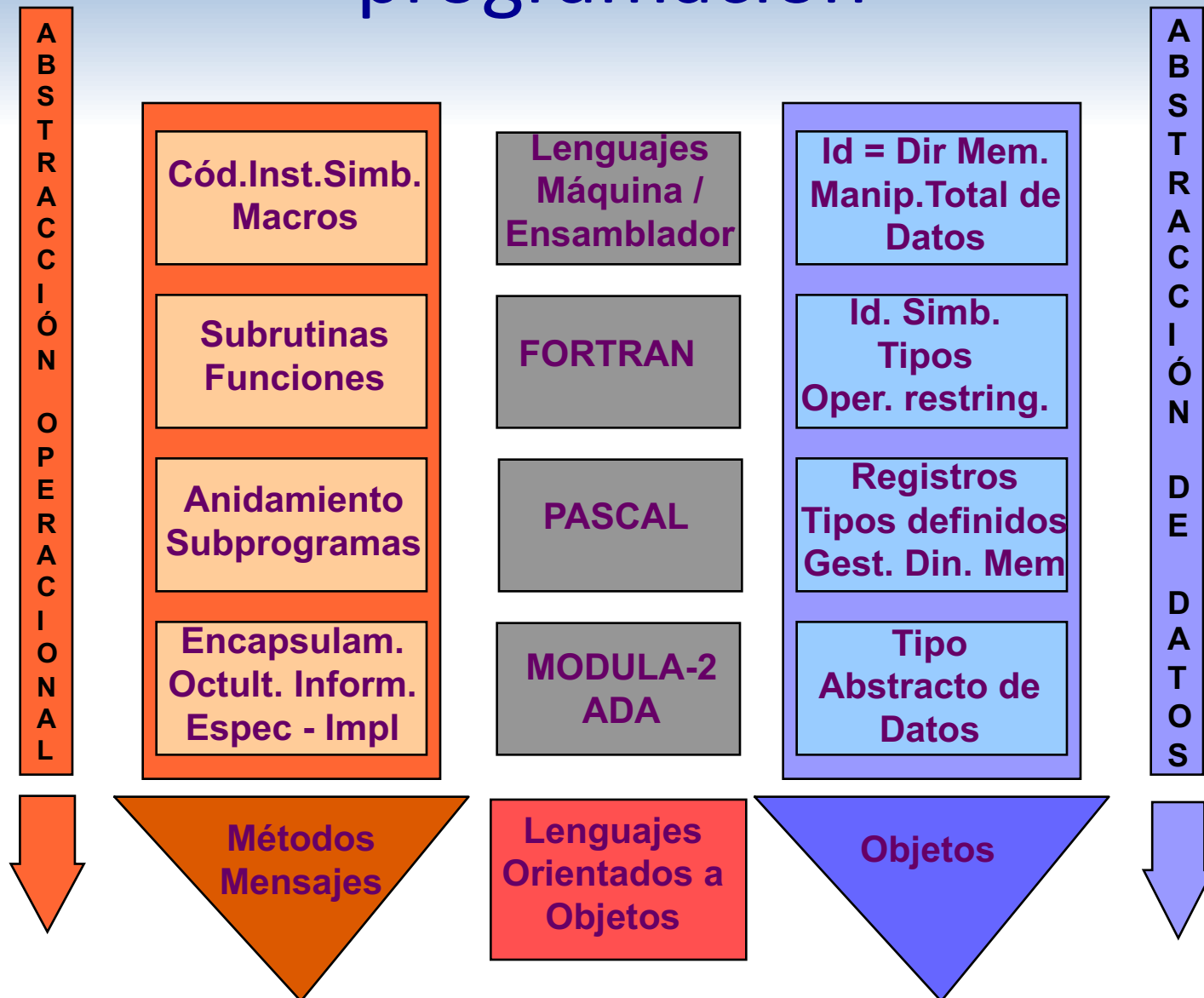


# Introducción a la Programación Orientada a Objetos

# Contenido

- Evolución de los lenguajes de programación
  - Evolución histórica
  - Abstracción procedimental y de datos
- Conceptos básicos de la P. O. O.
  - Clases y objetos
  - Métodos y mensajes
  - Polimorfismo y vinculación dinámica
  - Herencia / Composición
  - Clases abstractas

# Evolución de los lenguajes de programación



# ¿Qué es la POO?

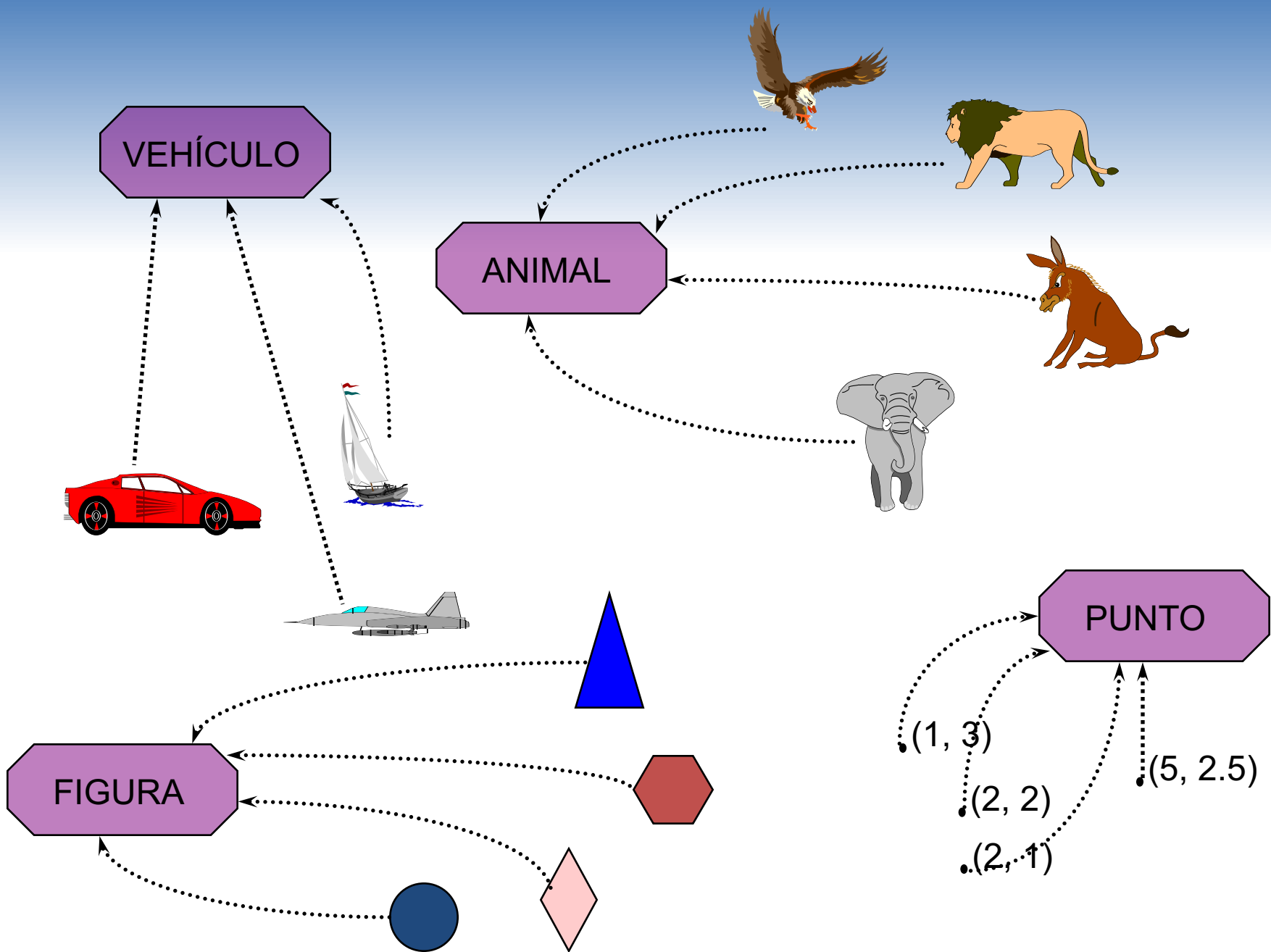
- Estilo de programación que trata de representar un modelo de la realidad basado en los datos a manipular.
  - Las abstracciones de datos se modelan con **objetos**.
  - Diseño enfocado al cliente. Los objetos se refieren a datos que el cliente entiende porque forman parte de la especificación del problema.

# Conceptos básicos de la P.O.O.

- Clases y objetos
- Métodos y mensajes
- Herencia
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Clases y Objetos

- **CLASE = MÓDULO + TIPO**
  - Criterio de estructuración del código
  - Estado + Comportamiento
  - Entidad estática (en general)
- **OBJETO = Instancia de una CLASE**
  - Objeto (Clase) = Valor (Tipo)
  - Entidad dinámica
  - Cada objeto tiene su propio estado
  - Objetos de una clase comparten su comportamiento



# Ejemplo: Urna

- Queremos manipular urnas capaces de contener bolas blancas y negras.
  - Utilizando la abstracción, una urna se puede representar por un objeto que contiene dos enteros (que llamaremos **su estado**)
    - nBlancas: int.    Número de bolas blancas
    - nNegras: int.    Número de bolas negras

Urna (nBlancas:34, nNegras: 16)

Urna (nBlancas:21, nNegras: 8)



# Comportamiento de Urna

- Una vez fijado el estado de la urna, el cliente especifica cómo operar con la urna.
- Por ejemplo
  - Queremos saber el total de bolas que tiene la urna.
  - Queremos poder introducir una bola del color que queramos (blanca o negra).
  - Queremos sacar aleatoriamente una bola de la urna para saber su color.
  - Queremos saber si la urna está vacía.
- Estas operaciones definen lo que llamaremos **el comportamiento** de la urna.

# Ejemplo: Jarra

- Queremos manipular jarras que tienen una capacidad y un contenido (siempre en litros):
  - capacidad: int. Lo que cabe en la jarra
  - contenido: int. Lo que actualmente tiene la jarra
- Una jarra la representamos como un objeto con dos enteros, uno para la capacidad y otro para el contenido (**su estado**)

Jarra(capacidad:7, contenido:3)

Jarra(capacidad:5, contenido:0)

# Comportamiento de Jarra

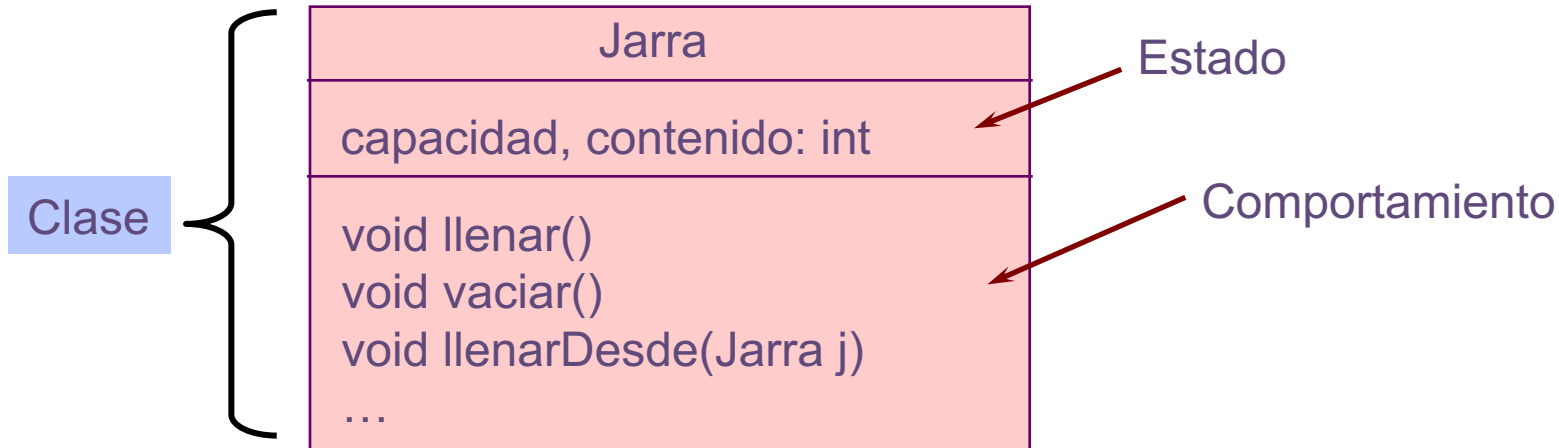
- ¿Cómo vamos a operar con la jarra?
  - Queremos poder llenar la jarra desde una fuente hasta completarla.
  - Queremos poder volcar la jarra en un sumidero hasta vaciarla.
  - Queremos poder volcar una jarra sobre otra hasta que la segunda se llene o la primera se vacíe.
  - Queremos saber si la jarra está vacía.
- Estas operaciones definen el **comportamiento** de las jarras.

# Urnas y Jarras. Conceptos

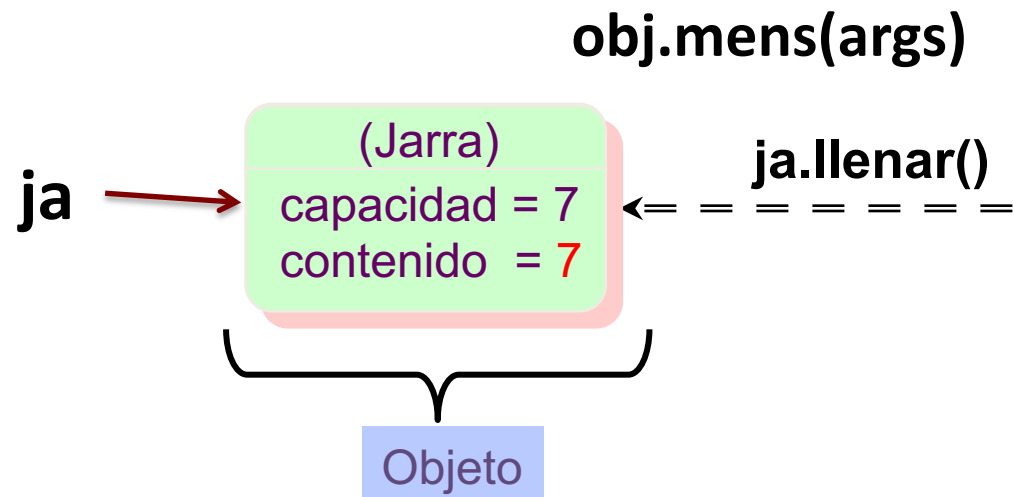
- Podemos crear muchas urnas y muchas jarras.
- Cada urna y cada jarra tiene **su propio estado**.
- Una urna **se diferencia** de otra en su estado  
Urna(3,5)   Urna(8,2)   Urna(32,17)   Urna(9,0)
- Una jarra **se diferencia** de otra en su estado.  
Jarra(7, 5)   Jarra(8, 8)   Jarra(9,0)   Jarra(2,1)
- Todas las urnas tienen **el mismo comportamiento**
- Todas las jarras tienen **el mismo comportamiento**.
- Cuando se actúa sobre una urna o jarra, responden con arreglo a su estado, y éste puede verse modificado.
- Es posible crear una abstracción mayor:
  - Que defina la forma del estado.
  - Que defina el comportamiento.

# Representación de una clase. Estado, Métodos y mensajes

- **Métodos:** definen el comportamiento de una clase



- Invocación de **métodos**: Paso de **mensajes**



# Paso de mensajes

- Los **mensajes** que se envían a un determinado objeto **deben “corresponderse”** con los **métodos** que la clase tiene definidos.
- Esta correspondencia se debe reflejar en la **signatura** del método: **nombre**, **argumentos** y sus **tipos**.
- En los lenguajes orientados a objetos con comprobación de tipos, la emisión de un mensaje a un objeto que no tiene definido el método correspondiente se detecta en tiempo de compilación.
- Si el lenguaje no realiza comprobación de tipos, los errores en tiempo de ejecución pueden ser inesperados.

# Paso de Mensajes

- Cuando actuamos sobre un objeto a través de su comportamiento, podemos:
  - Consultar el estado del objeto
  - Modificar el estado del objeto
- Ejemplo: si tenemos la siguiente urna `Urna(3,5)`
  - y le preguntamos por el total de bolas nos devolverá 8.
  - y si le introducimos una bola negra, su estado cambiará a `Urna(3,6)`
- Ejemplo: si tenemos las jarras `Jarra(7, 5)` y `Jarra(5,3)`
  - y le preguntamos si están vacías, ambas nos responderán con `false`.
  - y si volcamos la segunda sobre la primera, el estado de ambas jarras se modifica pasando a ser `Jarra(7, 7)` `Jarra(5,1)`
- La manera de actuar sobre un objeto a través de su comportamiento es por medio de **paso de mensajes**.
  - A una urna se le envía un mensaje pidiéndole el número total de bolas.
  - A una jarra se le envía un mensaje pidiéndole que se rellene con el contenido de otra jarra.

# Clases

- Estructuras que encapsulan *datos y métodos*

“Punto.java”

```
public class Punto {  
    private double x, y;  
    public Punto() { x = y = 0; }  
    public Punto(double a, double b) { x = a; y = b; }  
    public double abscisa() {return x;}  
    public double ordenada() {return y;}  
    public void abscisa(double a) { x = a; }  
    public void ordenada(double b) { y = b; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public double distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x - pto.x, 2) +  
            Math.pow(y - pto.y, 2));  
    }  
}
```

**VARIABLES DE ESTADO**

**CONSTRUCTORES**

**MÉTODOS**

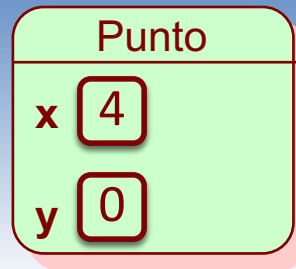


```
public class Punto {  
    private double x, y;
```

```
    public Punto(double a, double b) {  
        x = a; y = b;  
    }  
    ...
```

```
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }
```

```
    public double distancia(Punto p) { ... }  
};
```



pto

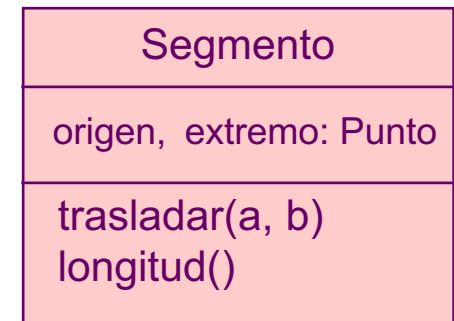
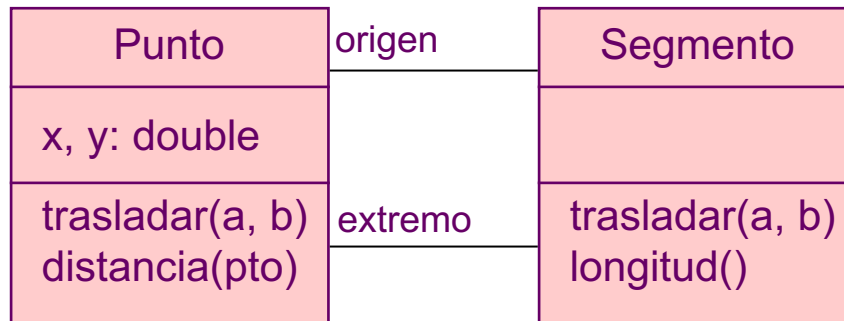
*trasladar(3, -1)*

```
Punto pto = new Punto(1, 1);
```

```
pto.trasladar(3, -1);
```

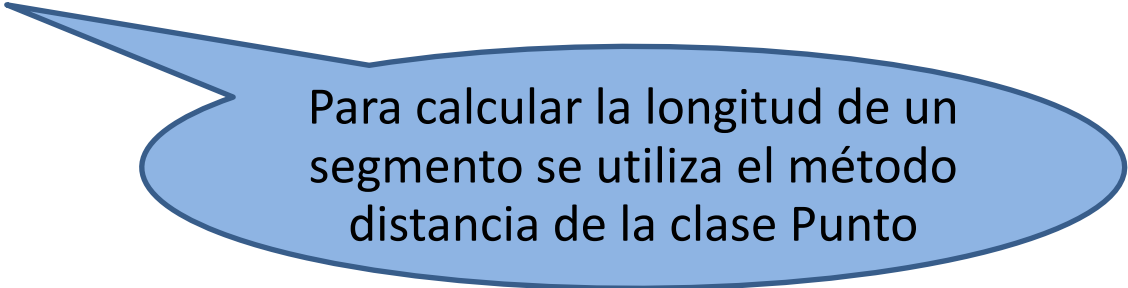
# Composición

- Mecanismo que permite la creación de nuevos objetos a partir de otros ya implementados.
- Responde a una relación de tipo “*está compuesto por*”, “*tiene*” o “*usa*”.
- Así, por ejemplo, un segmento **está compuesto por** dos puntos (origen y extremo)
  - También podemos decir que los puntos origen y extremo “*forman parte del*” segmento, o que el segmento “*tiene*” dos puntos.



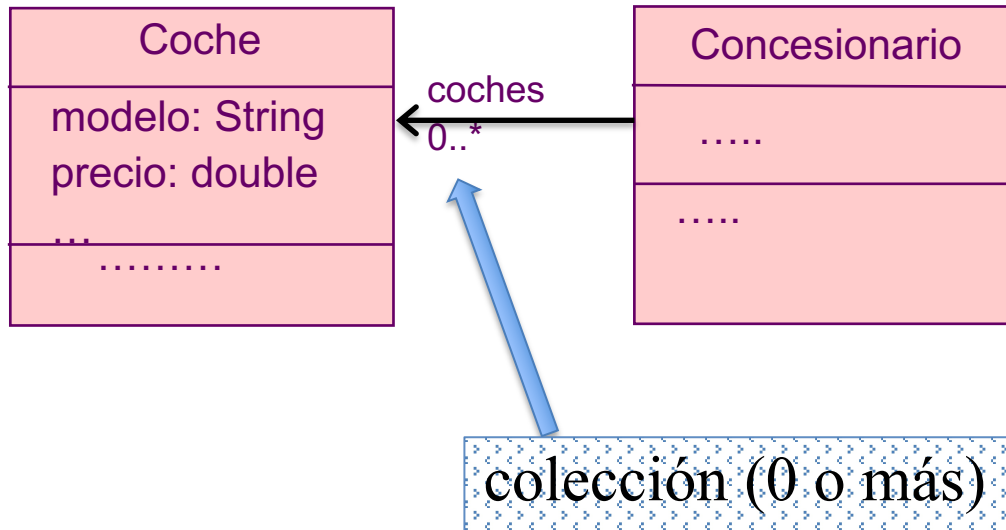
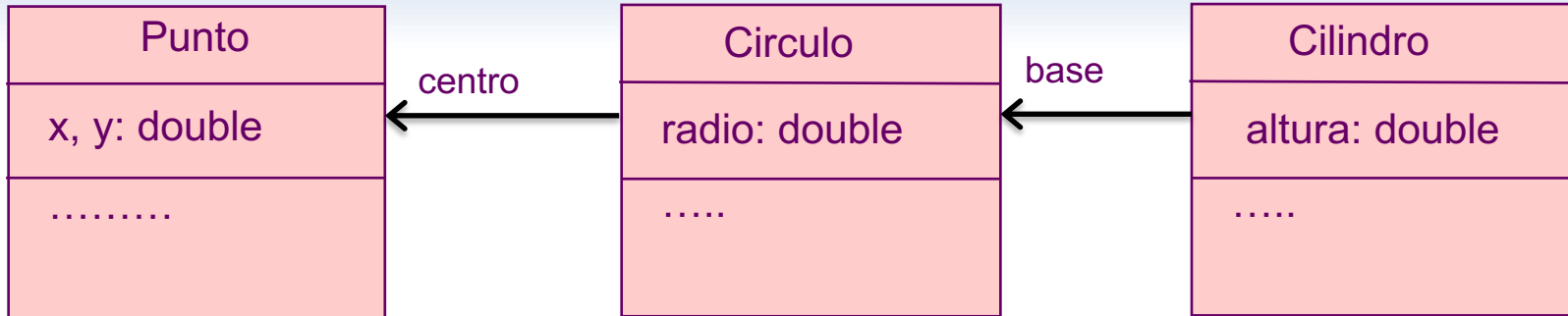
# Composición

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
  
    ... // Otros métodos  
  
    public double longitud() {  
        return origen.distancia(extremo);  
    }  
}
```



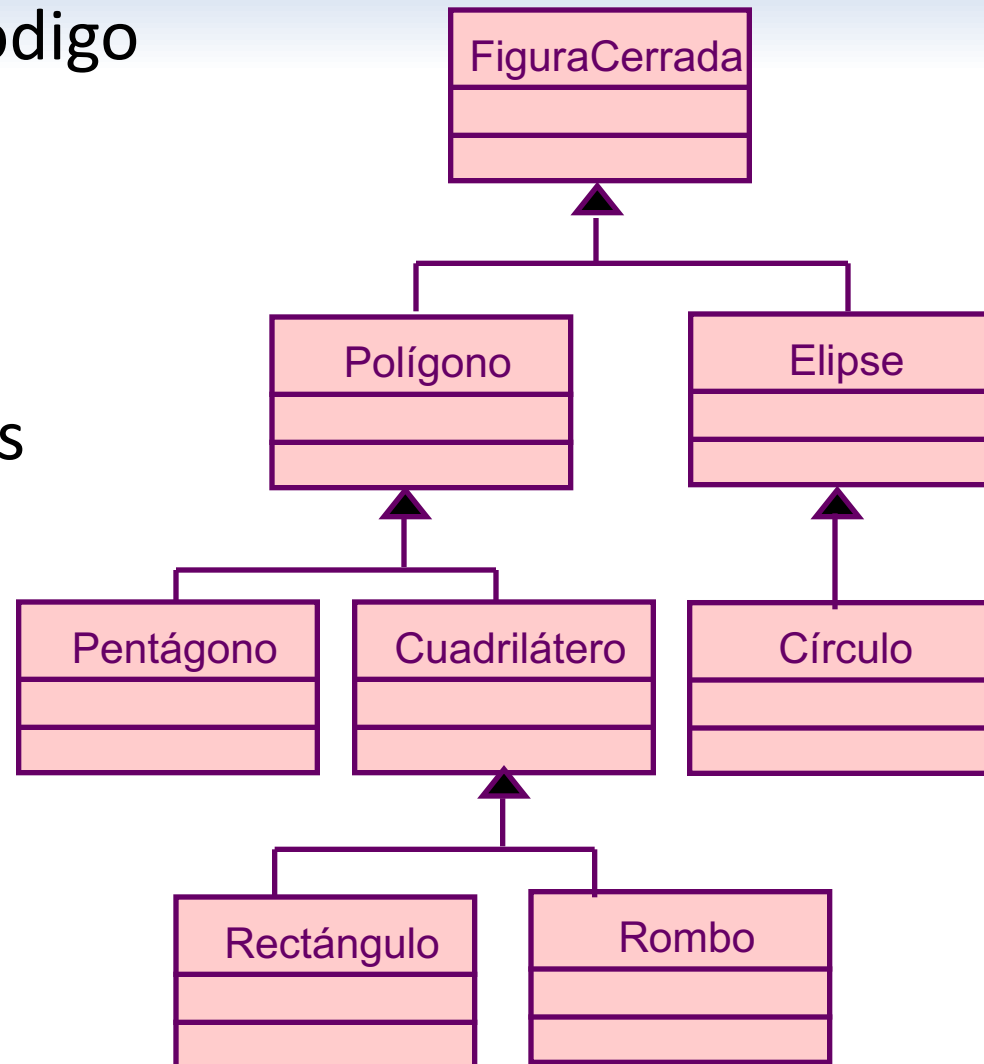
Para calcular la longitud de un segmento se utiliza el método distancia de la clase Punto

# Otros ejemplos de composición



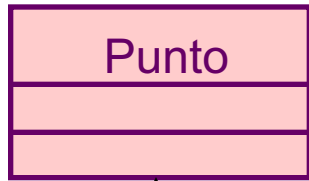
# Herencia

- Otra forma de **reutilizar** código
- Algo más que:
  - incluir ficheros, o
  - importar módulos
- Permite clasificar las clases en una jerarquía
- Responde a la relación “es un”



# Herencia

Madres / Ascendientes /  
Superclase



Hijas / Descendientes /  
Subclase

- Una subclase **dispone** de los **atributos** y **métodos** de la superclase, y **puede añadir** otros nuevos.
- La subclase puede **modificar** el comportamiento heredado (por ejemplo, redefiniendo algún método heredado) .
- La herencia es transitiva.
- Los objetos de una clase que hereda de otra **pueden verse** como objetos de esta última.

```
public class Partícula extends Punto {  
    protected double masa;  
    final static double G = 6.67e-11;
```

```
    public Partícula(double m) {  
        this(0,0,m);  
    }
```

Se refiere a  
Partícula(double, double, double)

```
    public Partícula(double a, double b, double m) {  
        super(a, b);  
        masa = m;  
    }
```

Se refiere a  
Punto(double, double)

```
    public void masa(double m) { masa = m; }
```

```
    public double masa() { return masa; }
```

```
    public double atracción(Partícula part) {
```

```
        double d = this.distancia(part);
```

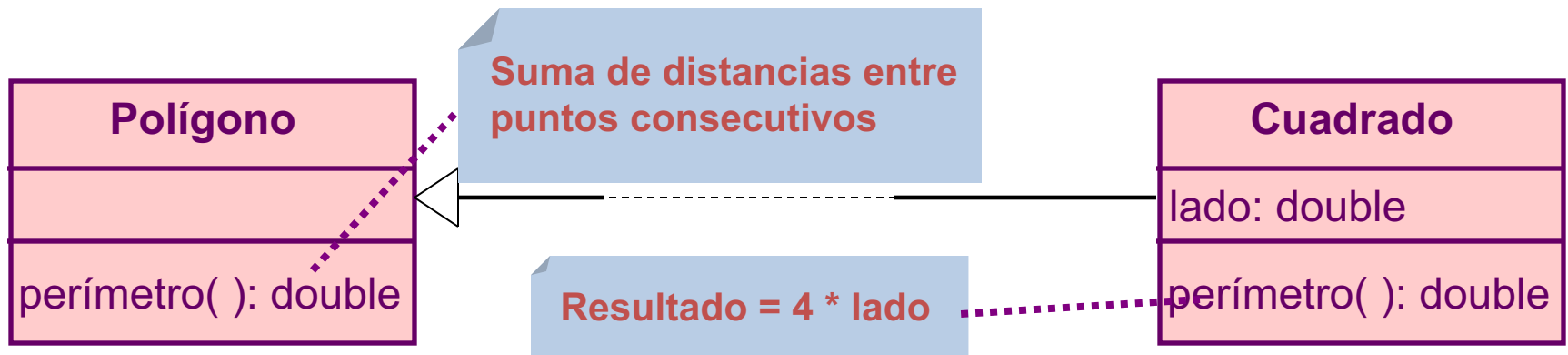
```
        return G * masa * part.masa() / (d * d);
```

```
    }  
}
```

Heredada de  
Punto

# Redefinición del comportamiento

- Es muy corriente la redefinición de un método en la subclase.

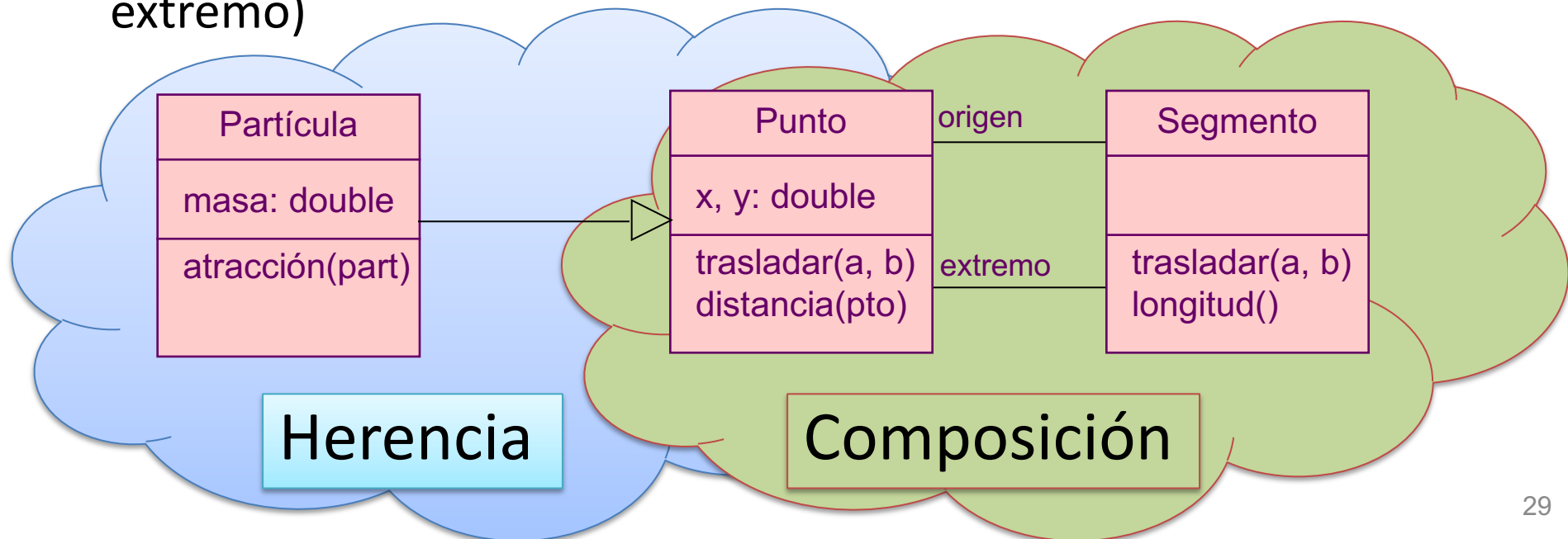


- La redefinición puede impedirse mediante el uso del calificador **final**.



# Herencia vs. composición

- Mientras que la herencia establece una relación de tipo “*es-un*”, la composición responde a una relación de tipo “*tiene*” o “*está compuesto por*”.
- Así, por ejemplo, una partícula **es un** punto (con masa), mientras que un segmento **está compuesto** por dos puntos (origen y extremo)



# Polimorfismo sobre los datos

- Un lenguaje tiene **capacidad polimórfica** sobre los datos cuando
  - una variable declarada de un tipo (o clase) –*tipo estático*– determinado puede hacer referencia en tiempo de ejecución a valores (objetos) de tipo (clase) distinto –*tipo dinámico*–.
- La capacidad polimórfica de un lenguaje no suele ser ilimitada, y en los LOO está habitualmente restringida por la relación de herencia:
  - El *tipo dinámico* debe ser **descendiente** del *tipo estático*.

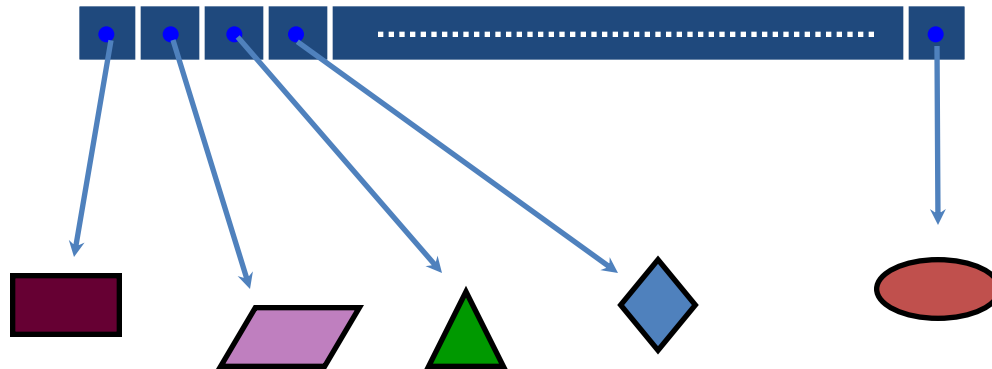
```
Punto pto = new Partícula(3, 5, 22);
```

Tipo estático de **pto**  
(corresponde a la  
declaración)

Tipo dinámico de **pto**  
en el momento de esta  
asignación

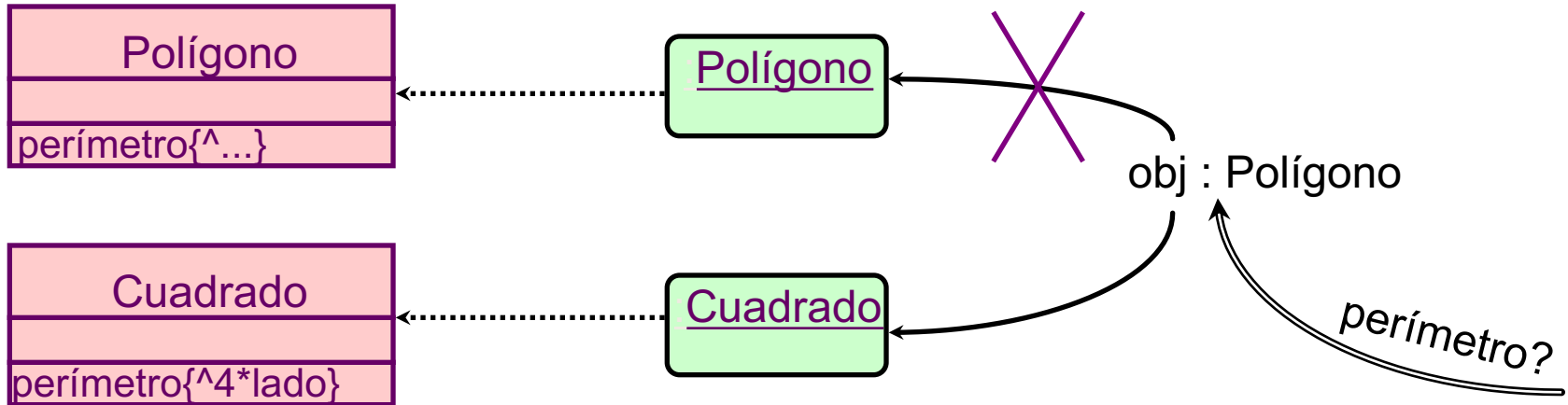
# Polimorfismo sobre los datos

- Una variable puede referirse a objetos de clases distintas de la que se ha declarado. Esto afecta a:
  - asignaciones explícitas entre objetos,
  - paso de parámetros,
  - devolución del resultado en una función.
- La restricción dada por la herencia permite construir estructuras con elementos de naturaleza distinta, pero con un comportamiento común:



# Vinculación dinámica

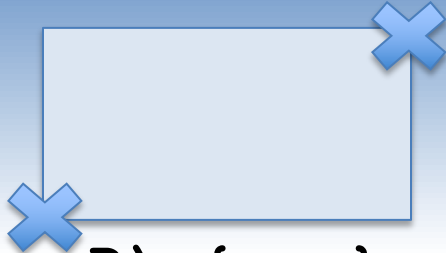
- La **vinculación dinámica** resulta el complemento indispensable del polimorfismo sobre los datos, y consiste en que:
  - La **invocación del método** que ha de resolver un mensaje **se retrasa al tiempo de ejecución**, y se hace depender del **tipo dinámico** del objeto



- El compilador admitirá la expresión **obj.perímetro()** ;  
si el **tipo estático** de obj (es decir la clase Polígono) acepta el mensaje **perímetro()** , aunque para resolver utilice **vinculación dinámica** (es decir, el método **perímetro()** de la clase Cuadrado)

```
public class PuntoAcotado extends Punto {
    private Punto esquinaI, esquinaD;

    public PuntoAcotado() { ... }
    public PuntoAcotado(Punto eI, Punto eD) { ... }
    public double ancho() { ... }
    public double alto() { ... }
    public void trasladar(double a, double b) {
        super.trasladar(a, b);
        if (abscisa() < esquinaI.abscisa())
            abscisa(esquinaI.abscisa())
        if (abscisa() > esquinaD.abscisa())
            abscisa(esquinaD.abscisa())
        if (ordenada() < esquinaI.ordenada())
            ordenada(esquinaI.ordenada())
        if (ordenada() > esquinaD.ordenada())
            ordenada(esquinaD.ordenada())
    }
}
```



```
public class Punto {  
    private double x, y;  
    public Punto() { ... }
```

```
...
```

```
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }
```

```
    public double distancia(Punto p) { ... }
```

```
}
```

```
Punto eI = new Punto(0, 0);
```

```
Punto eD = new Punto(5, 5);
```

```
Punto pto;
```

```
PuntoAcotado pac = new PuntoAcotado(eI, eD);
```

```
pto = pac;
```

```
pto.trasladar(2, 7);
```

PuntoAcotado

x = 2

y = 5

pac

pto

trasladar(2, 7)

# Clases abstractas

- Clases de la que **no se pueden** crear instancias
  - Pueden declarar métodos sin implementar
    - Métodos abstractos
  - Las subclases están obligadas a implementarlas
- Se pueden declarar variables cuyo tipo estático sea una clase abstracta que puedan referirse a objetos de diversas clases descendientes

## CLASE ABSTRACTA

```
abstract public class Polígono {  
    private Punto vértices[];  
    public void trasladar(double a, double b){  
        for (int i = 0; i < vértices.length; i++)  
            vértices[i].trasladar(a, b);  
    }  
    public double perímetro() {  
        double per = 0;  
        for (int i = 1; i < vértices.length; i++)  
            per = per + vértices[i - 1].distancia(vértices[i]);  
        return per  
            + vértices[0].distancia(vértices[vértices.length]);  
    }  
    abstract public double área();  
}
```

## MÉTODO ABSTRACTO

~~Polígono pol = new Polígono();~~



# Clases abstractas

- Las clases abstractas definen un protocolo común en una jerarquía de clases.
- Obligan a sus subclasses a implementar los métodos que se declararon como abstractos.
  - De lo contrario, esas subclasses se siguen considerando abstractas.
- En Java, además de clases abstractas se pueden definir **interfaces** (que se pueden considerar clases “completamente” abstractas):
  - Las interfaces constituyen una declaración del comportamiento de un grupo de objetos, sin entrar en cómo se implementa.
- Cuando se desarrolla un sistema orientado a objetos es importante distinguir las **interfaces** (nivel de abstracción más alto) que **declaran** el comportamiento de los objetos, de las **clases**, que proporcionan **implementaciones** de ese comportamiento:
  - Implementaciones parciales se corresponden con **clases abstractas**.