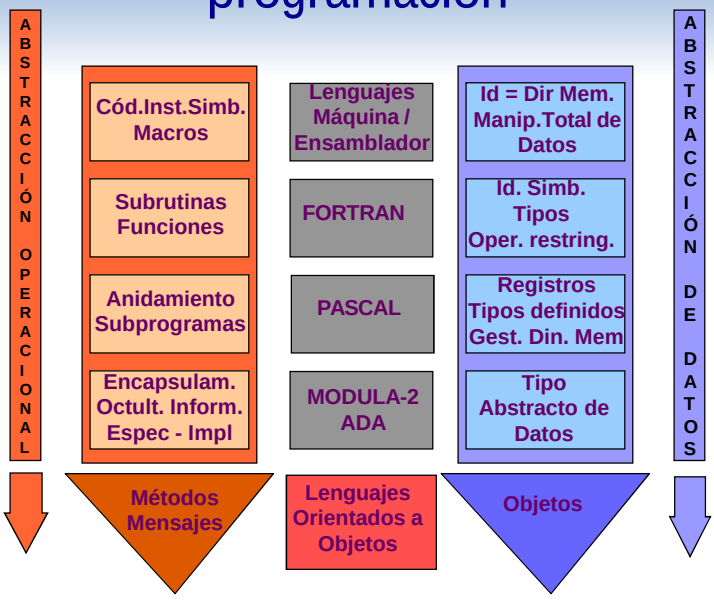
Introducción a la Programación Orientada a Objetos

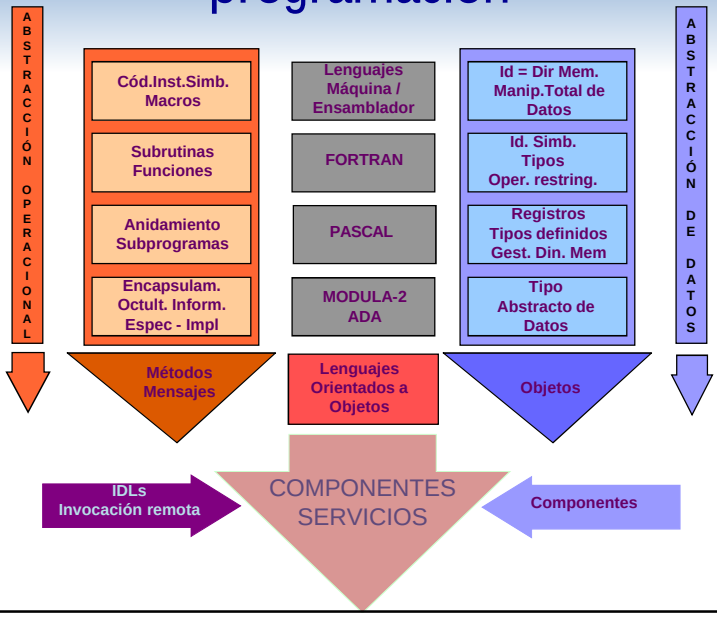
Contenido

- 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
  - Herencia
  - Polimorfismo y vinculación dinámica

Evolución de los lenguajes de programación



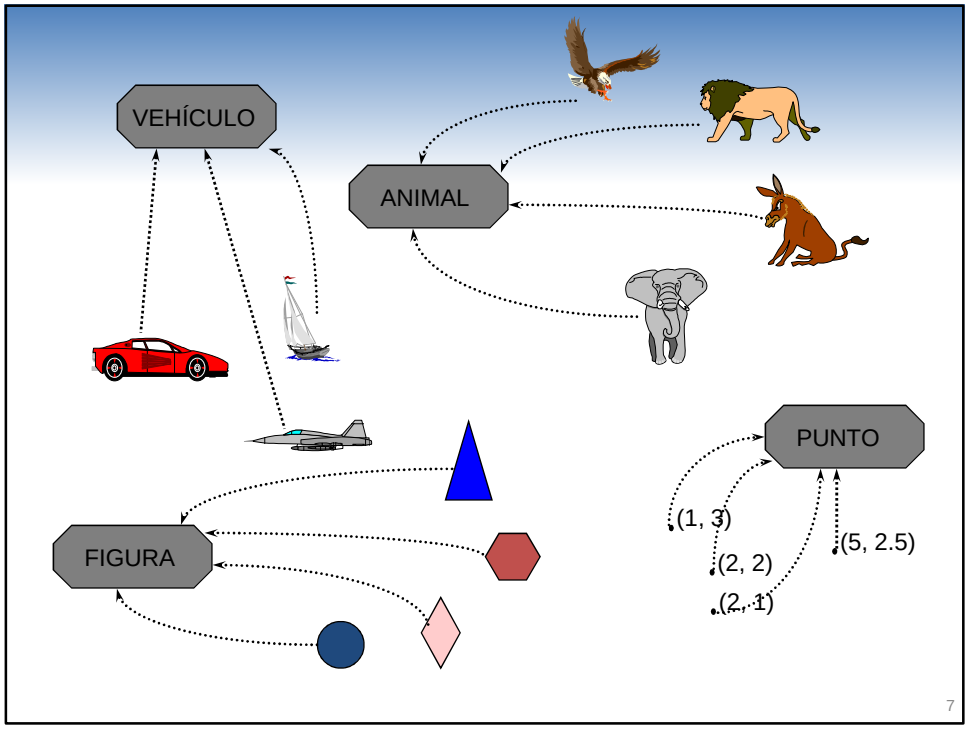
Evolución de los lenguajes de programación



Conceptos básicos de la P.O.O.

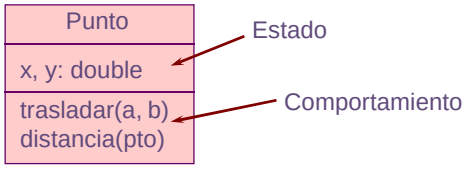
- Clases y Objetos
- Métodos y Mensajes
- Herencia
- Polimorfismo y Vinculación Dinámica

- 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



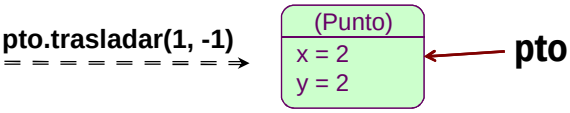
Métodos y Mensajes

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



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

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



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"

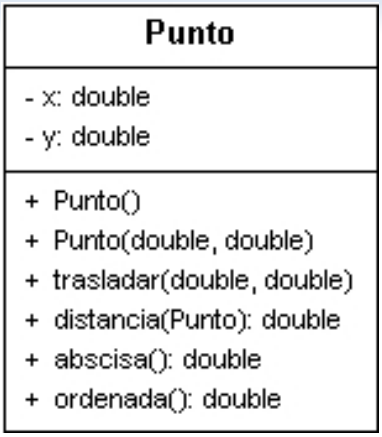
```
class Punto {
  private double x, y;
  public Punto() { x = y = 0; }
  public Punto(double a, double b) { x = a; y = b; }
  public double abscisa() {return x;}
  public double ordenada() {return y;}
  public void abscisa(double a){ x = a; }
  public void ordenada(double b){ y = b; }
  public void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

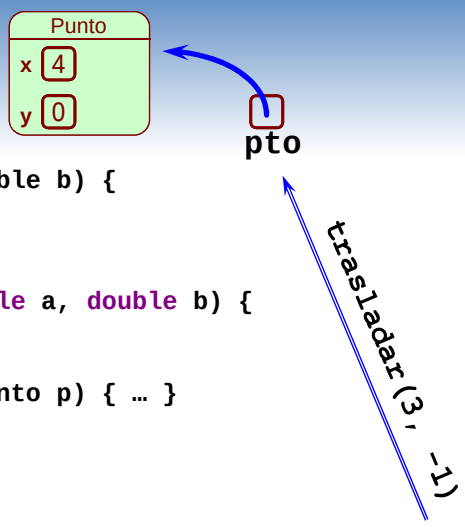
MÉTODOS

Clases



Descripción UML de la clase **`Punto`**

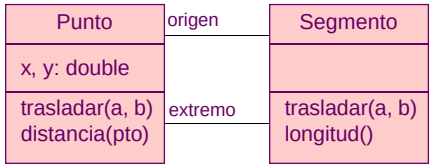

```
class Punto {
   private double x, y;

   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);
```

Composición

- Mecanismo que permite la creación de nuevos objetos a partir de otros ya implementados.
- Responde a una relación de tipo *"tiene"* o "*está compuesto 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



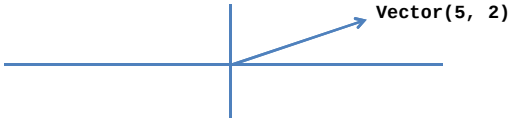
Composición

```
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);
  }
}
```

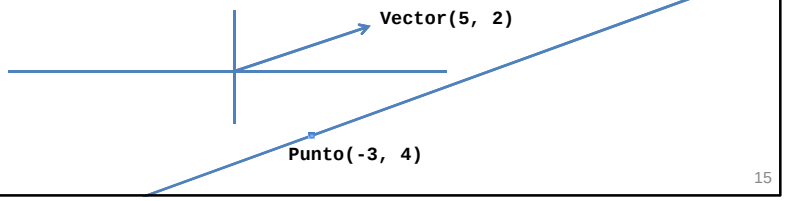
Para calcular la longitud de un segmento se utiliza el método distancia de la clase Punto

Otros ejemplos de composición

- Otros ejemplos:
  - Un vector puede implementarse con un punto que representa el extremo del vector cuando su origen está en el origen de coordenadas.
    - El vector tiene un punto.



  - Una recta podría implementarse con un vector y un punto. El punto es uno cualquiera de la recta y el vector proporciona la dirección de la misma.
    - La recta "tiene" o "está definida por" un punto y un vector

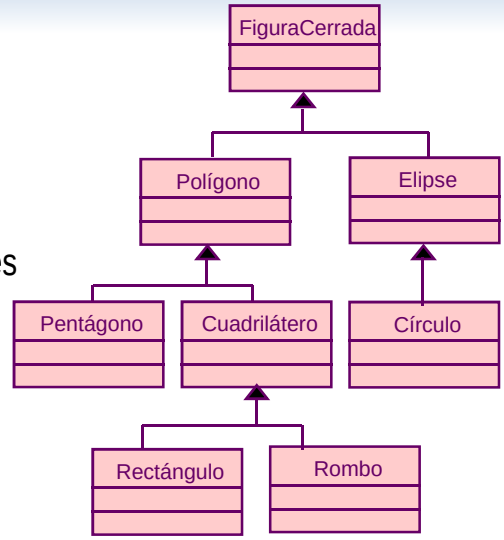


Más ejemplos de composición

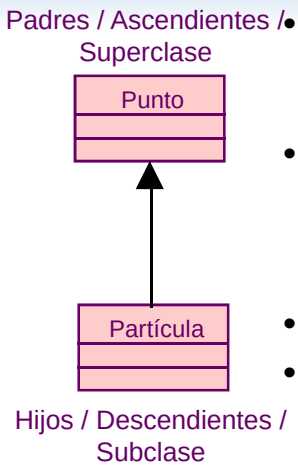
- La composición de una clase dependerá del uso que se haga con sus objetos.
  - Una persona tiene una edad (que se representa con un entero) y un nombre (que se representa con un String).
  - Una persona tiene un nombre (que se representa con un String) y un DNI (que se representa con otro String)
  - Una persona tiene edad, DNI, curso, asignaturas, etc.
  - Un coche tiene un modelo (que se representa por un String) y un precio (que se representa con un float)
  - Un coche está compuesto por un motor, 4 puertas, 4 ruedas, chasis, etc.

Herencia

- Nueva posibilidad para reutilizar código
- Algo más que:
  - incluir ficheros, o
  - importar módulos
- Permite clasificar las clases en una jerarquía
- Responde a la relación "es un"



Herencia



- 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) .
- La herencia es transitiva.
- Los objetos de una clase que hereda de otra pueden verse como objetos de esta última.

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

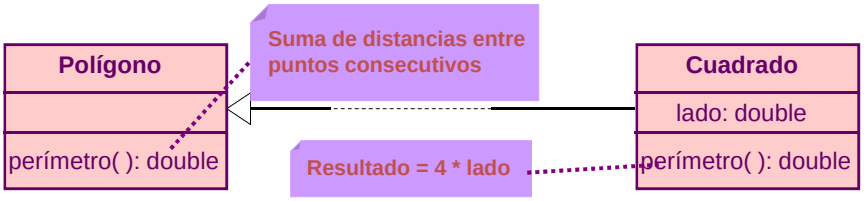

```
class Partícula extends Punto {
   protected double masa;
   final static double G = 6.67e-11;

   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) {
      double d = this.distancia(part);
      return G * masa * part.masa() / (d * d);
   }
}
```

Se refiere a Partícula(double, double, double)

Se refiere a Punto(double, double)

Redefinición del comportamiento

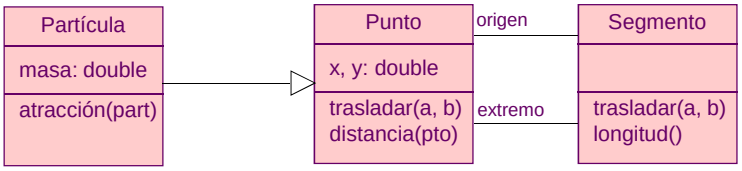
- Es muy corriente la redefinición de un método en la subclase.



- 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 **tiene** dos puntos (origen y extremo)



Polimorfismo sobre los datos

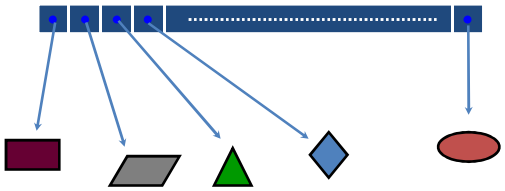
- Un lenguaje tiene capacidad polimórfica sobre los datos cuando
  - una variable declarada de un tipo (o clase) –*tipo estático*– determinado
  - puede hacer referencia en tiempo de ejecución a valores (objetos) de tipo (clase) distinto –*tipo dinámico* –.

- La capacidad polimórfica de un lenguaje no suele ser ilimitada, y en los LOOs está habitualmente restringida por la relación de herencia:
  - El tipo dinámico debe ser descendiente del tipo estático.

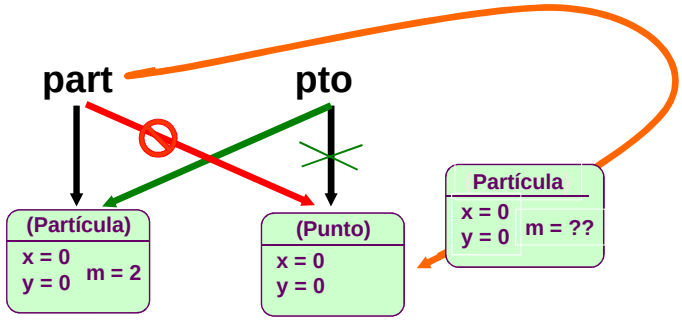
Punto pto = new Partícula(3, 5, 22);

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:

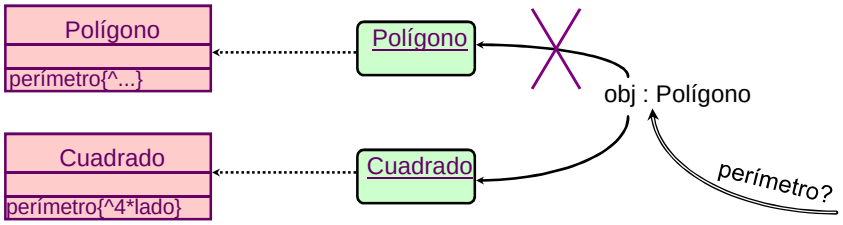


```
Punto pto = new Punto();
Partícula part = new Partícula(2);

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

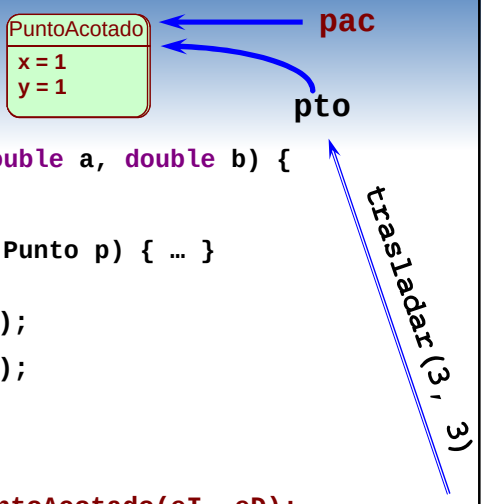


- El compilador admitirá la expresión
  obj.perímetro();
  si el tipo estático de **obj** (es decir la clase **Polígono**) acepta el mensaje **perímetro()**, aunque para resolver utilice vinculación dinámica (es decir, el método **perímetro()** de la clase **Cuadrado**)

```
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())
   }
}
```

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 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 (ya con todos sus métodos implementados)

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).