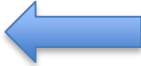


# Tema 1

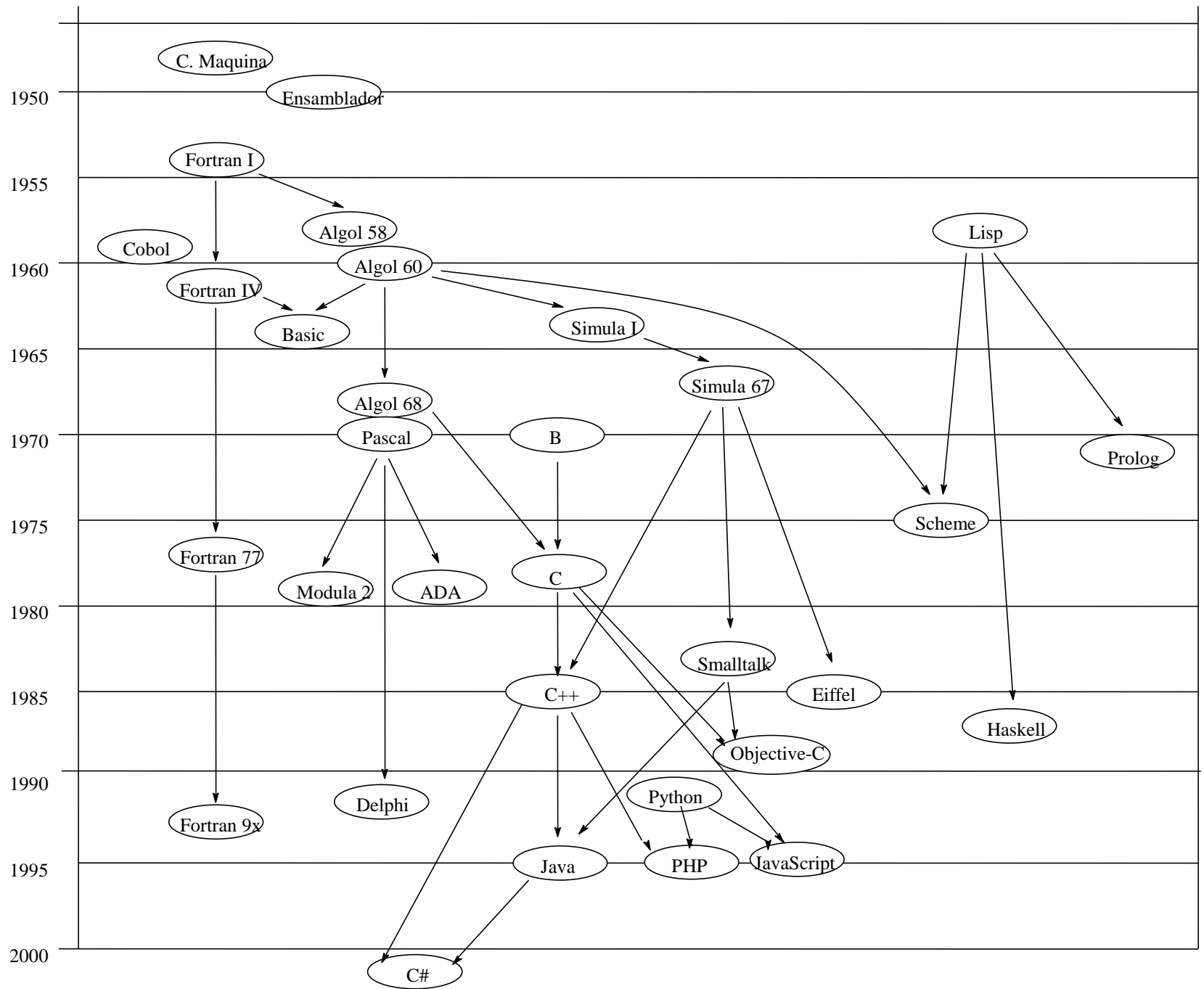
## Introducción a la Programación Orientada a Objetos

# Contenido

- Evolución de los Lenguajes de Programación
  - Evolución histórica. Paradigmas de Programación 
  - Abstracción Procedimental y de Datos
- Conceptos fundamentales de la Programación Orientada a Objetos
  - Clases y Objetos
  - Métodos y Mensajes
  - Composición
  - Herencia y Redefinición del Comportamiento
  - Polimorfismo y Vinculación Dinámica
  - Clases Abstractas e Interfaces (Tema 2)

# Evolución de los Lenguajes de Programación

Evolución histórica. Paradigmas de  
Programación  
(repaso tema 1 del 1er cuatrimestre)



# Lenguajes de Programación

- Es posible clasificar los lenguajes de programación siguiendo diferentes criterios.
- Se considerarán tres:
  - Nivel de abstracción
  - Finalidad del lenguaje
  - Características del lenguaje

# Lenguajes de Programación

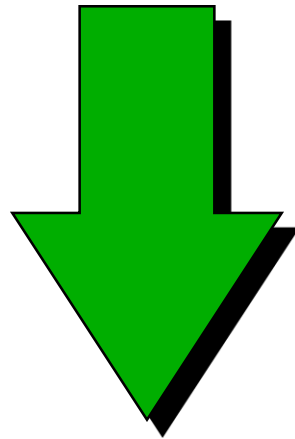
- Según el Nivel de abstracción
  - Lenguajes de bajo nivel
    - Cercanos a la máquina
    - Bajo nivel de abstracción
      - Lenguaje máquina
      - Ensamblador
  - Lenguajes de alto nivel
    - Cercanos al problema
    - Alto nivel de abstracción

# Lenguajes de Programación

- Según la Finalidad del lenguaje:
  - Científicos: ALGOL, FORTRAN, ...
  - de Ingeniería: ADA, DYNAMO, ...
  - de Gestión: COBOL, dBASE, ...
  - de Inteligencia Artificial: LISP, PROLOG, ...
  - de Aplicaciones Web: PHP, JavaScript, ...
  - Multipropósito: PASCAL, MODULA-2, C++, Java, ..
  - . . .

# Lenguajes de Programación

Clasificación de los lenguajes de programación según sus Características



Paradigmas de Programación

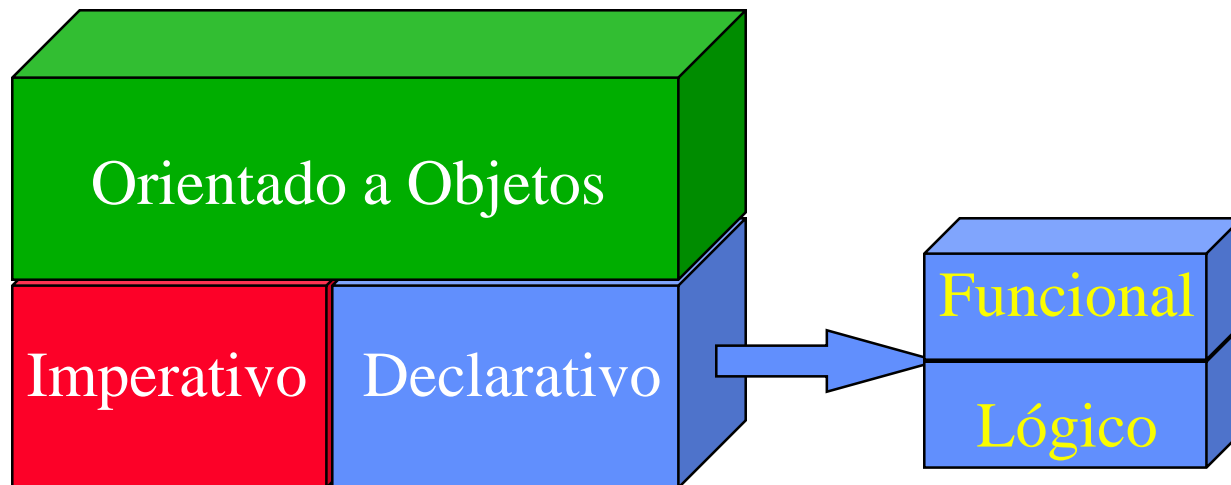


# Lenguajes de Programación

- Un paradigma de programación es un modelo que engloba a ciertos lenguajes que comparten:
  - Elementos estructurales:
    - ¿con qué se confeccionan los programas?
  - Elementos metodológicos:
    - ¿cómo se confecciona un programa?

# Lenguajes de Programación

- Consideramos los siguientes paradigmas:



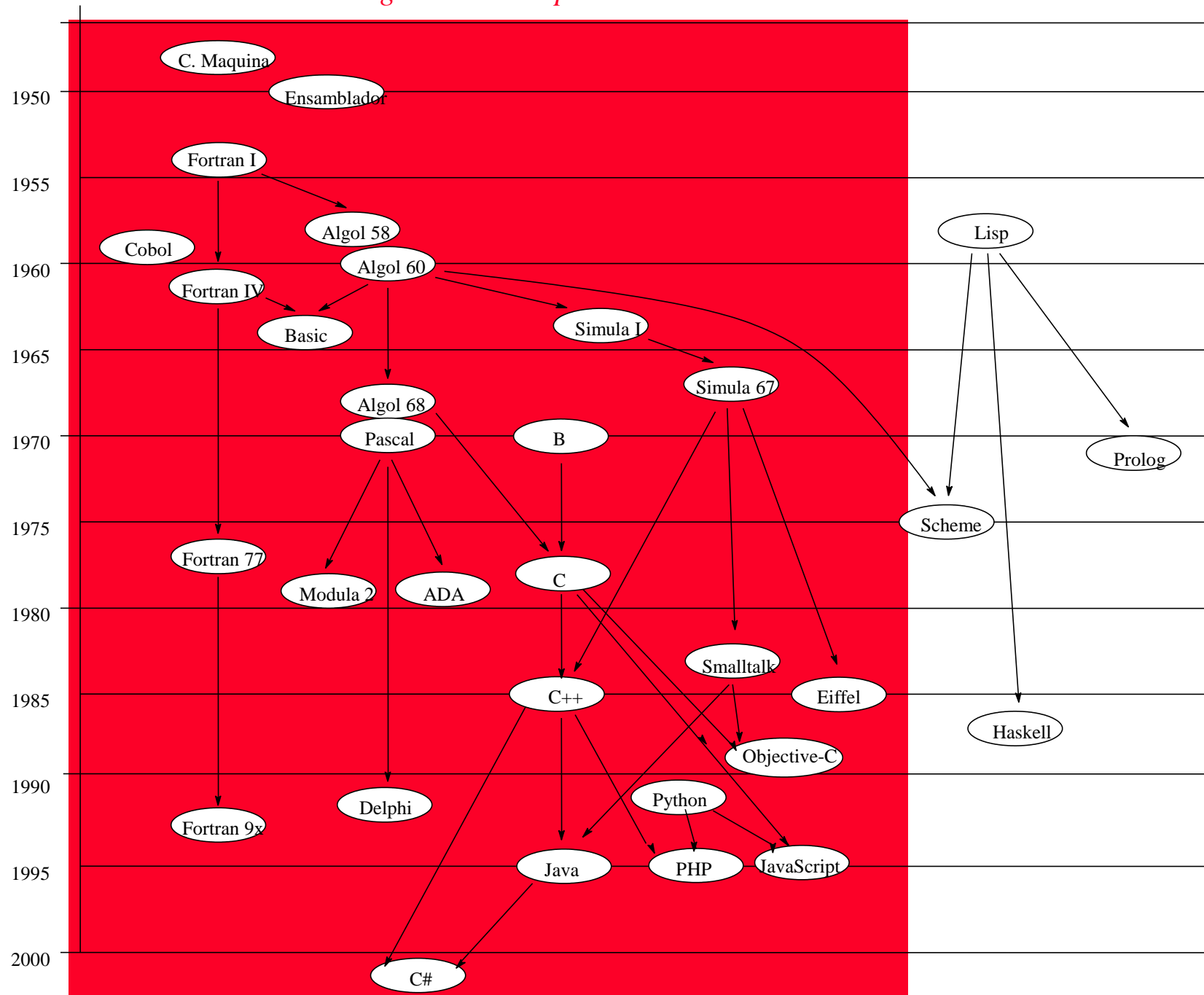
- Los paradigmas no son disjuntos.

# Lenguajes de Programación

## Programación Imperativa

- Es la más antigua → máquina de von Neumann
- Un programa es una secuencia de acciones que se realizan en orden.
- Existen herramientas para modificar el orden de ejecución de las acciones.

# Programación Imperativa



# Lenguajes de Programación

## Programación Declarativa

- **Programación Funcional**: definición de una serie de funciones.
- **Programación Lógica**: definición de hechos y relaciones lógicas entre éstos.
- No se indica el orden en el que se computa una función o se deriva un nuevo hecho.

# Lenguajes de Programación

Antecesor(X,Y) :- Padre(X,Y).

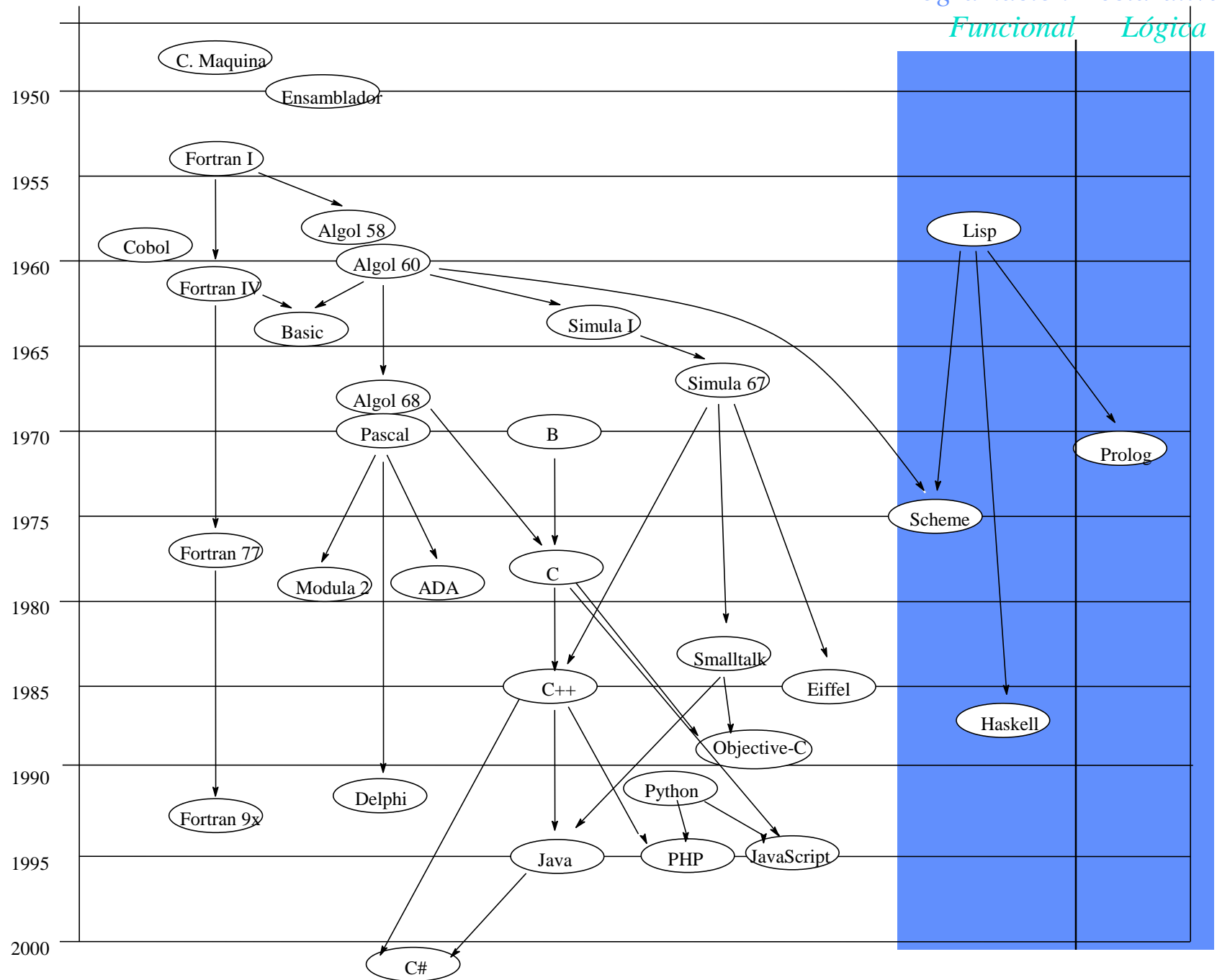
Antecesor(X,Y) :- Padre(Z,Y), Antecesor(X,Z).

Padre(---,---).

Padre(---,---).

....

? Antecesor(Pepe,Antonio).



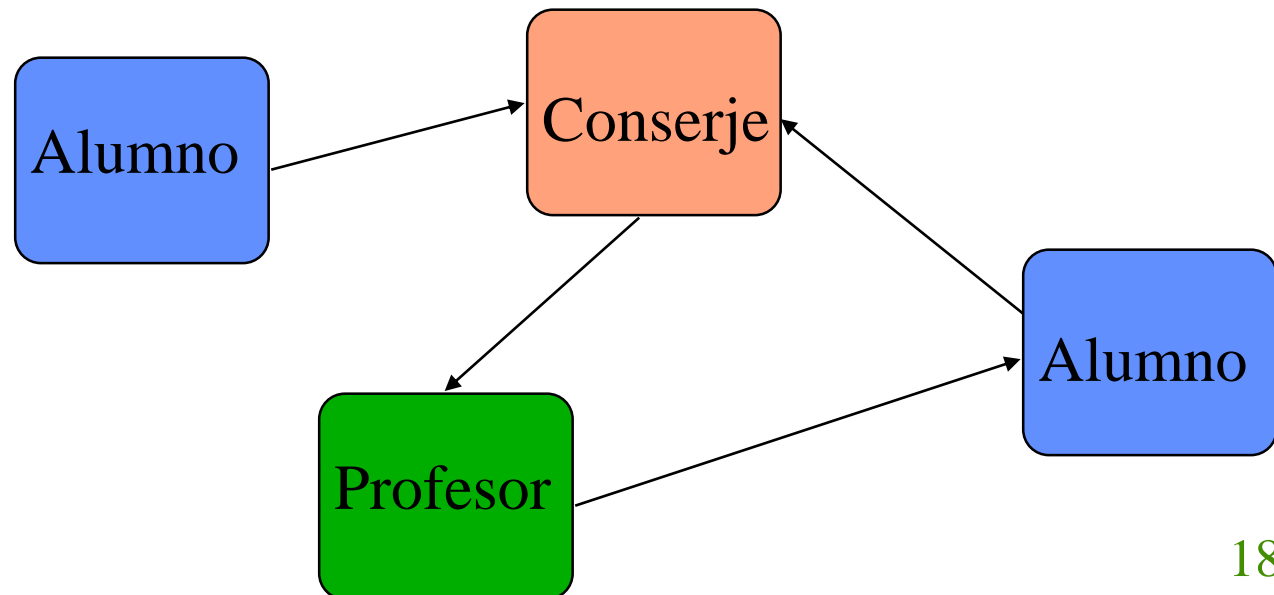
## Programación Orientada a Objetos

- El paradigma de la POO introduce un 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.



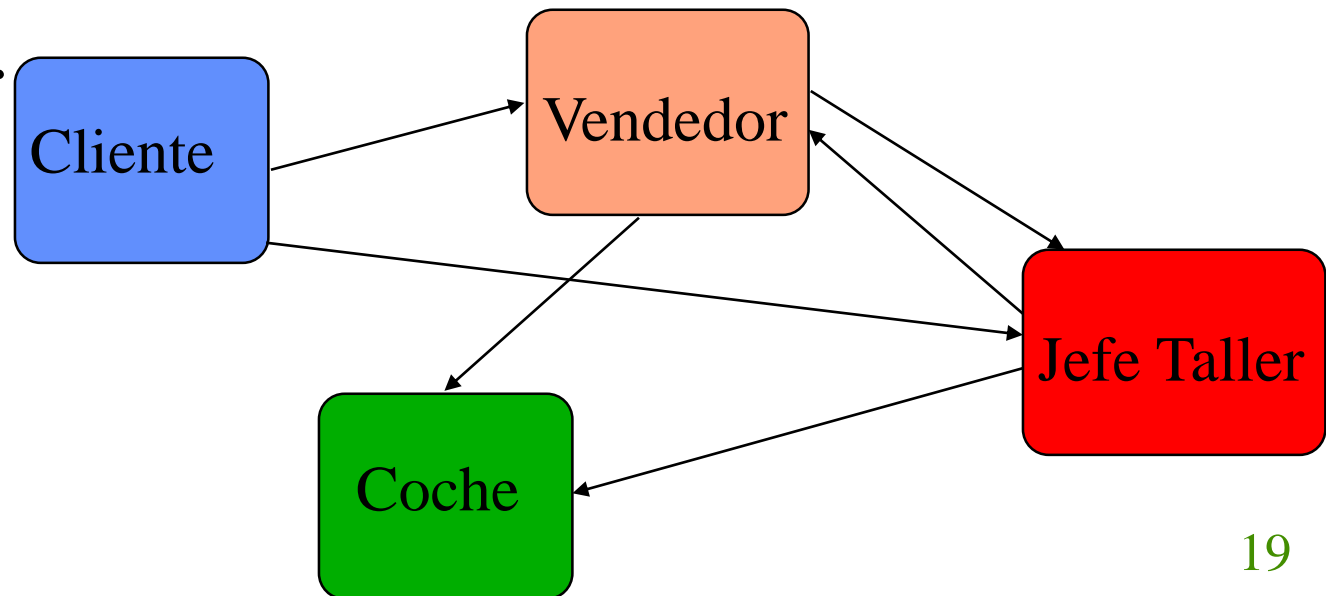
## Programación Orientada a Objetos

- El paradigma de la POO introduce un estilo de programación que trata de representar un modelo de la realidad basado en los datos a manipular.



## Programación Orientada a Objetos

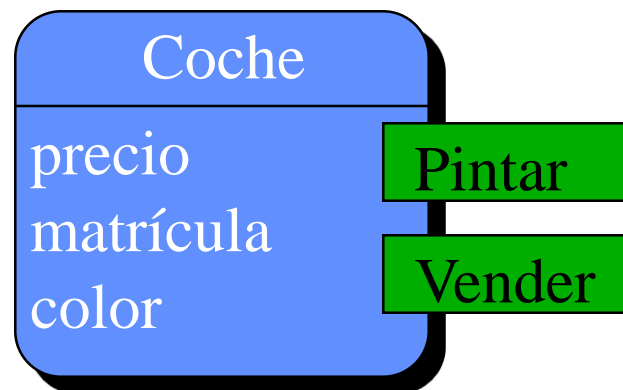
- El paradigma de la POO introduce un estilo de programación que trata de representar un modelo de la realidad basado en los datos a manipular.

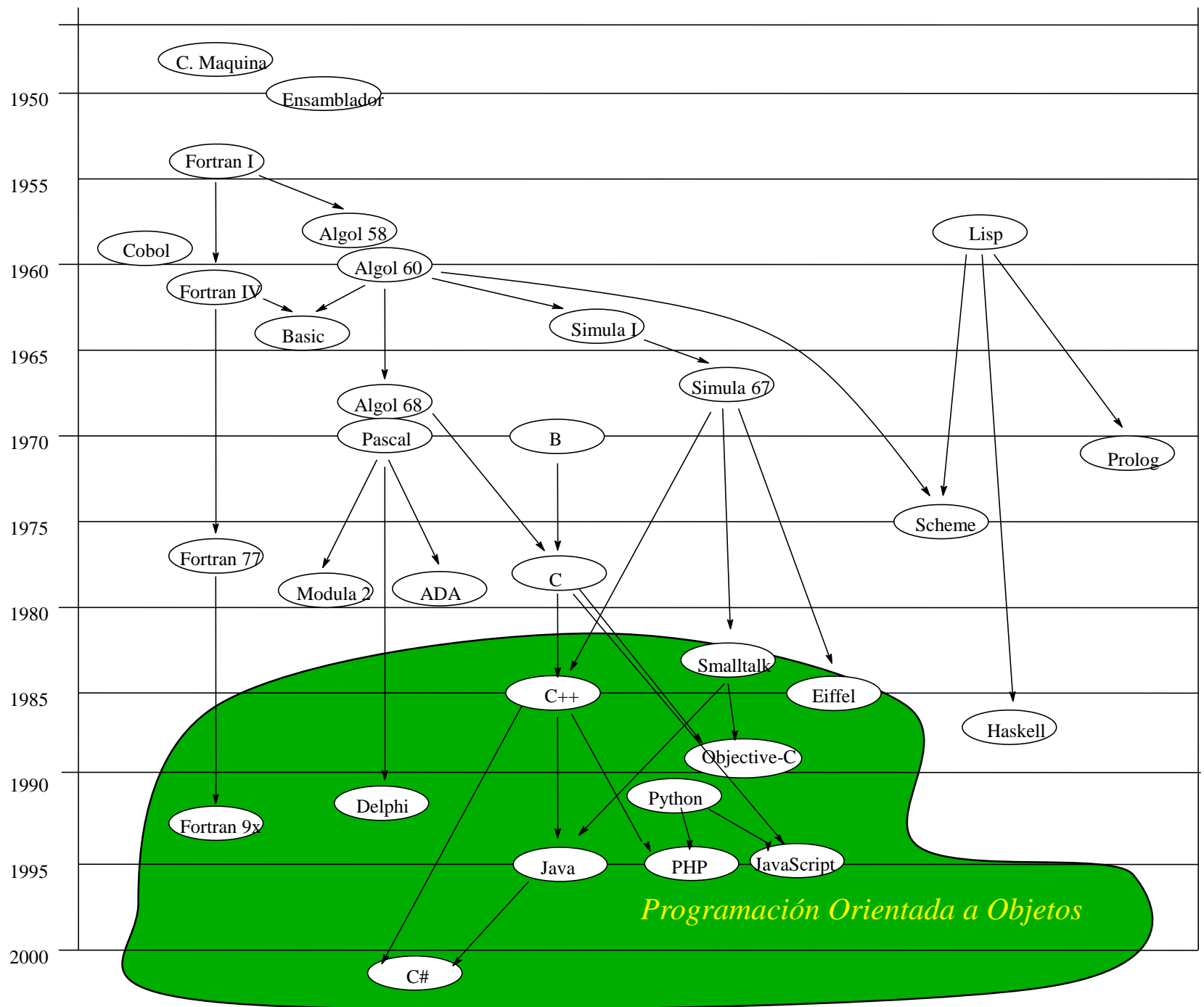


# Lenguajes de Programación

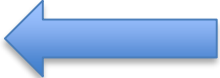
## Programación Orientada a Objetos

- Cada objeto es una entidad que agrupa una cierta información (**estado**) y un conjunto de mecanismos para manipularla (**métodos**).



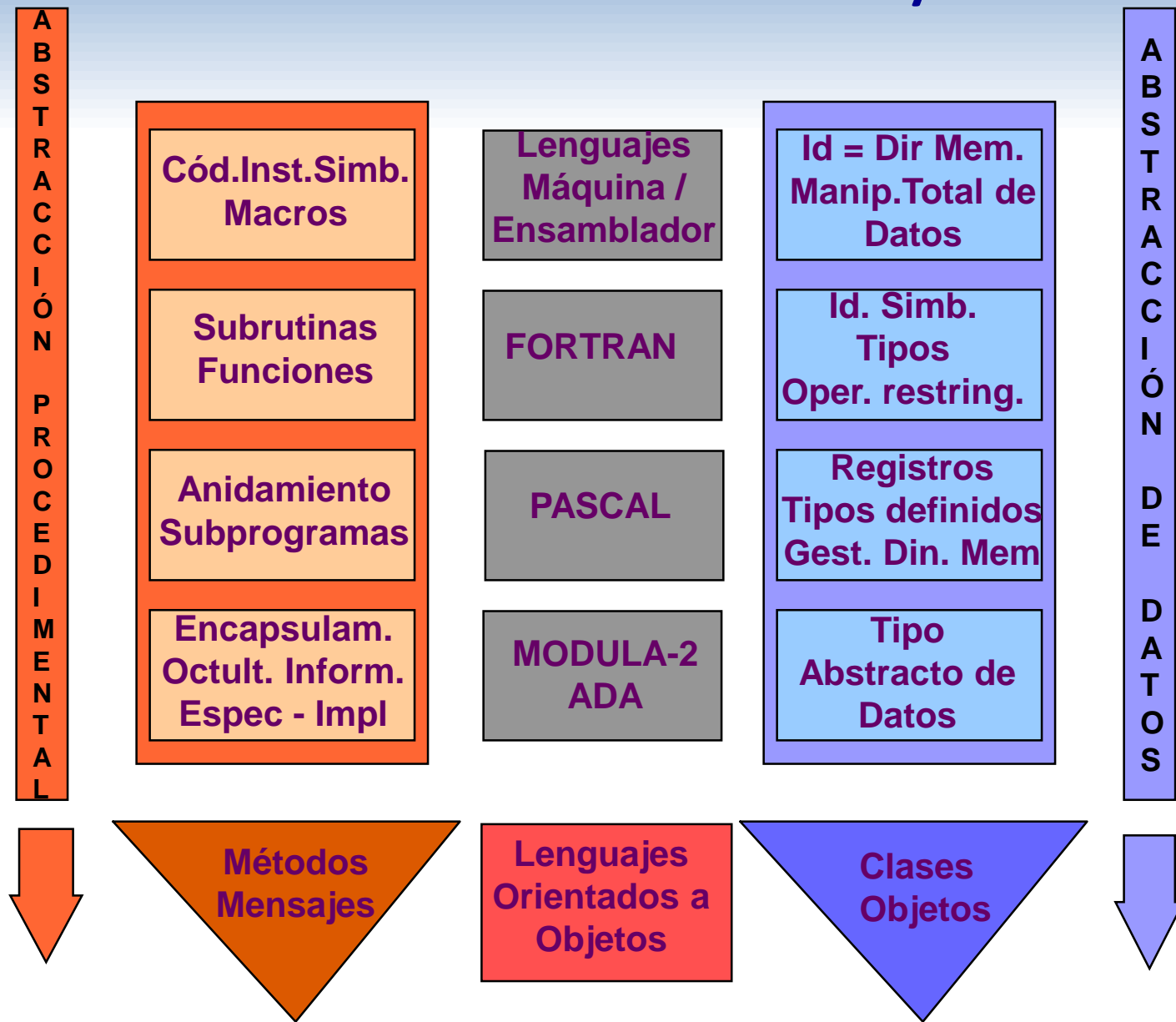


# Contenido

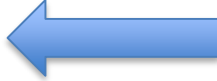
- Evolución de los Lenguajes de Programación
  - Evolución histórica. Paradigmas de Programación
  - Abstracción Procedimental y de Datos 
- Conceptos fundamentales de la Programación Orientada a Objetos
  - Clases y Objetos
  - Métodos y Mensajes
  - Composición
  - Herencia y Redefinición del Comportamiento
  - Polimorfismo y Vinculación Dinámica
  - Clases Abstractas e Interfaces (Tema 2)

# Evolución de los Lenguajes de Programación

## Abstracción Procedimental y de Datos



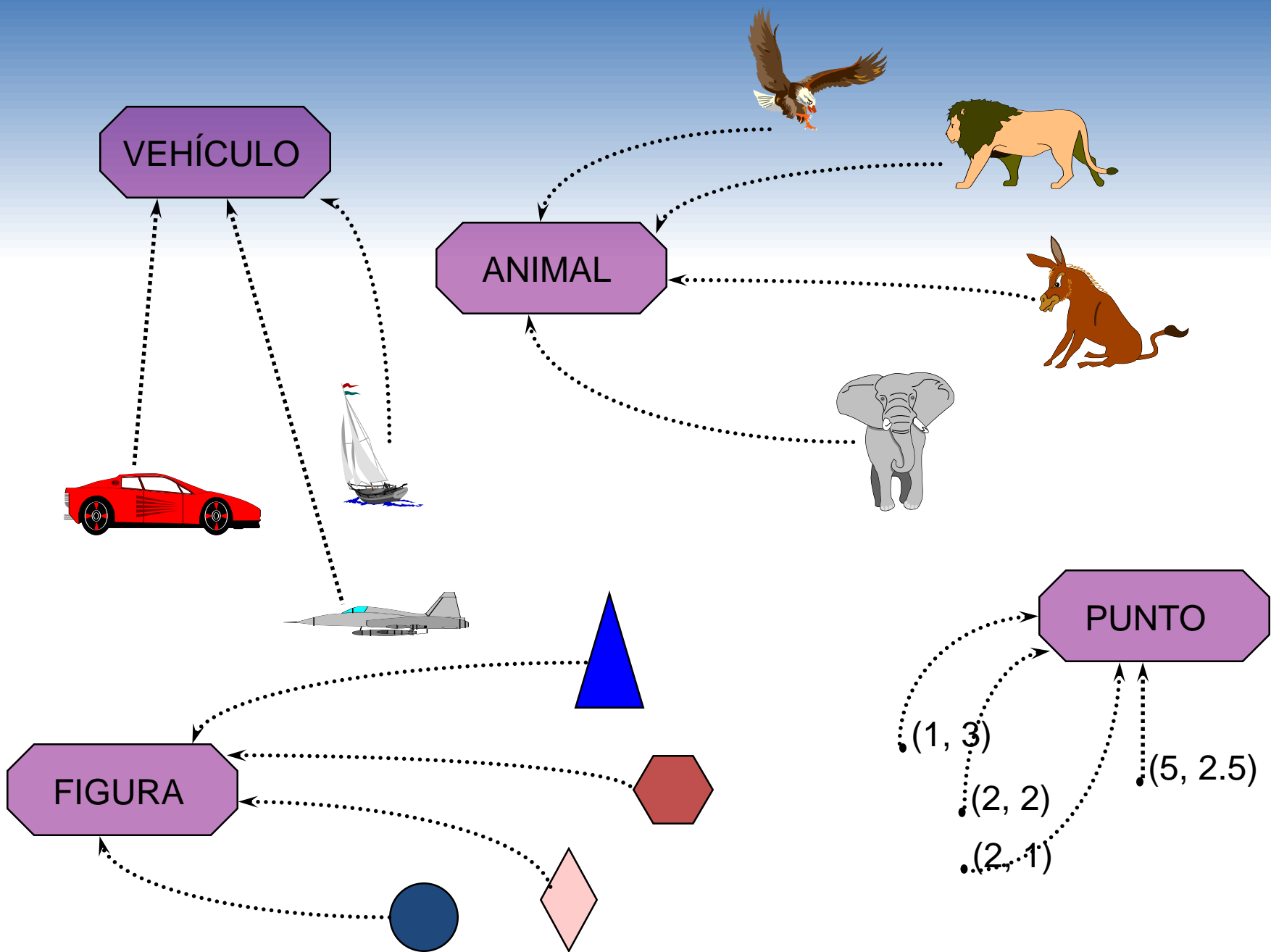
# Contenido

- Evolución de los Lenguajes de Programación
  - Evolución histórica. Paradigmas de Programación
  - Abstracción Procedimental y de Datos
- Conceptos fundamentales de la Programación Orientada a Objetos
  - Clases y Objetos 
  - Métodos y Mensajes
  - Composición
  - Herencia y Redefinición del Comportamiento
  - Polimorfismo y Vinculación Dinámica
  - Clases Abstractas e Interfaces (Tema 2)

# 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: Punto

- Queremos manipular puntos del plano.
  - Utilizando la abstracción, un punto se puede representar por un objeto que contiene dos valores reales (que llamaremos **su estado**)
    - **x** de tipo double. Abscisa
    - **y** de tipo double. Ordenada

```
Punto(x:1, y:3)
```

```
Punto(x:5, y:2.5)
```

# Ejemplo: Punto

- Una vez fijado el estado de un punto, se especifica cómo operar con el punto.
- Por ejemplo
  - Queremos saber los valores de abscisa y ordenada.
  - Queremos modificar los valores de abscisa y ordenada.
  - Queremos trasladar un punto.
  - Queremos calcular la distancia a otro punto.
  - ...
- Estas operaciones definen lo que llamaremos **el comportamiento** del punto.

# Ejemplo. Jarra

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

Jarra(capacidad:7, contenido:3)

Jarra(capacidad:5, contenido:0)

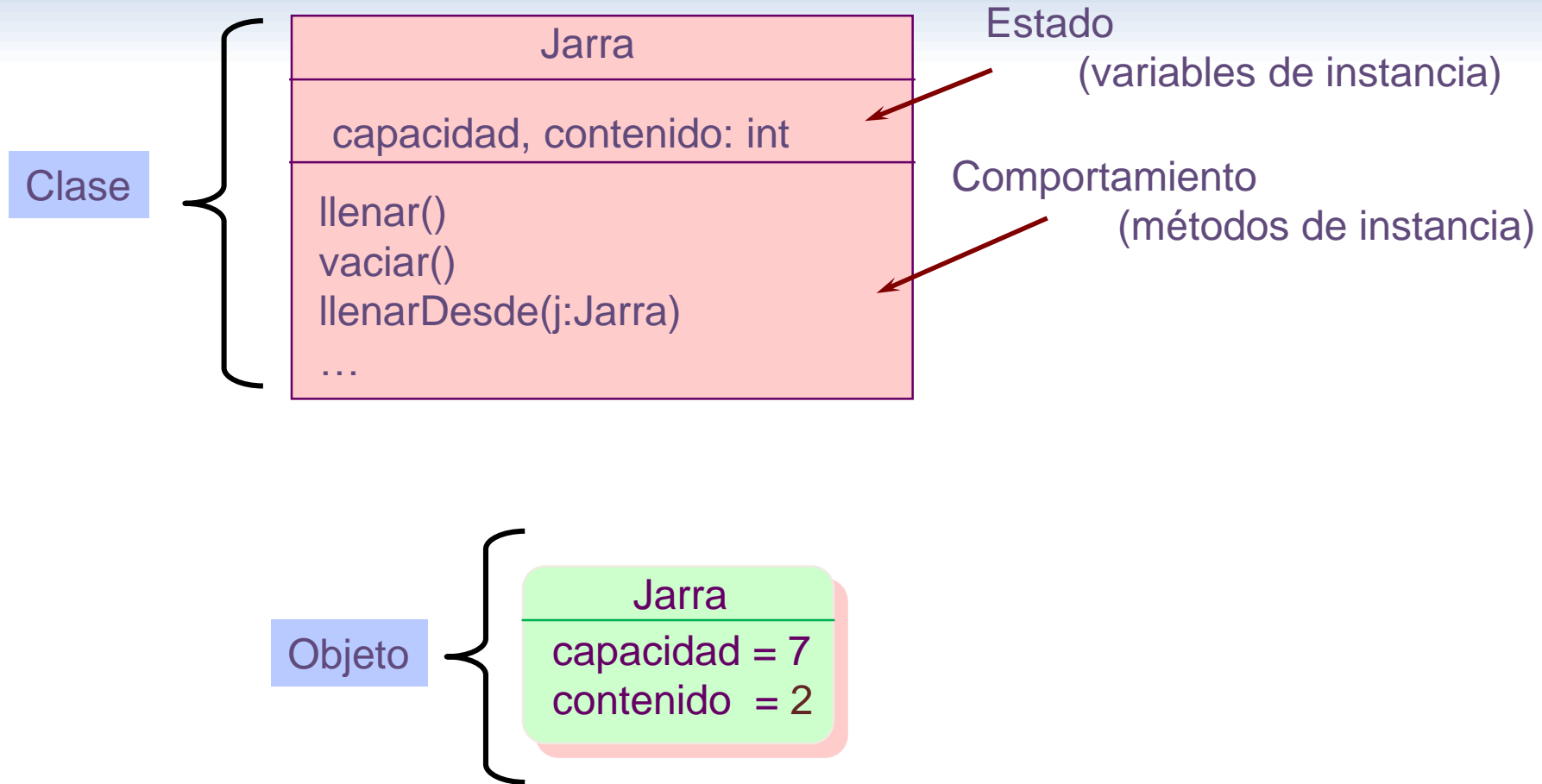
# Ejemplo. Jarra

- ¿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.

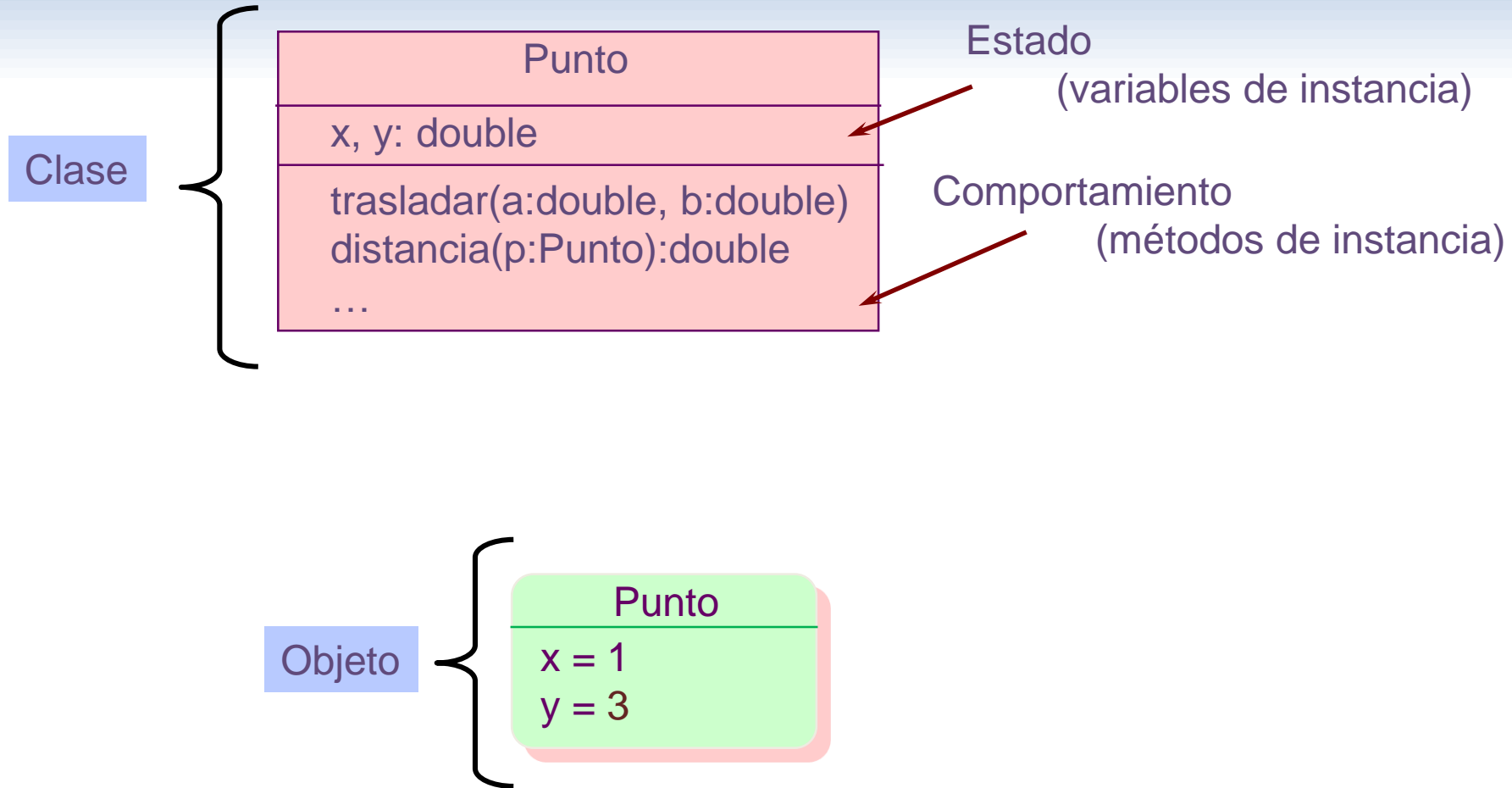
# Puntos y Jarras. Conceptos

- Punto y Jarra son clases (tipos). Definen un estado y un comportamiento.
- Podemos crear muchos puntos y muchas jarras (objetos).
- Cada punto y cada jarra tienen **su propio estado**.
- Un punto **se diferencia** de otro en su estado  
Punto(3,5)   Punto(8,2)   Punto(5,2.5)   Punto(0,0)
- Una jarra **se diferencia** de otra en su estado.  
Jarra(7, 5)   Jarra(8, 8)   Jarra(9,0)   Jarra(2,1)
- Todos los puntos tienen **el mismo comportamiento**
- Todas las jarras tienen **el mismo comportamiento**.
- Cuando se actúa sobre un punto o una jarra, responden conforme a su estado.

# Representación (UML)




# Representación (UML)



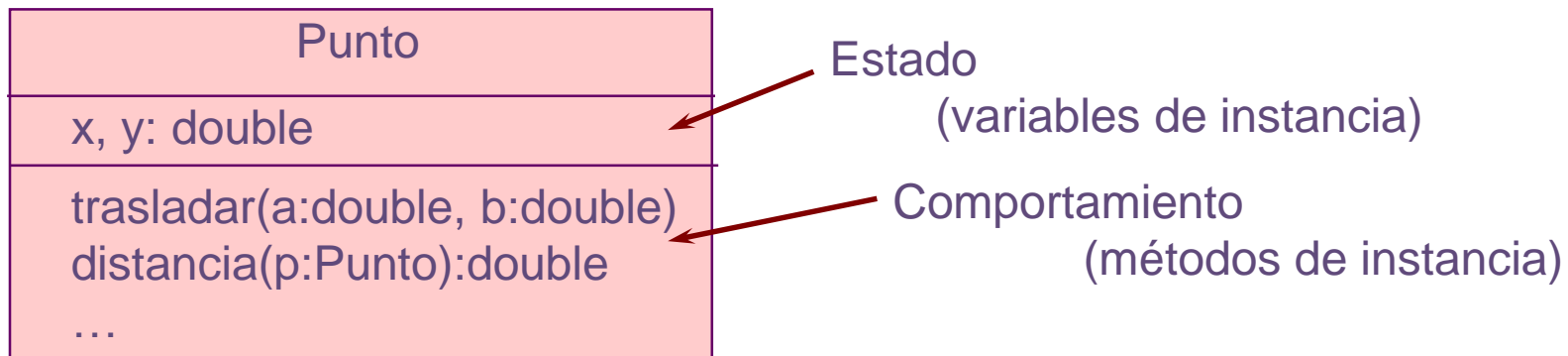


# Contenido

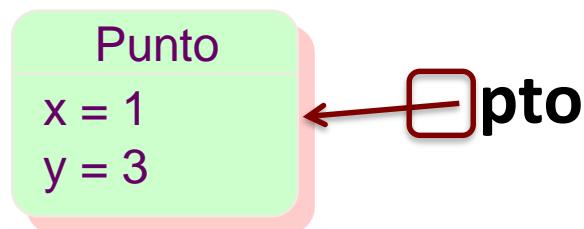
- Evolución de los Lenguajes de Programación
  - Evolución histórica. Paradigmas de Programación
  - Abstracción Procedimental y de Datos
- Conceptos fundamentales de la Programación Orientada a Objetos
  - Clases y Objetos
  - Métodos y Mensajes 
  - Composición
  - Herencia y Redefinición del Comportamiento
  - Polimorfismo y Vinculación Dinámica
  - Clases Abstractas e Interfaces (Tema 2)

# Métodos y Mensajes

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

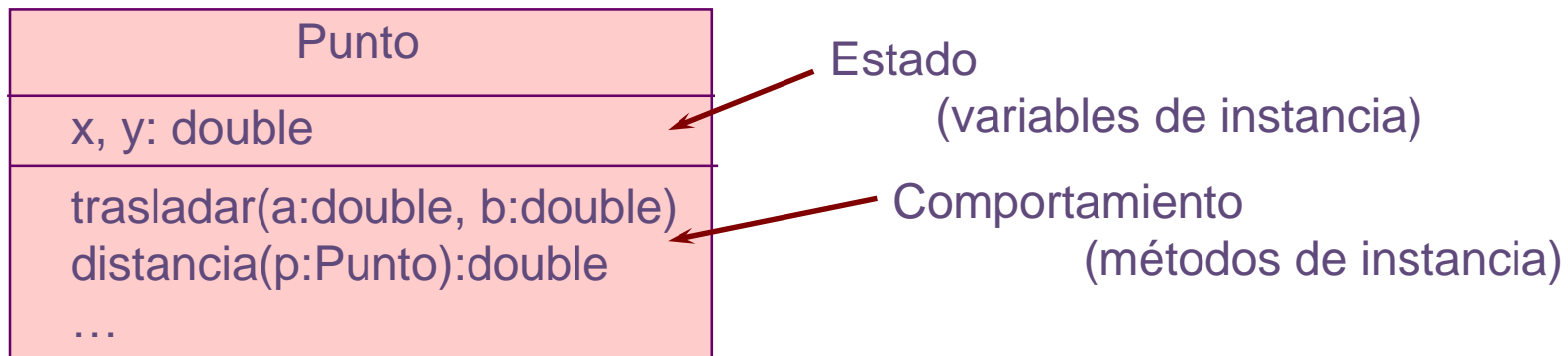


- Invocación a métodos: Paso de **Mensajes**  
**obj.mens(args)**



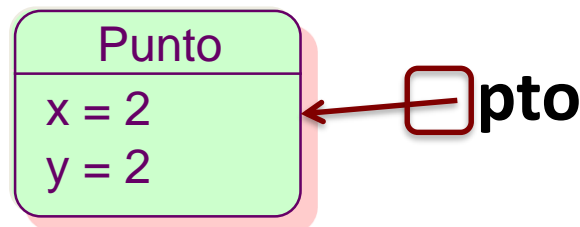
# Métodos y Mensajes

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



- Invocación a métodos: Paso de **Mensajes**  
**obj.mens(args)**

**pto.trasladar(1, -1)**  
=====⇒



# 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**.
- El número de parámetros de un método en la programación orientada a objetos es “uno menos” que en la programación imperativa (**parámetro implícito**).  
Ej: `pto.trasladar(1,-1)` frente a `trasladar(pto,1,-1)`
- 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.

# Clases

- Estructuras que encapsulan *variables 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(this.x - pto.x, 2) +  
            Math.pow(this.y - pto.y, 2));  
    }  
}
```

**VARIABLES**

**CONSTRUCTORES**

**MÉTODOS**

# Clases

- Estructuras que encapsulan *variables 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**

**CONSTRUCTORES**

**MÉTODOS**

# Clases

- Estructuras que encapsulan *variables 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 x){ this.x = x; }  
    public void ordenada(double y){ this.y = y; }  
    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**

**CONSTRUCTORES**

**MÉTODOS**

# Clases

- Estructuras que encapsulan *variables y métodos*

“Punto.java”

```
public class Punto {  
    private double x, y;  
    public Punto() { this(0,0); }  
    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(this.x - pto.x, 2) +  
            Math.pow(this.y - pto.y, 2));  
    }  
}
```

**VARIABLES**

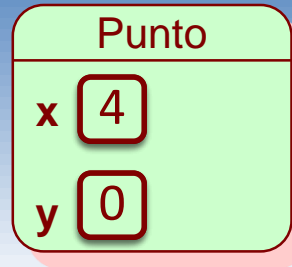
**CONSTRUCTORES**

**MÉTODOS**



# Objetos

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



pto

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

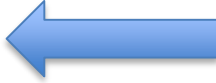
```
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }
```

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

*trasladar(3,-1)*

```
Punto pto = new Punto(1,1);  
pto.trasladar(3,-1);
```

# Contenido

- Evolución de los Lenguajes de Programación
  - Evolución histórica. Paradigmas de Programación
  - Abstracción Procedimental y de Datos
- Conceptos fundamentales de la Programación Orientada a Objetos
  - Clases y Objetos
  - Métodos y Mensajes
  - Composición 
  - Herencia y Redefinición del Comportamiento
  - Polimorfismo y Vinculación Dinámica
  - Clases Abstractas e Interfaces (Tema 2)

# Composición

- Mecanismo que permite la definición de nuevas clases a partir de otras ya definidas.
- Responde a una relación de tipo *“tiene”* o *“está compuesto/a por”*.

# Composición

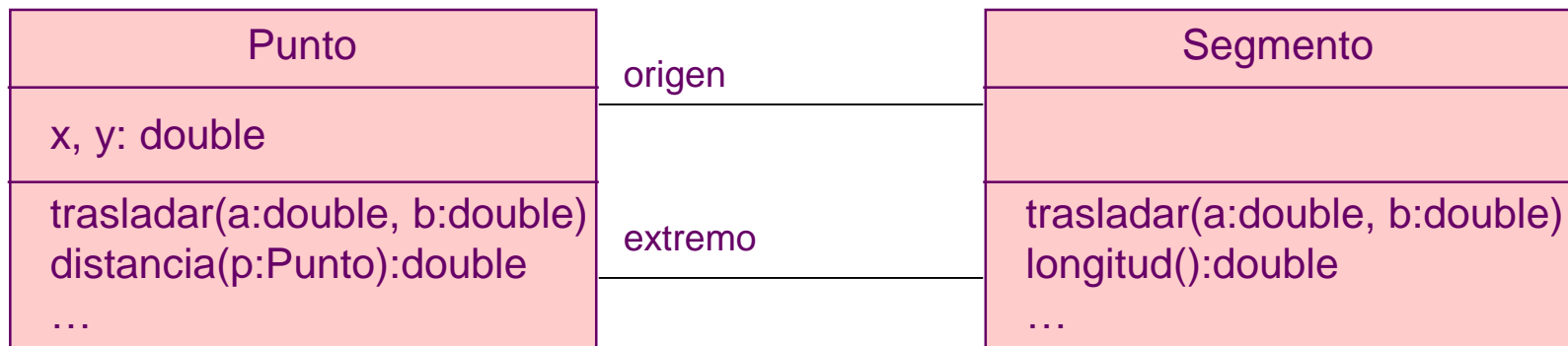
- Mecanismo que permite la definición de nuevas clases a partir de otras ya definidas.
- Responde a una relación de tipo *“tiene”* o *“está compuesto/a por”*.
- Así, por ej., un segmento **tiene** dos puntos (origen y extremo)
  - También podemos decir que los puntos origen y extremo *“forman parte del”* segmento, o que el segmento *“está compuesto por”* dos puntos

Punto
x, y: double
trasladar(a:double, b:double) distancia(p:Punto):double ...

Segmento
origen, extremo: Punto
trasladar(a:double, b:double) longitud():double ...

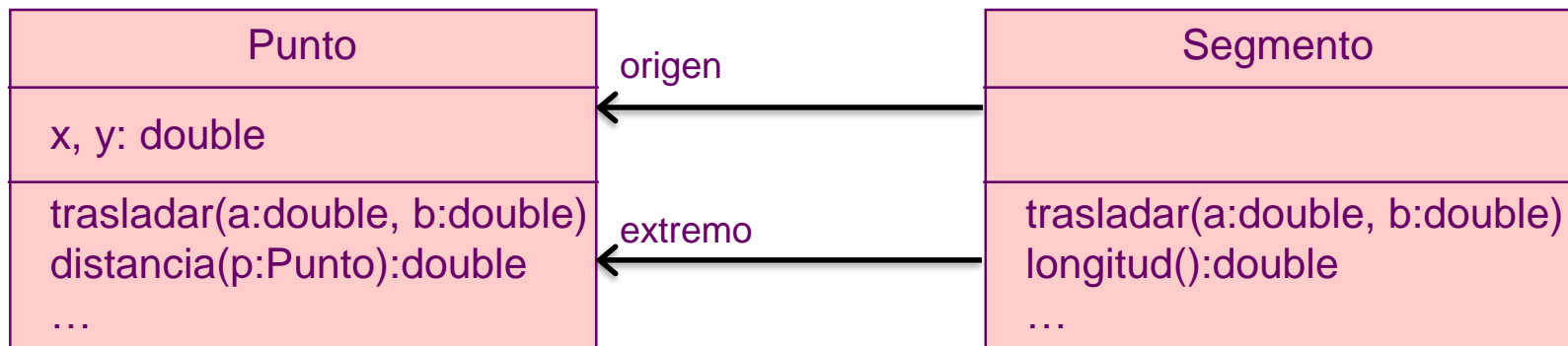
# Composición

- Mecanismo que permite la definición de nuevas clases a partir de otras ya definidas.
- Responde a una relación de tipo *“tiene”* o *“está compuesto/a por”*.
- Así, por ej., un segmento **tiene** dos puntos (origen y extremo)
  - También podemos decir que los puntos origen y extremo *“forman parte del”* segmento, o que el segmento *“está compuesto por”* dos puntos



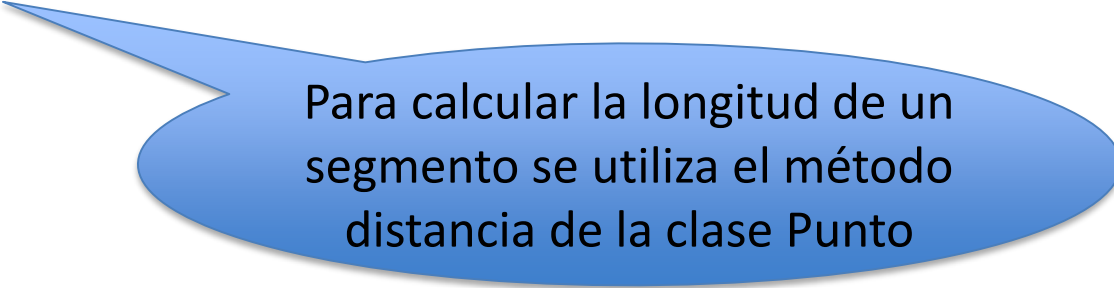
# Composición

- Mecanismo que permite la definición de nuevas clases a partir de otras ya definidas.
- Responde a una relación de tipo *“tiene”* o *“está compuesto/a por”*.
- Así, por ej., un segmento **tiene** dos puntos (origen y extremo)
  - También podemos decir que los puntos origen y extremo *“forman parte del”* segmento, o que el segmento *“está compuesto por”* dos puntos



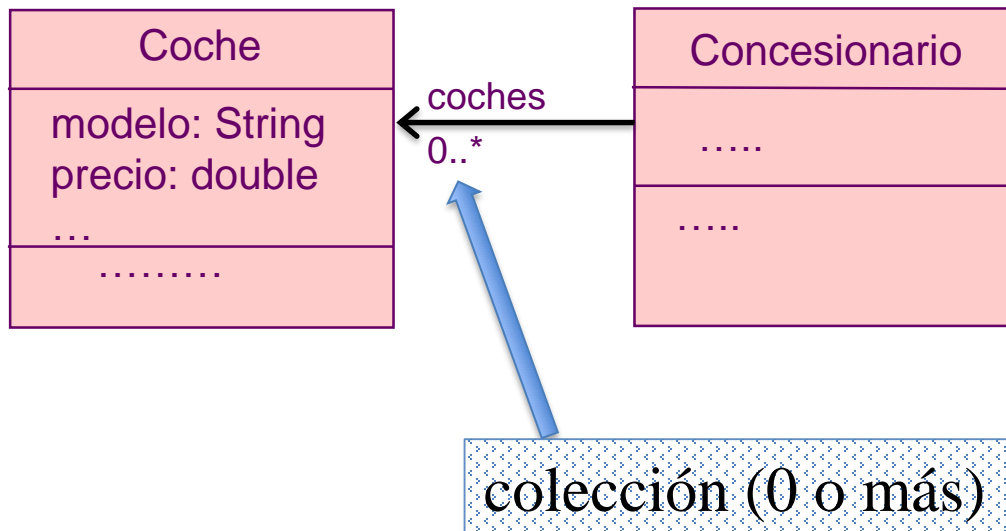
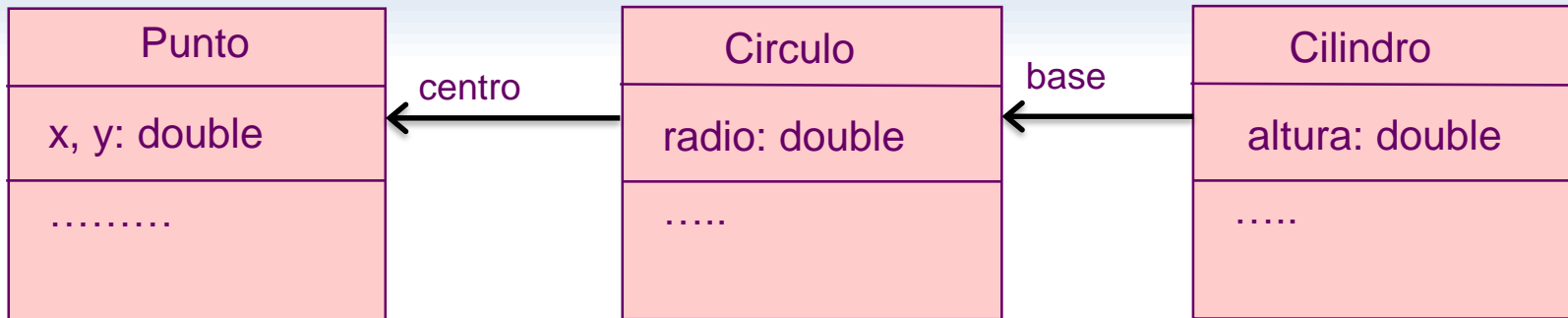
# Composición

```
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

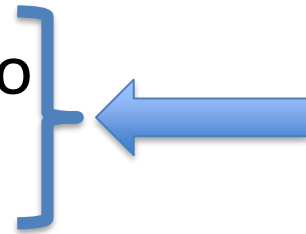
# Otros ejemplos de composición





# Contenido

- Evolución de los Lenguajes de Programación
  - Evolución histórica. Paradigmas de Programación
  - Abstracción Procedimental y de Datos
- Conceptos fundamentales de la Programación Orientada a Objetos
  - Clases y Objetos
  - Métodos y Mensajes
  - Composición
  - Herencia y Redefinición del Comportamiento
  - Polimorfismo y Vinculación Dinámica
  - Clases Abstractas e Interfaces (Tema 2)

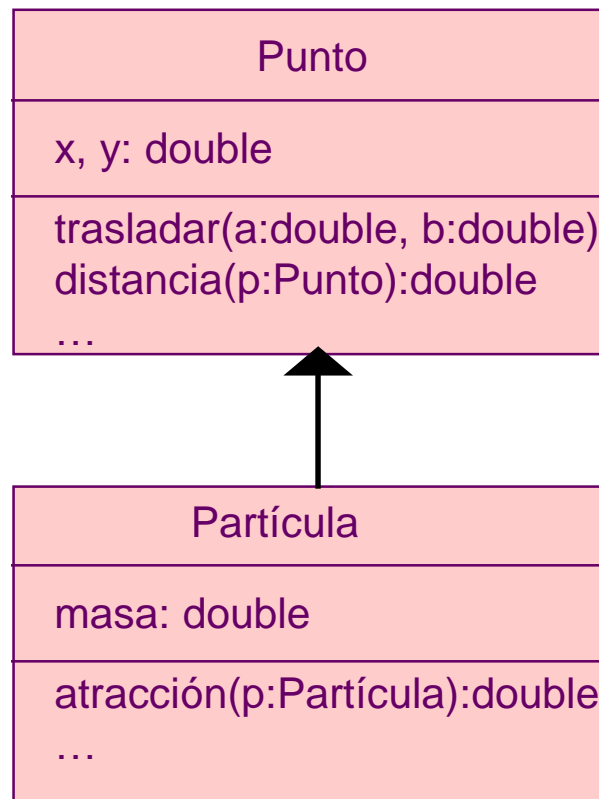


# Herencia

- Mecanismo que, como la composición, también permite la definición de nuevas clases a partir de otras ya definidas.
- Pero la Herencia responde a una relación de tipo “*es un/a*”.

# Herencia

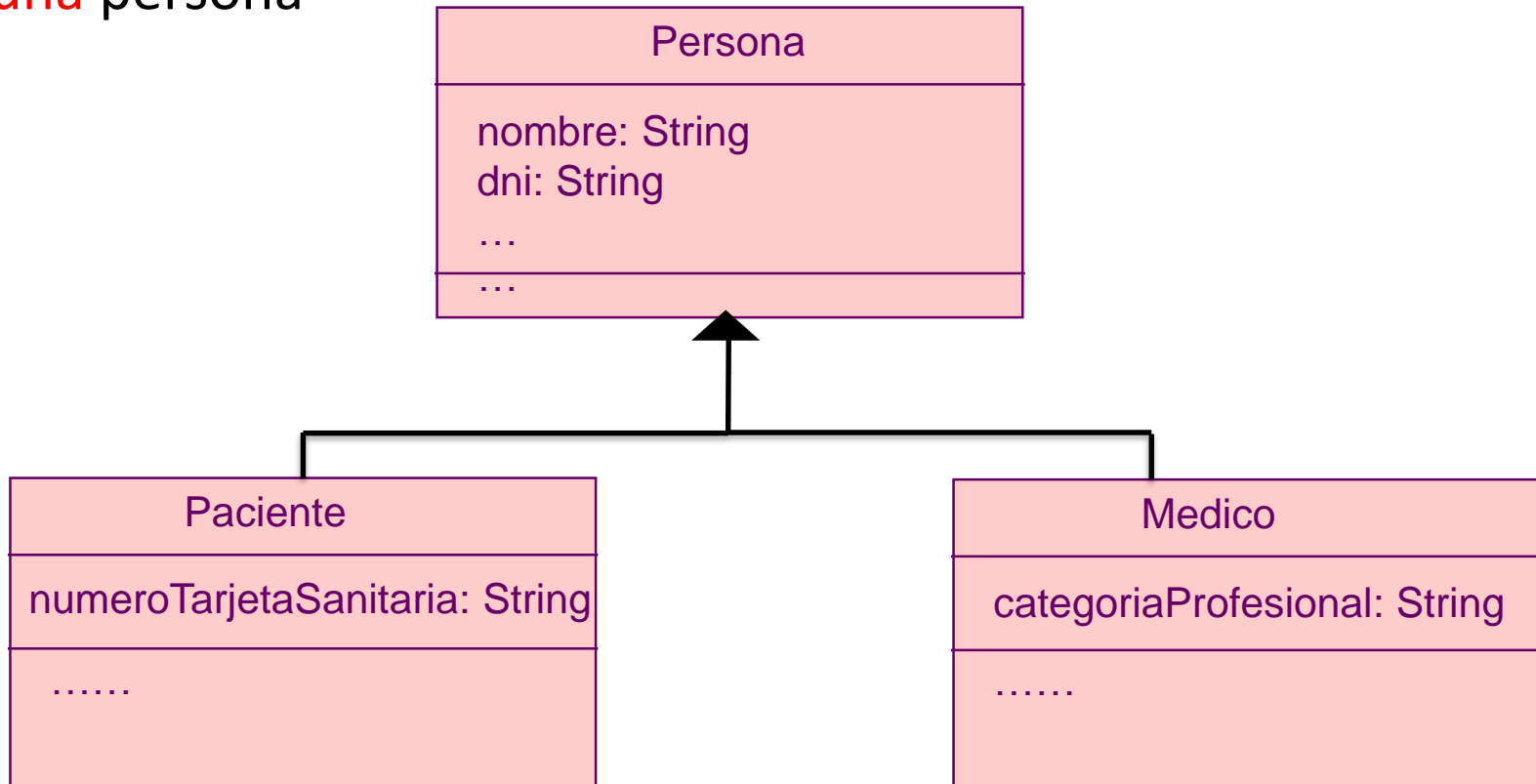
- Mecanismo que, como la composición, también permite la definición de nuevas clases a partir de otras ya definidas.
- Pero la Herencia responde a una relación de tipo “*es un/a*”.
- Así, por ej., una partícula *es un* punto (con masa)



↑ Otra forma

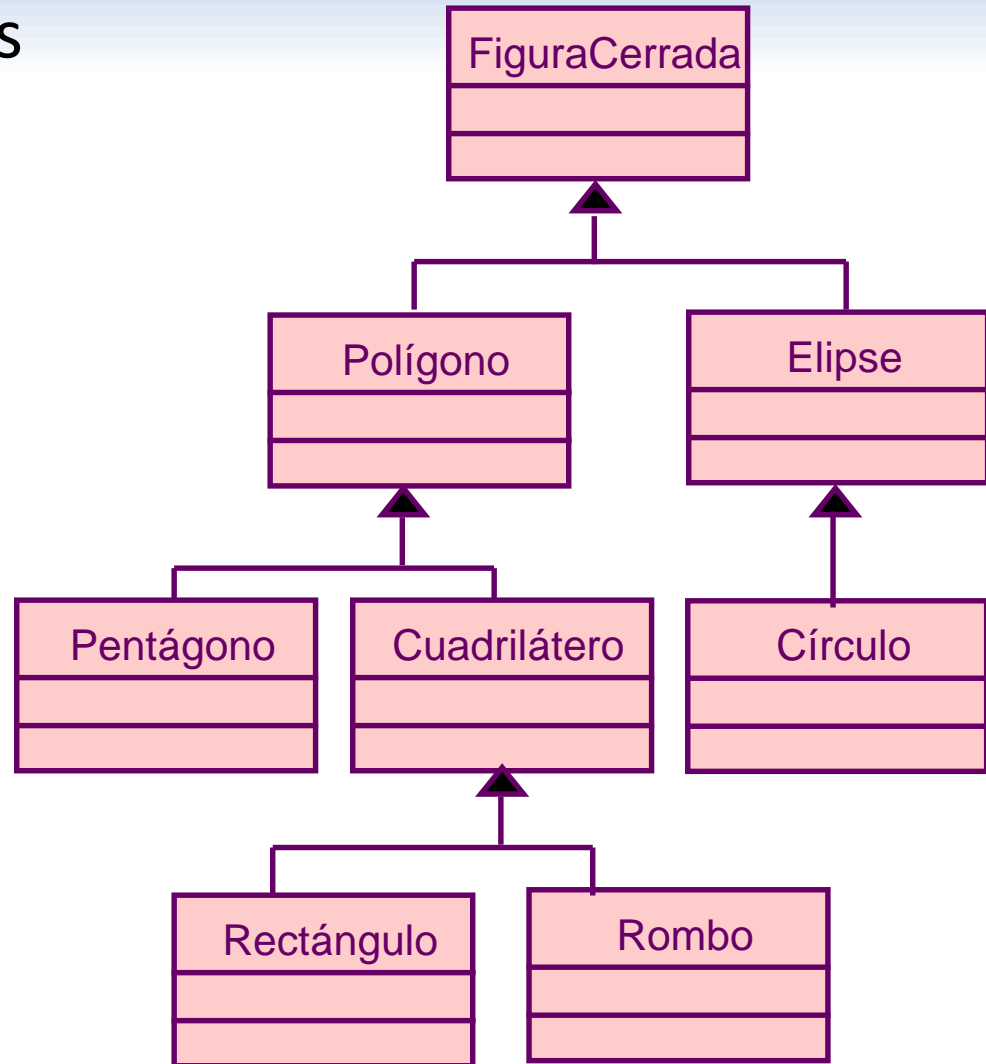
# Herencia

- Mecanismo que, como la composición, también permite la definición de nuevas clases a partir de otras ya definidas.
- Pero la Herencia responde a una relación de tipo “*es un/a*”.
- Otro ejemplo: un paciente *es una* persona y un médico también *es una* persona



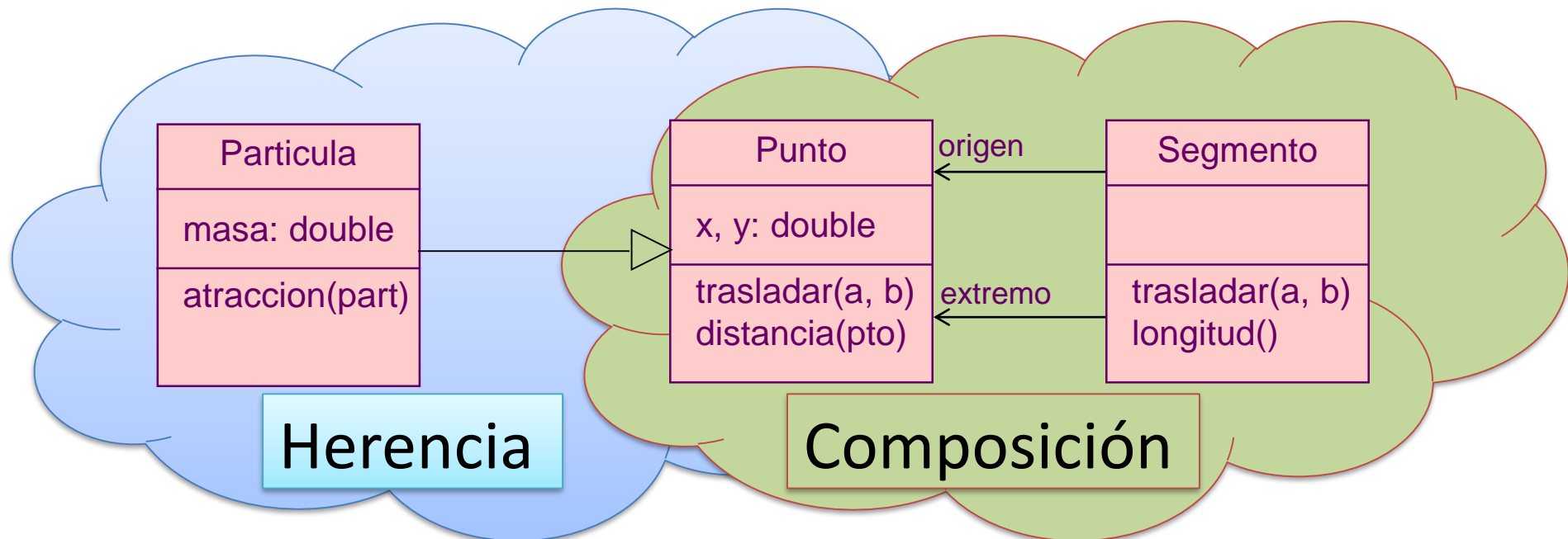
# Herencia

- Permite clasificar las clases en una jerarquía



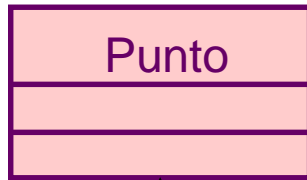
# Herencia vs. composición

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



# Herencia

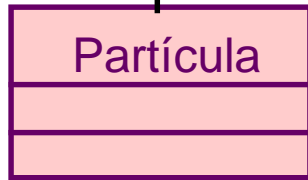
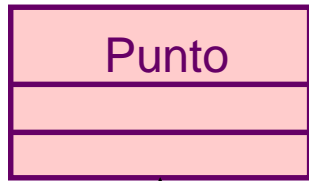
Padres / Ascendientes /  
Superclase



Hijos / Descendientes /  
Subclase

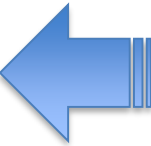
# Herencia

Padres / Ascendientes /  
Superclase



Hijos / Descendientes /  
Subclase

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





# Herencia

Estado

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

Comportamiento

# Herencia

```
public class Partícula extends Punto {  
    final static double G = 6.67e-11;  
    private double masa;  
    public Partícula(double m) {  
        this(0,0,m);  
    }  
    public Partícula(double a, double b, double m) {  
        super(a, b);  
        masa = m;  
    }  
    public void masa(double m) { masa = m; }  
    public double masa() { return masa; }  
    public double atracción(Partícula part) {  
        return G * this.masa * part.masa /  
            Math.pow(this.distancia(part), 2);  
    }  
}
```

Estado  
(+ Estado  
Punto)

Se refiere a  
Partícula(double, double,  
double)

Se refiere a  
Punto(double, double)

Comportamiento  
(+ Comportamiento  
Punto)

Heredada de  
Punto

# Herencia

```
public class Partícula extends Punto {  
    final static double G = 6.67e-11;  
    private double masa;  
    public Partícula(double m) {  
        this(0,0,m);  
    }  
    public Partícula(double a, double b, double m) {  
        super(a, b);  
        masa = m;  
    }  
    public void masa(double m) { masa = m; }  
    public double masa() { return masa; }  
    public double atracción(Partícula part) {  
        return G * masa * part.masa /  
            Math.pow(distancia(part), 2);  
    }  
}
```

Estado  
(+ Estado  
Punto)

Se refiere a  
Partícula(double, double,  
double)

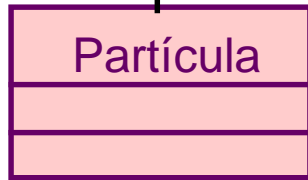
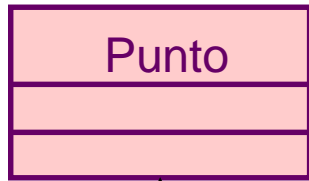
Se refiere a  
Punto(double, double)

Heredada de  
Punto

Comportamiento  
(+ Comportamiento  
Punto)

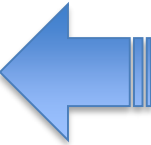
# Herencia

Padres / Ascendientes /  
Superclase



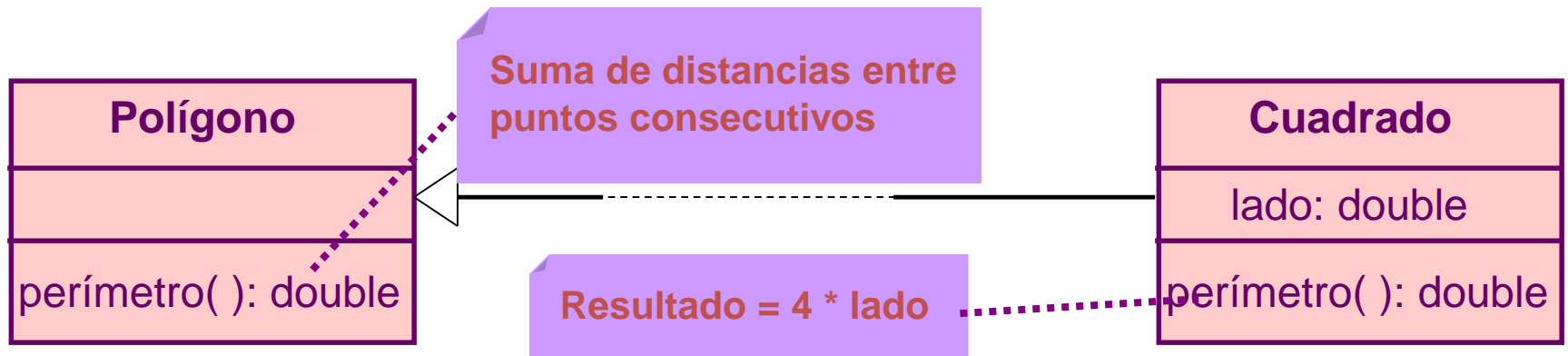
Hijos / Descendientes /  
Subclase

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



# Redefinición del comportamiento

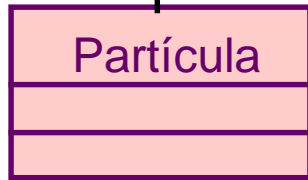
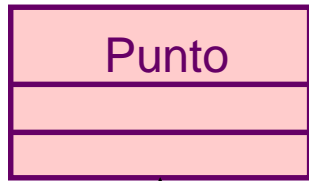
- En la mayoría de lenguajes orientados a objetos las clases herederas pueden heredar un método, y luego redefinirlo, modificando su implementación.



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

# Herencia

Padres / Ascendientes /  
Superclase



Hijos / Descendientes /  
Subclase


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




# Polimorfismo sobre los datos

- Un lenguaje tiene **capacidad polimórfica** sobre los datos cuando
  - una variable declarada de un tipo (o clase) determinado –*tipo estático*– 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 LOOs está habitualmente restringida por la relación de herencia:
  - El *tipo dinámico* debe ser **descendiente** del *tipo estático*.

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



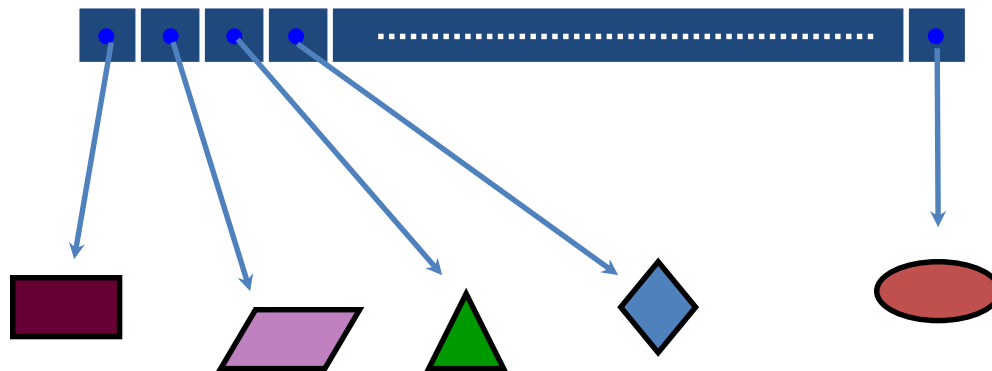
Tipo estático  
de **pto**



Tipo dinámico  
de **pto**

# Polimorfismo sobre los datos

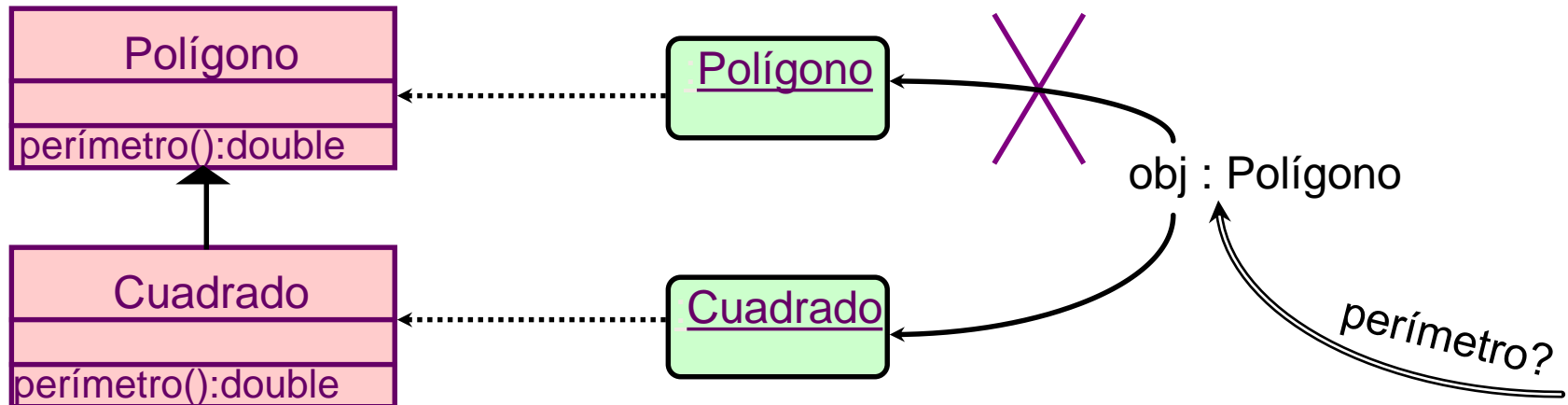
- Una variable puede referirse a objetos de clases distintas de la que se ha declarado. Esto afecta a:
  - asignaciones explícitas entre objetos,
  - paso de parámetros,
  - devolución de resultado en una función.
- La restricción dada por la herencia permite construir estructuras con elementos de naturaleza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