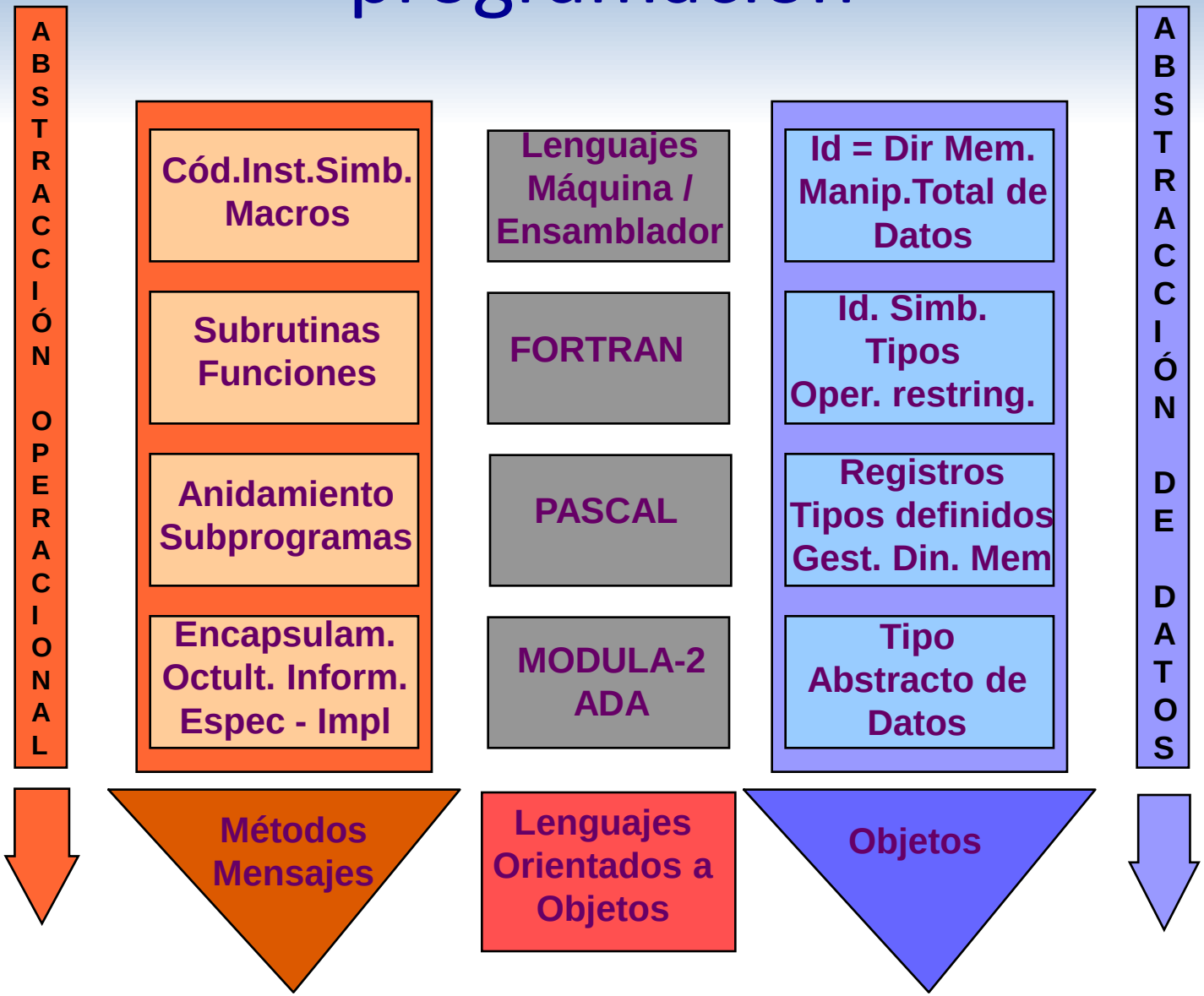
Introducción a la Programación Orientada a Objetos

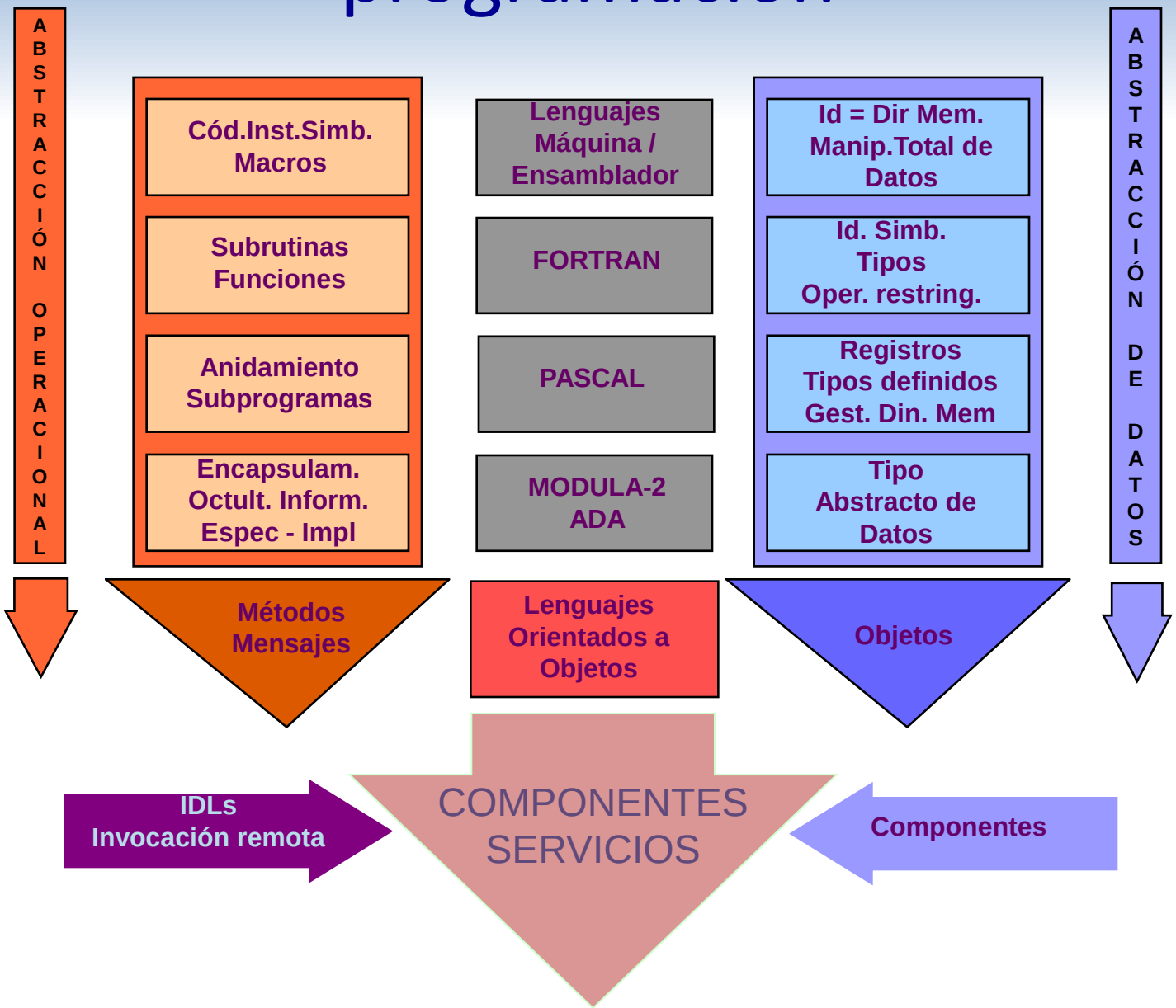
Contenido

- 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
  - Clases abstractas e interfaces

Evolución de los lenguajes de programación



Evolución de los lenguajes de programación

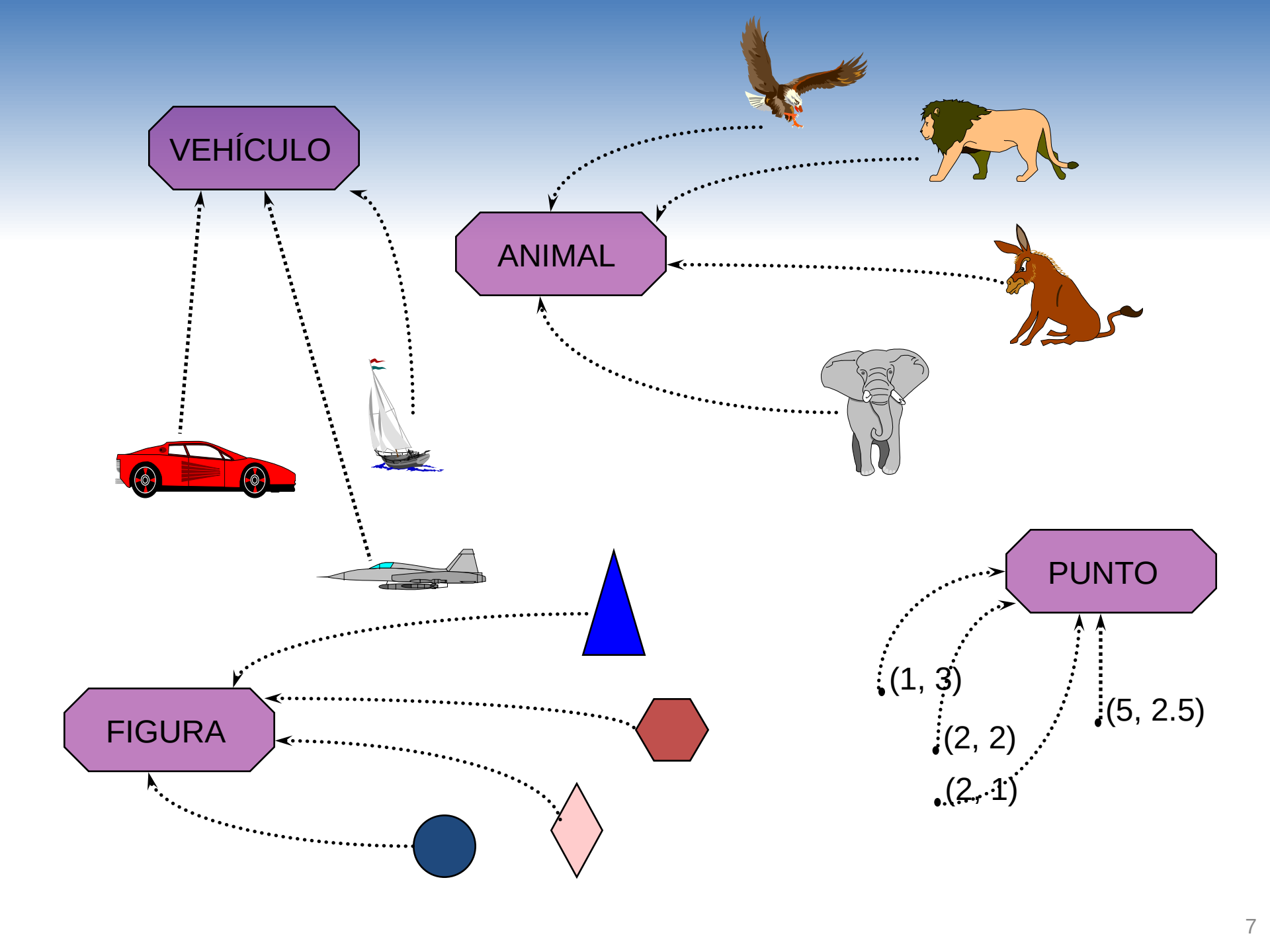


Conceptos básicos de la P.O.O.

- Clases y Objetos
- Métodos y Mensajes
- Herencia
- Polimorfismo y Vinculación Dinámica
- Clases abstractas e interfaces

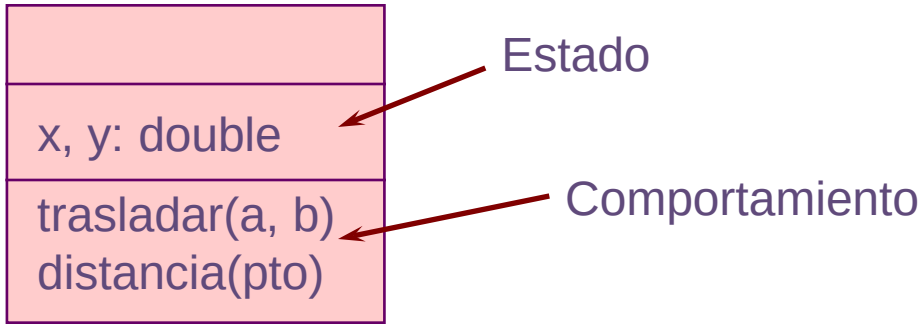
Clases y Objetos

- 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


VEHÍCULO
ANIMAL
PUNTO
FIGURA
(1, 3)
(2, 2)
(2, 1)
(5, 2.5)

Métodos y Mensajes

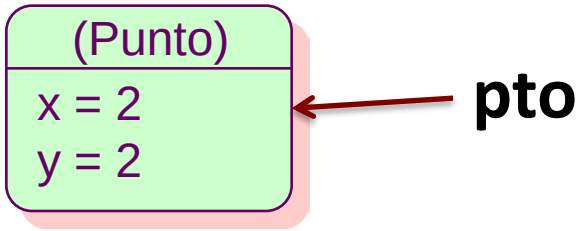
- **Métodos**: definen el comportamiento de una clase



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

obj.mens(args) **mens(obj, 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.
- 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.

Clases

- **Estructuras que encapsulan *datos* y *métodos***

"Punto.java"

```
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

"Punto.hpp"

```
class Punto{
    double x, y;
  public:
    Punto(){x = y = 0;}
    Punto(double a, double b): x(a),y(b) {}
    double abscisa() {return x;}
    double ordenada() {return y;}
    void abscisa(double a){ x = a; }
    void ordenada(double b){ y = b; }
    void trasladar(double a, double b);
    double distancia(Punto&);
};
```

VARIABLES DE ESTADO (DATOS MIEMBRO)

CONSTRUCTORES

MÉTODOS (FUNCIONES MIEMBRO)

"Punto.cpp"

```
#include "Punto.hpp"
#include <math.h>

void Punto::trasladar(double a, double b) {
     x += a;
     y += b;
};

double Punto::distancia(Punto& pto){
     return sqrt(pow(x - pto.x, 2) +
                 pow(y – pto.y, 2));
};
```

```
class Punto
creation origen, nuevo;
feature
  x, y: REAL;
  origen is do x := 0; y := 0 end;
  nuevo(a, b: REAL) is
    do x := a; y := b end;
  abscisa(a: REAL) is
    do x := a end;
  ordenada(b: REAL) is
    do y := b end;
  trasladar(a, b: REAL) is
    do x := x + a; y := y + b end;
  distancia(pto: Punto): REAL is
    do
     Result := sqrt(pow(x - pto.x, 2)
                  + pow(y - pto.y, 2))
    end
end Punto;
```

ATRIBUTOS

CONSTRUCTORES

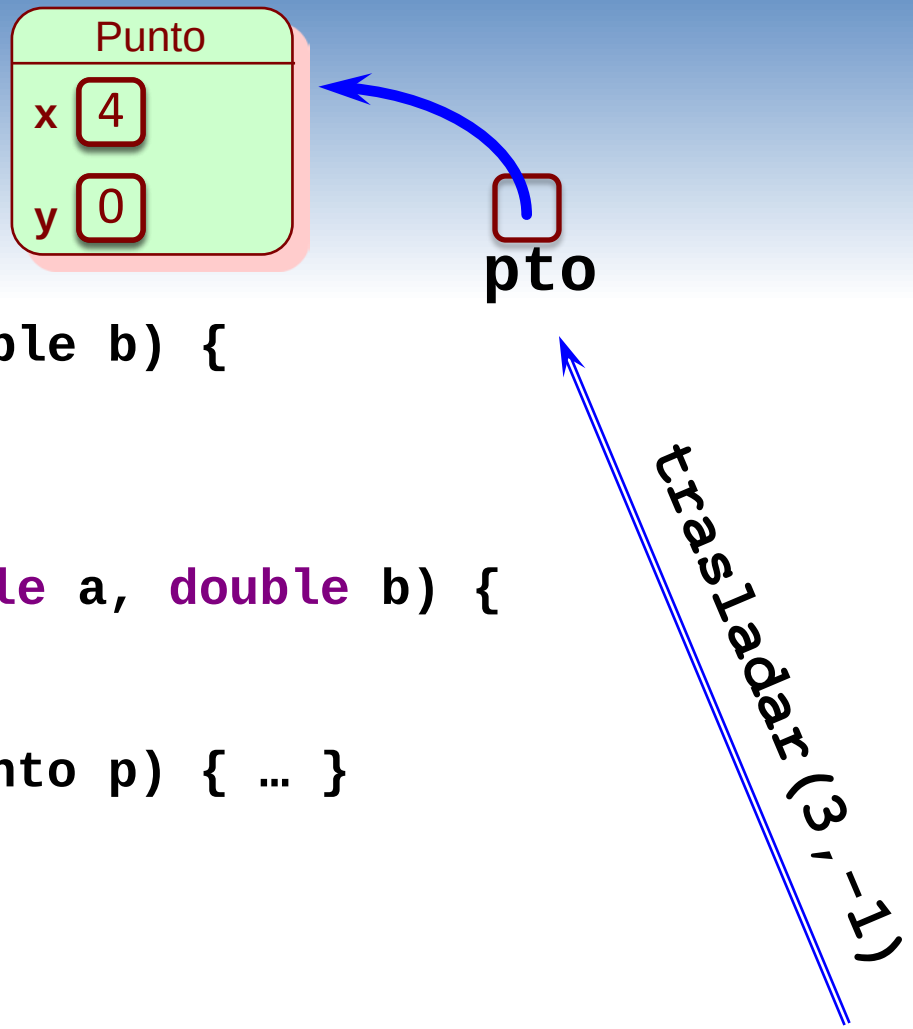
RUTINAS

```
Object subclass: #Punto
        instanceVariableNames: ‘ x y ’
        classVariableNames: "
        poolDictionaries: "
origen
  ^(self new) abscisa: 0; ordenada: 0
x: unNum y: otroNum
  ^(self origen) tras: unNum ladar: otroNum
abscisa
  ^x
ordenada
  ^y
abscisa: unNum
  x := unNum
ordenada: unNum
  y := unNum
tras: unNum ladar: otroNum
  x := x + unNum. y := y + otroNum
distancia: unPunto
  ^ ((x - unPunto abscisa) squared +
      (y - unPunto ordenada) squared) sqrt
```

Métodos de clase

Métodos de instancia

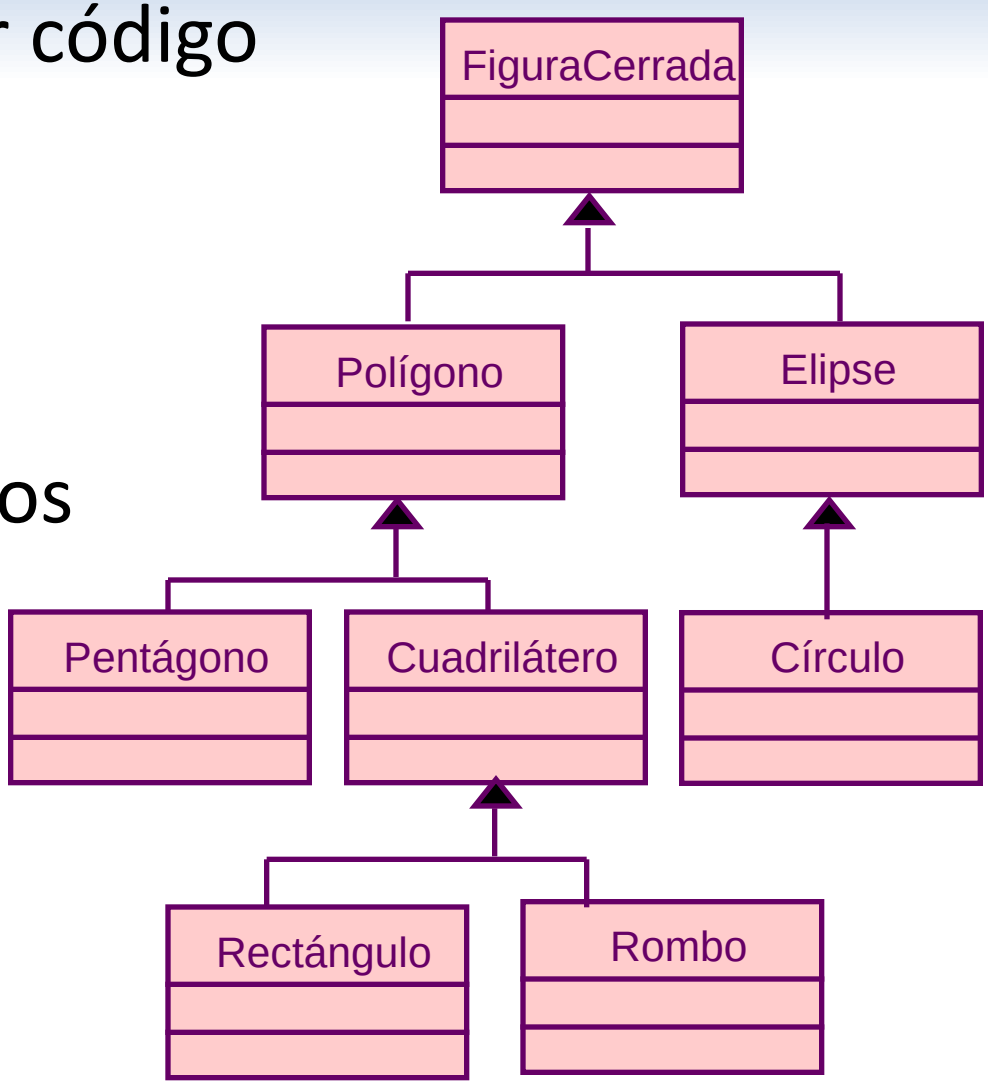
```
class Punto {
   private double x, y;

   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) { … }
};
```



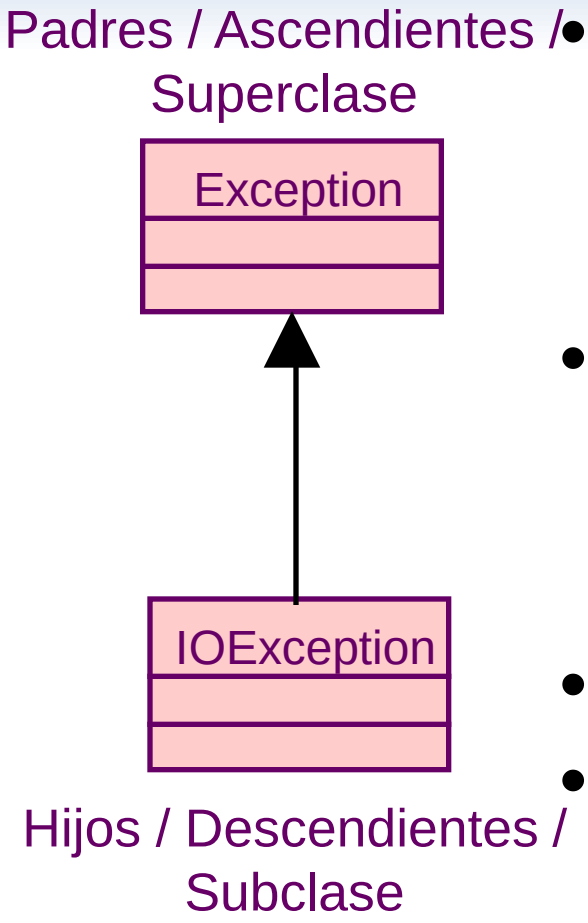

```
Punto pto = new Punto(1,1);

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

Herencia

- Posibilidad de reutilizar código
- Algo más que:
  - incluir ficheros, o
  - importar módulos
- Permite clasificar objetos
- Simple versus múltiple



Herencia



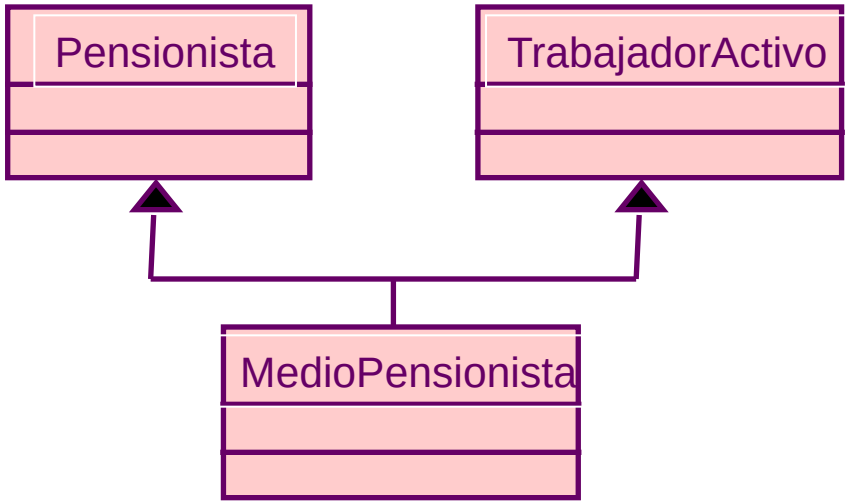
- Una subclase dispone de los atributos y métodos de la superclase, y puede añadir otros nuevos.
- La subclase puede modificar el comportamiento heredado (por ejemplo, redefiniendo algún método heredado) .
- La herencia es transitiva.
- Los objetos de una clase que hereda de otra pueden verse como objetos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

```
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));
  }
}
```

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

   public Partícula(double m) {
      super(0, 0);
      masa = 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) {
      double d = this.distancia(part);
      return G * masa * part.masa() / (d * d);
   }
}
```

Herencia simple y múltiple

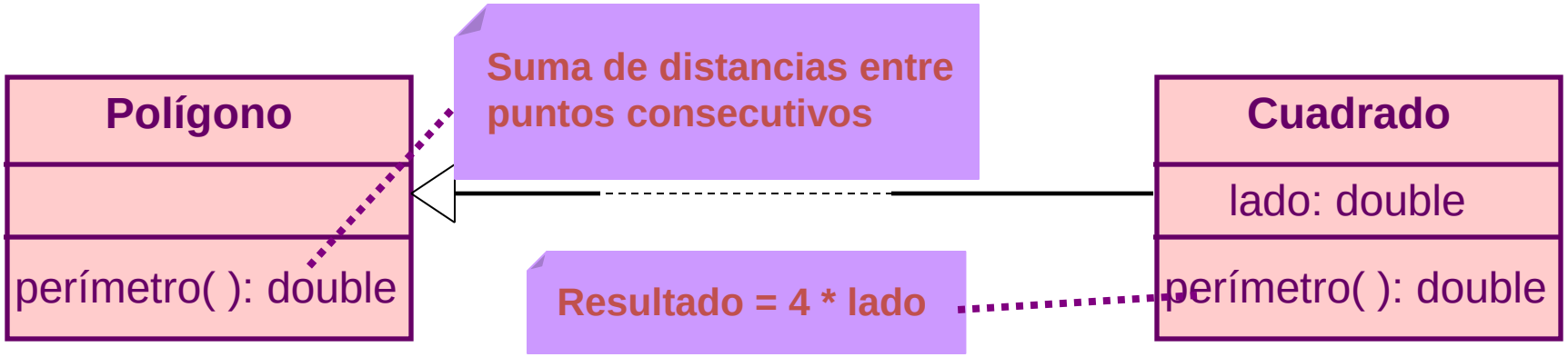
- Existen lenguajes con herencia múltiple, lo que permite que una clase reutilice la funcionalidad ofrecida por varias clases.



- Lenguajes con herencia múltiple: C++, Eiffel
- Lenguajes con herencia simple: Java, C#, Smalltalk

Redefinición del comportamiento

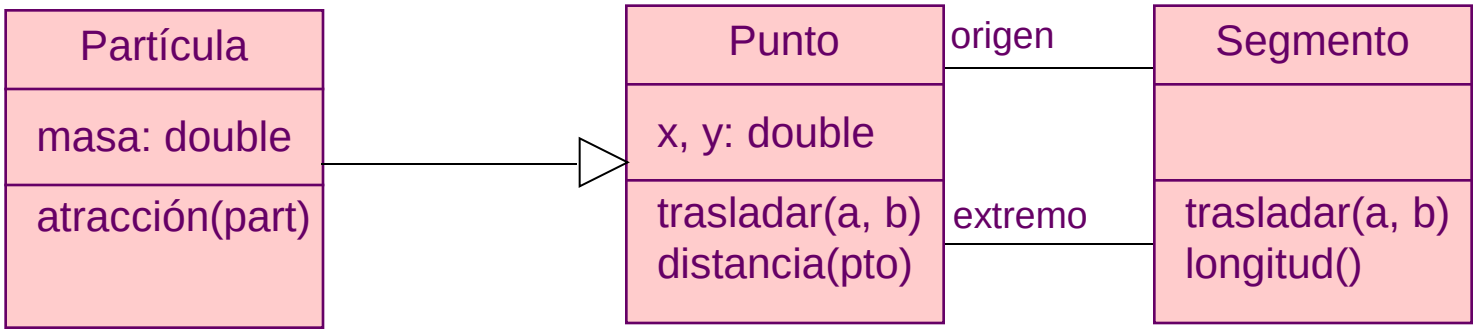
- En la mayoría de lenguajes orientados a objetos las clases herederas pueden heredar un método o servicio, y luego redefinirlo, modificando su implementación.



- Independientemente de que la redefinición es habitual, su efecto puede variar de unos a otros, y puede limitarse mediante el uso de construcciones diversas:
  - Por ejemplo, en Java se pueden declarar métodos y atributos como **final**, impidiendo que sus herederos los redefinan.

Composición

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



Composición

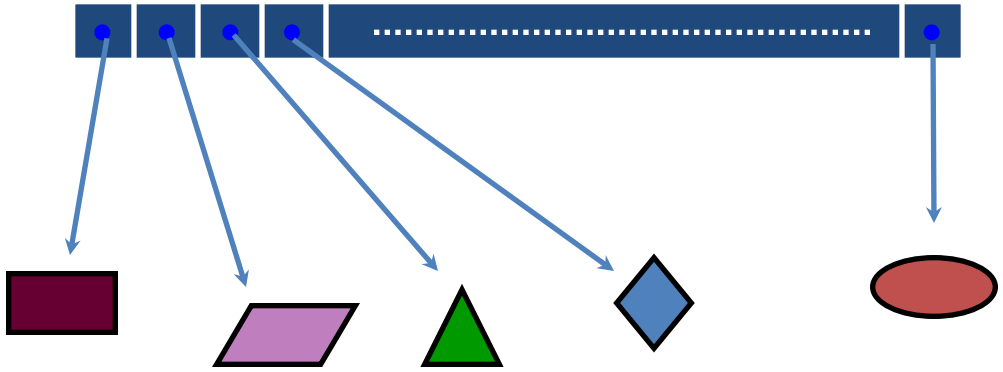
```
class Segmento {
  private Punto origen, extremo;
  public Segmento(double x1, double y1, double x2, double y2) {
     origen = new Punto(x1, y1);
     extremo = new Punto(x2, y2);
  }

  ... // Otros métodos

  public double longitud() {
     return origen.distancia(extremo);
  }
}
```

Polimorfismo sobre los datos

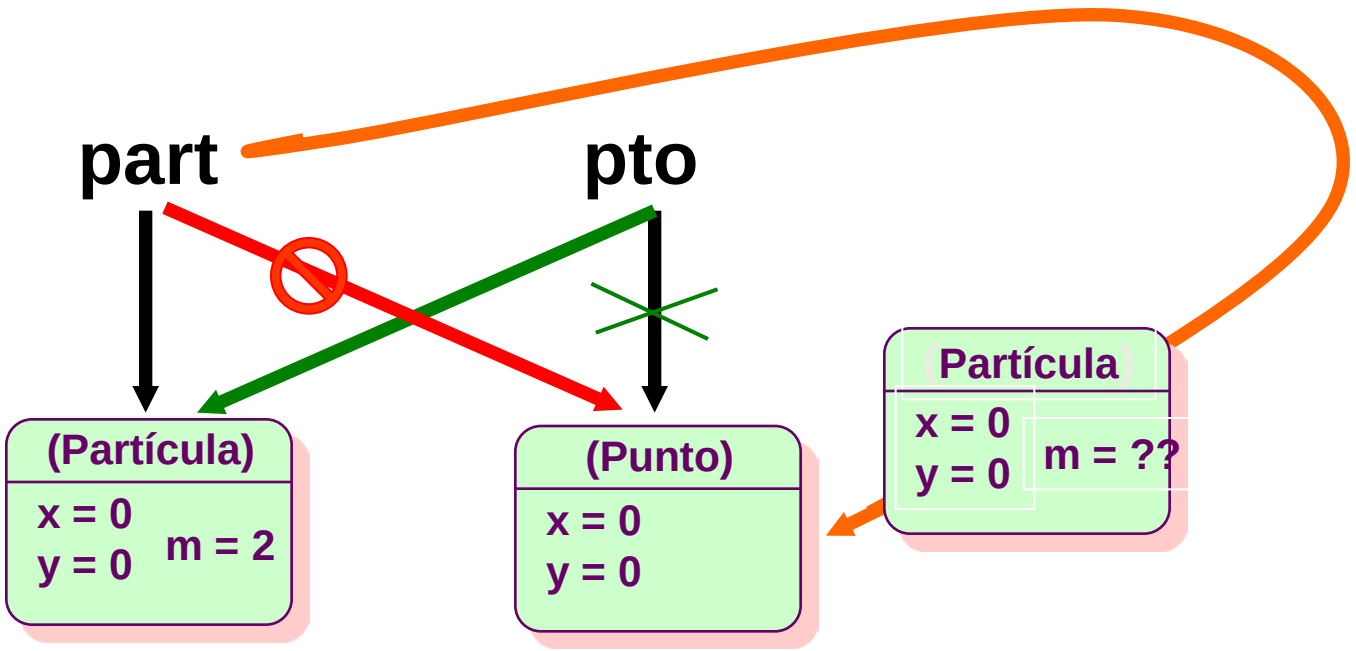
- Un lenguaje tiene capacidad polimórfica sobre los datos cuando una variable declarada de un tipo (o clase) –*tipo estático*– determinado puede hacer referencia en tiempo de ejecución a valores (objetos) de tipo (clase) distinto –*tipo dinámico* –.
- La capacidad polimórfica de un lenguaje no suele ser ilimitada, y en los LOOs está habitualmente restringida por la relación de herencia:
  - El tipo dinámico debe ser descendiente del tipo estático.
- El polimorfismo sobre los datos sólo se aplica a las referencias a objetos:
  - En Java cualquier variable es una referencia a un objeto.
  - En C++ se distingue entre objetos y punteros a objetos. El polimorfismo sólo se puede aplicar a estos últimos.

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:

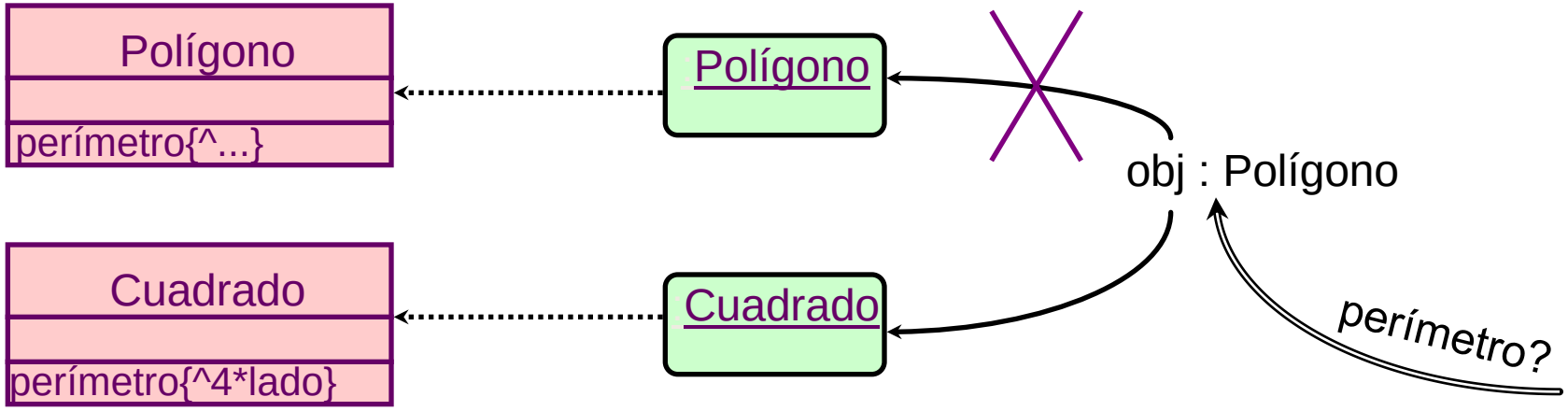


```
Punto pto = new Punto();
Partícula part = new Partícula(2);

pto  = part; // Asignación correcta
part = pto; // Asignación incorrecta
part = (Partícula) pto; // Peligroso
```



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 receptor.

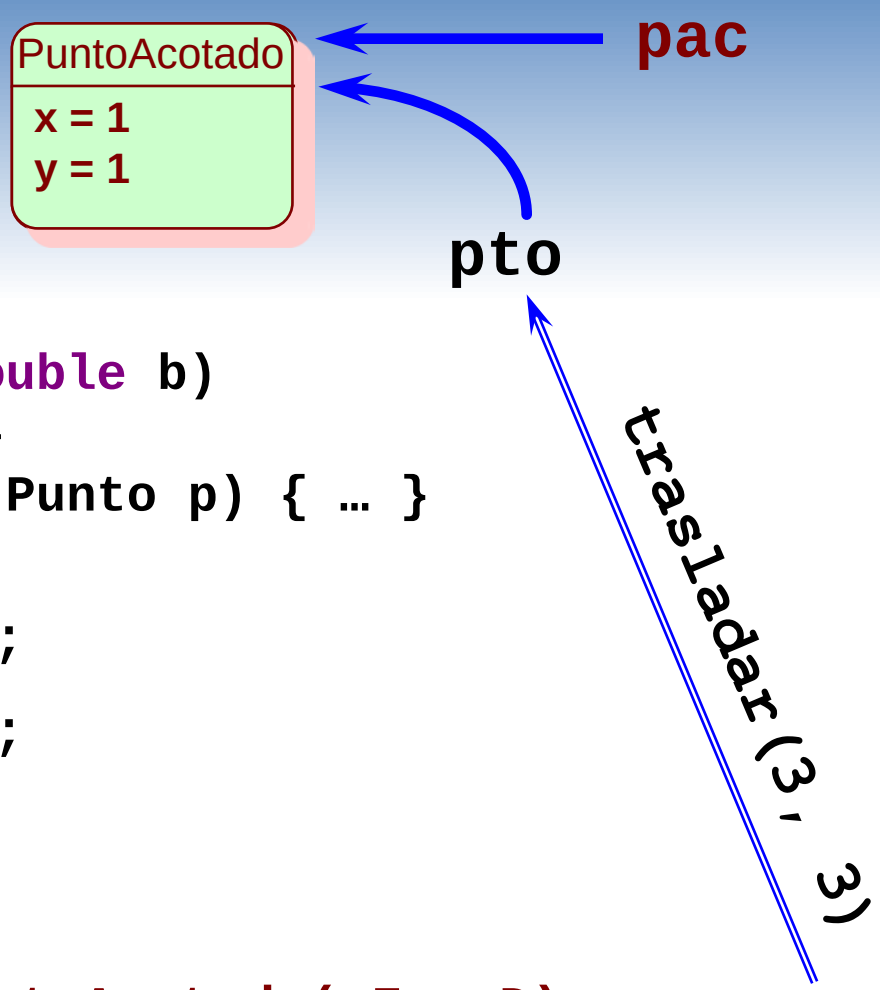


- En general, todos los lenguajes orientados a objetos establecen por defecto un mecanismo de vinculación dinámica para resolver los mensajes.
  - No obstante, algunos de ellos (C++) necesitan etiquetar de forma explícita las funciones que han de resolverse dinámicamente: funciones **virtual**.

```
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) {
     double excesoX, excesoY;
     excesoX = (abscisa()+a-esquinaI.abscisa()) % ancho();
     excesoY = (ordenada()+b-esquinaI.ordenada()) % alto();
     abscisa(excesoX + (excesoX>0 ? esquinaI.abscisa()
                                  : esquinaD.abscisa()));
     ordenada(excesoY + (excesoY>0 ? esquinaI.ordenada()
                                  : esquinaD.ordenada()));
   }
}
```

VINCULACIÓN DINÁMICA

```
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(2,2);


Punto pto;
PuntoAcotado pac = new PuntoAcotado(eI, eD);
pto = pac;
pto.trasladar(3, 3);
```



Clases abstractas

- Clases con funciones sin implementar
  - funciones abstractas en Java
  - funciones virtuales puras en C++
  - rutinas “deferred” en Eiffel
  - métodos “implementedBySubclass” en Smalltalk
- No es posible crear instancias, pero sí declarar variables que puedan referirse a objetos de diversas clases descendientes

punteros a objetos en C++

CLASE ABSTRACTA

```
abstract 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 subclases a implementar los métodos que se declararon como abstractos. De lo contrario, esas subclases se siguen considerando abstractas.
- En Java, además de clases abstractas se pueden definir *interfaces* (que se pueden considerar clases “completamente” abstractas).