Tema 1. Introducción a la Programación Orientada a Objetos

Vicente Benjumea García

Programación Orientada a Objetos
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 1. Introducción a la programación orientada a objetos

- Evolución de los lenguajes de programación
- Conceptos fundamentales de la POO:
  - Clases y objetos
  - Metodos, mensajes y atributos
  - Composición
  - Herencia
  - Polimorfismo y vinculación dinámica

Evolución de los lenguajes de programación

- A medida que aumenta la potencia de los ordenadores
- También aumenta la complejidad de los problemas que se resuelven.
- También aumenta la dificultad del diseño y desarrollo de los programas.
- Motiva la creación de nuevos paradigmas de programación que faciliten la creación de programas cada vez más complejos.
- Los lenguajes de programación evolucionan adaptándose a los nuevos entornos y paradigmas.

ALGORITMOS	LENGUAJES	DATOS
Código Máquina	Código Máquina	Dirección Memoria
Instrucciones Simbólicas	Ensamblador	Etiquetas Simbólicas
Subprogramas	Fortran	Tipos y Variables
Programación Estructurada	Algol	Tipos Estructurados
Módulos	Módula-2	Tipos Abstractos Datos
POO y Métodos	Smalltalk	Clases y Objetos

Conceptos fundamentales de la POO

Programación Orientada a Objetos

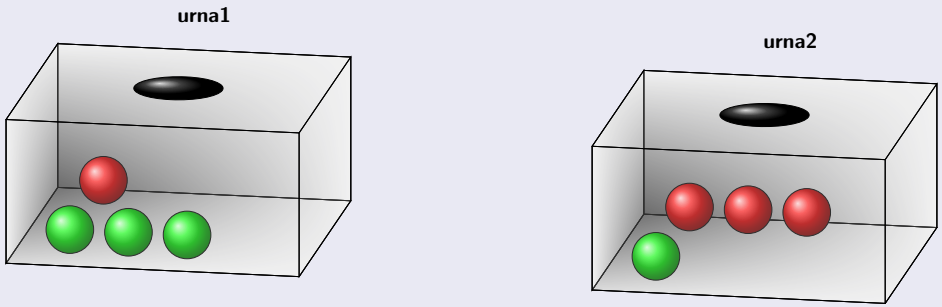
- Es un paradigma de programación que nos permite diseñar programas definiendo **abstracciones** (clases) que modelan los **datos** que representan el problema que queremos resolver.
- Se caracteriza por la definición y creación de **objetos** que encapsulan datos y algoritmos:
  - **Atributos** (datos): almacenan el estado interno del objeto.
  - **Métodos** (algoritmos): definen el comportamiento del objeto. Permiten la manipulación e interacción entre objetos.
  - La **Clase** define las características de los **objetos** que pertenecen a ella.
- La relación de **Herencia** representa una *relación* en la cual una Clase (subclase) **es una** *especialización* o *extensión* de otra Clase (superclase).
  - El **polimorfismo** permite que un objeto de una **subclase** pueda *ser considerado* y *referenciado* como un objeto de la **superclase**. *Principio de sustitución*.
  - La **vinculación dinámica** permite que las subclases puedan redefinir el comportamiento de los métodos definidos en la superclase.
- La relación de **Composición** representa una *relación* en la cual un objeto **tiene** o **está compuesto** por otros objetos.

Ejemplo

Abstracción: Urna

Una *urna* es una *caja* que contiene votos positivos y negativos.

- Comportamiento de la *urna* (**métodos**):
  - Introducir votos positivos y negativos.
  - Calcular el resultado de la votación.
- Estado interno de la *urna* (**atributos**):
  - La cuenta del total de votos positivos dentro de la urna.
  - La cuenta del total de votos negativos dentro de la urna.

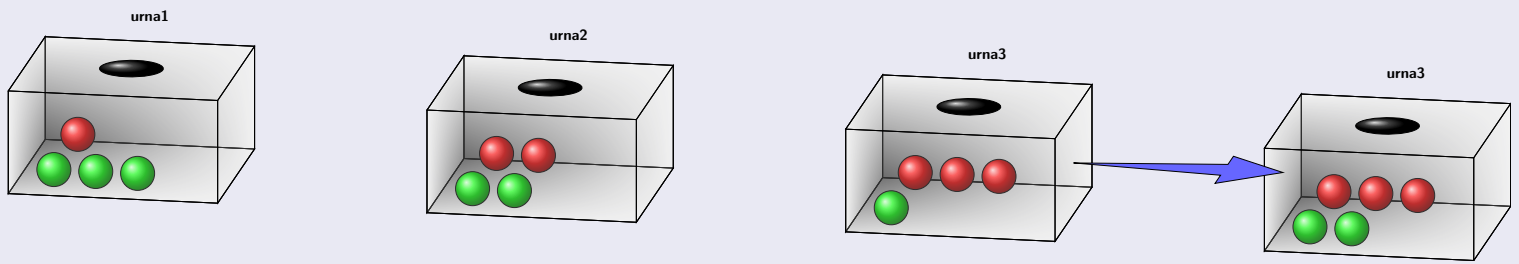


Ejemplo

Abstracción: Urna

Dada la abstracción (**clase**) *Urna* especificada en el ejemplo anterior:

- Podemos crear múltiples *instancias* (**objetos**) de ella.
  - `urna1: Urna(3,1)`, `urna2: Urna(2,2)`, `urna3: Urna(1,3)`
- Cada *instancia* (objeto) de la *Clase Urna* posee su propio estado (**atributos**).
  - La cuenta de votos positivos y votos negativos que contiene.
- Todas las *instancias* de la *Clase Urna* tienen el mismo comportamiento.
- Podemos interactuar con cada objeto, independientemente, a través de los **métodos** que definen su comportamiento.
  - Manipular el estado de cada objeto. Por ejemplo, podemos introducir un *voto positivo* en el objeto `urna3`.
  - Consultar el resultado de la votación. Por ejemplo, si consultamos el resultado de la urna `urna3` después de la operación anterior, se devolverá el valor *falso*.



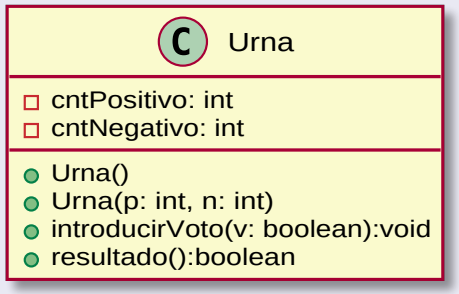
Conceptos fundamentales de la POO

Clase

- Una **Clase** representa una *abstracción de datos*, especifica las **características** de unos **objetos**: su estado y su *comportamiento*.
- Es una **entidad estática**, se define en tiempo de compilación.

Representación Gráfica UML de Clases

- Una clase se representa mediante una *caja* con tres compartimentos, conteniendo cada uno de ellos el nombre, los atributos y los constructores y métodos de la clase.



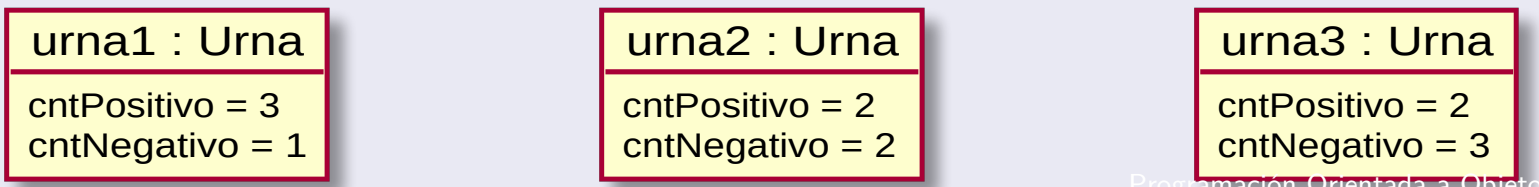
Conceptos fundamentales de la POO

Objeto

- Un **objeto** es una *instancia* (*caso particular concreto*) de una determinada **Clase**.
- Las características del *objeto* (estado y comportamiento) están determinadas por la *Clase* a la que pertenece.
  - Cada *objeto* **almacena** y contiene su propio estado interno (*atributos*), de forma *independiente* de los otros *objetos.*
  - El objeto podrá ser **manipulado** e interactuar con otros objetos a través de los *métodos* definidos por la *Clase* a la que pertenece.
- Es una **entidad dinámica**, se crea y se destruye durante la ejecución del programa.
  - Puede haber muchos objetos **distintos** que sean de la misma *Clase* (y también de distintas *Clases*).

Representación Gráfica UML de Objetos

- Un objeto se representa mediante una *caja* con dos compartimentos, conteniendo el primero el **nombre** del objeto y de la clase a la que pertenece, y el segundo los **valores** concretos de los atributos del objeto.



Conceptos fundamentales de la POO

Métodos

- La clase representa una abstracción de datos, y los métodos definen su comportamiento.
- Los métodos son algoritmos *especiales* definidos por la *Clase*, y se aplican sobre los objetos.
- Manipulan el estado interno del objeto sobre el que se aplican.
- También se les denomina **métodos de instancia**, o **métodos del objeto**.

Invocación a Métodos. Paso de Mensajes

- Se puede invocar a un método, aplicándolo sobre un determinado objeto.
  - Invocar a un método también se denomina como *envíar un mensaje* a un objeto.
- La invocación a métodos puede llevar parámetros asociados, produce un resultado, y manipula el estado interno del objeto sobre el que se aplica.
- Los objetos responden a las invocaciones de los métodos dependiendo de su estado interno.
- Para invocar a un determinado método sobre un objeto, ese método debe estar definido por la clase a la que el objeto pertenece.

Conceptos fundamentales de la POO

Atributos

- Almacenan los valores del estado interno del objeto.
- Cada objeto tiene su propio estado interno asociado, independiente de los otros objetos.
- Los atributos estan **protegidos**. Sólo se permite su acceso y manipulación a través de los métodos.
- También se les denomina **variables de instancia**, o **variables del objeto**.

Ejemplo: Urna con Abstracción Procedimental en C++

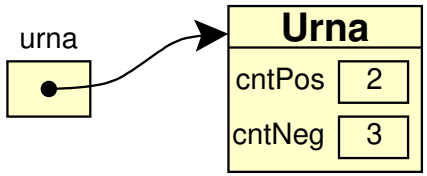
```
struct Urna {          // Estructura que representa una urna de votación
    int cntPositivo;   // cuenta de votos positivos
    int cntNegativo;   // cuenta de votos negativos
};
void inicializar(Urna& urna)
{
    urna.cntPositivo = 0;            // Inicializa la cuenta de positivos
    urna.cntNegativo = 0;            // Inicializa la cuenta de negativos
}
void introducirVoto(Urna& urna, bool voto) // Añade un voto positivo o negativo
{
    if (voto) {
        ++urna.cntPositivo;            // Incrementa la cuenta de positivos
    } else {
        ++urna.cntNegativo;            // Incrementa la cuenta de negativos
    }
}
bool resultado(const Urna& urna)   // Devuelve el resultado de la votación
{
    return (urna.cntPositivo >= urna.cntNegativo);
}
```

Ejemplo: Clase Urna

```java
// Urna.java
package votacion;                    // Paquete donde se define la clase

public class Urna {                  // Abstracción sobre una urna de votación
    //--Atributos-------------------
    private int cntPositivo;         // Estado interno privado de cada objeto
    private int cntNegativo;         // Estado interno privado de cada objeto
    //--Constructores---------------
    public Urna() {                  // Constructor del objeto
        cntPositivo = 0;             // Inicializa la cuenta de positivos
        cntNegativo = 0;             // Inicializa la cuenta de negativos
    }
    //--Métodos---------------------
    public void introducirVoto(boolean voto) { // Añade un voto positivo o negativo
        if (voto) {
            ++cntPositivo;             // Incrementa la cuenta de positivos
        } else {
            ++cntNegativo;             // Incrementa la cuenta de negativos
        }
    }
    public boolean resultado() {   // Devuelve el resultado de la votación
        return (cntPositivo >= cntNegativo);
    }
}
```

Ejemplo: Clase PruebaUrna

```java
// PruebaUrna.java
import votacion.Urna;        // Utilización de la clase Urna del paquete votacion
import java.util.Scanner;    // Utilización de la clase Scanner del paquete java.util

public class PruebaUrna {
    public static void main(String[] args) {     // Principal
        Urna urna = new Urna();                   // Creación de un objeto Urna
        votacion(urna);
        System.out.println("resultado final: " + urna.resultado());
    }
    private static void votacion(Urna urna) {    // añade votos a una Urna
        urna.introducirVoto(true);
        urna.introducirVoto(true);
        urna.introducirVoto(false);
        System.out.println("res. intermedio: " + urna.resultado()); // true
        urna.introducirVoto(false);
        urna.introducirVoto(false);
    }
    private static void votacionTeclado(Urna urna) {     // añade votos leídos de teclado
        System.out.println("Introduce votos (true, false) hasta FIN");
        Scanner teclado = new Scanner(System.in);    // la mayoría de recursos deben cerrarse, pero
        String voto;                                 // System.in es un recurso que no debe cerrarse
        voto = teclado.next();
        while ( ! voto.equalsIgnoreCase("FIN") ) {
            urna.introducirVoto(Boolean.valueOf(voto));
            voto = teclado.next();
        }
    }
}
```

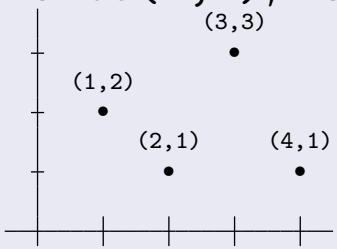


Ejemplo

Abstracción: Punto del plano Cartesiano

Un *punto* representa una determinada posición en el plano Cartesiano.

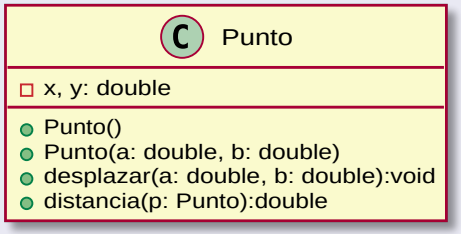
- Comportamiento de un *punto*:
  - Especificar el valor de sus coordenadas X e Y.
  - Consultar el valor de sus coordenadas X e Y.
  - Calcular la distancia que lo separa de otro objeto *punto*.
  - Desplazar según una distancia especificada en ambos ejes.
- Estado interno del *punto*:
  - El valor de la coordenada X (abscisa).
  - El valor de la coordenada Y (ordenada).
- Podemos crear múltiples objetos de la *Clase Punto*:
  - `Punto(1,2)`, `Punto(2,1)`, `Punto(3,3)`, `Punto(4,1)`



Representación Gráfica UML de Clases y Objetos

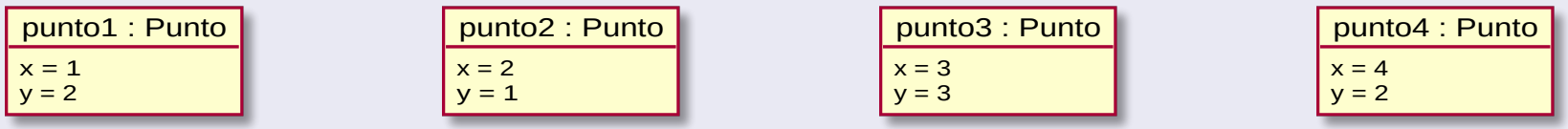
Representación Gráfica UML de Clases

- Una clase se representa mediante una *caja* con tres compartimentos, conteniendo cada uno de ellos el nombre, los atributos y los métodos de la clase.



Representación Gráfica UML de Objetos

- Un objeto se representa mediante una *caja* con dos compartimentos, conteniendo el primero el **nombre** del objeto y de la clase a la que pertenece, y el segundo los **valores** de los atributos del objeto.

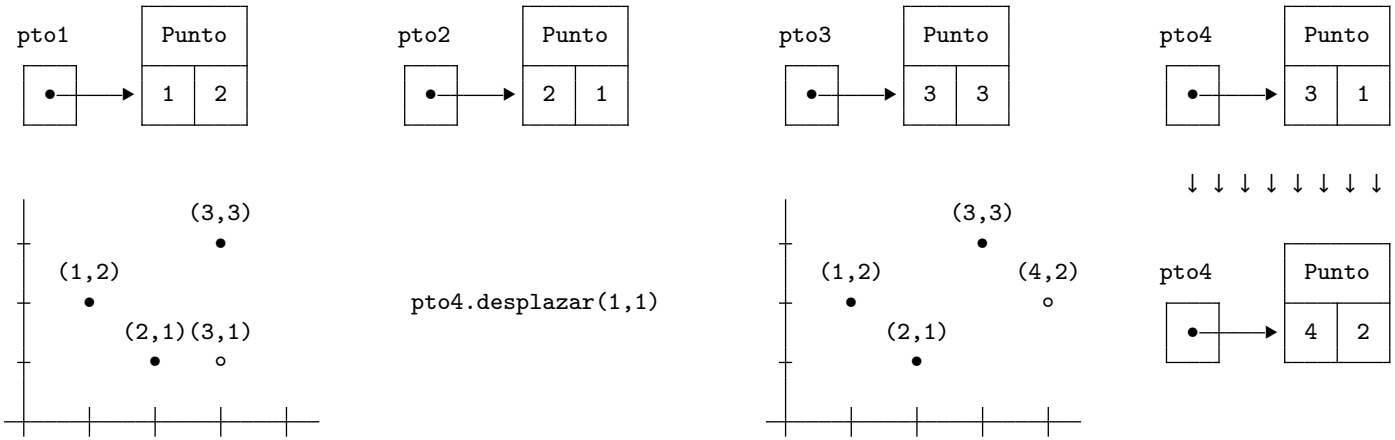


Ejemplo: Clase Punto

```
package geometria;                  // paquete al que pertenece la clase
public class Punto {                // fichero Punto.java
    private double x, y;                // Atributos
    public Punto() {                    // Constructores
        x = 0;          // this(0, 0);
        y = 0;
    }
    public Punto(double a, double b) {
        x = a;
        y = b;
    }
    public double getAbscisa() {    // Métodos
        return x;
    }
    public double getOrdenada() {
        return y;
    }
    public void setAbscisa(double a) {
        x = a;
    }
    public void setOrdenada(double b) {
        y = b;
    }
    public void desplazar(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));
    }
}
```

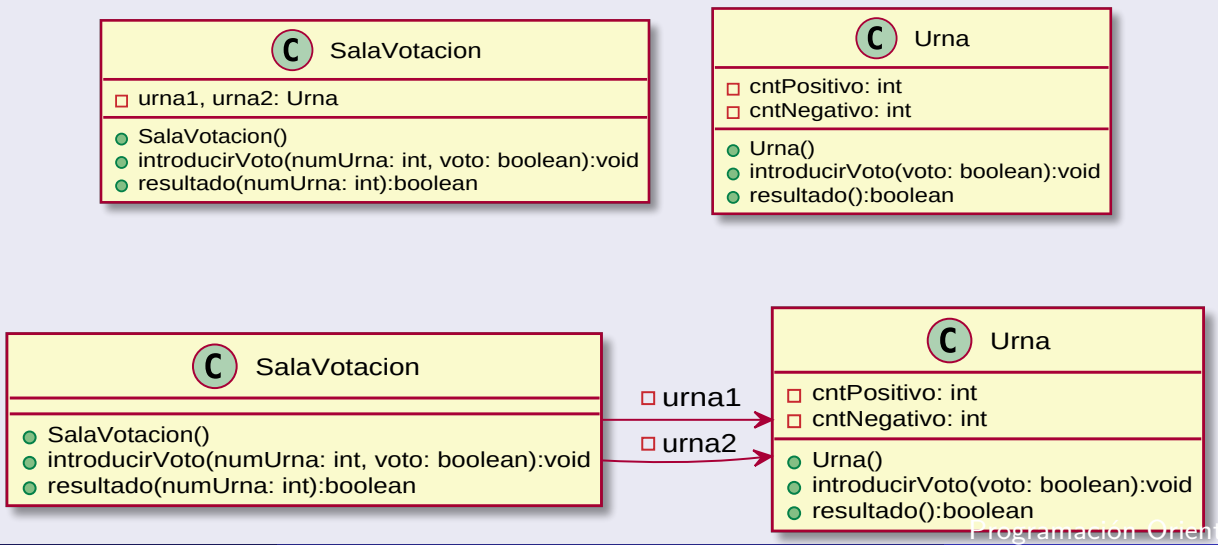
Ejemplo: Clase PruebaPunto

```java
import geometria.Punto;
public class PruebaPunto {
  public static void main(String[] args) {
    Punto pto1 = new Punto(1, 2);
    Punto pto2 = new Punto(2, 1);
    Punto pto3 = new Punto(3, 3);
    Punto pto4 = new Punto(3, 1);
    pto4.desplazar(1, 1);
    double d = pto1.distancia(pto3);
  }
}
```



Composición

- Permite la definición de *nuevas clases* a partir de otras clases ya definidas.
- Representa una *relación* en la cual un objeto **tiene** o **está compuesto** por otros objetos.
  - Por ejemplo, una `SalaVotacion` **tiene** dos `Urnas`, la *urna1* y la *urna2*.
  - El objeto *contenido* forma parte de los **atributos** del objeto *contenedor*.
  - La composición se puede expresar en UML mediante una línea continua con flecha de **punta abierta** entre la *clase poseedora* y la *clase poseída*.
  - Esta línea continua con flecha de punta abierta se suele etiquetar con el nombre del atributo, su nivel de acceso, y opcionalmente su multiplicidad (donde el asterisco indica múltiples elementos).

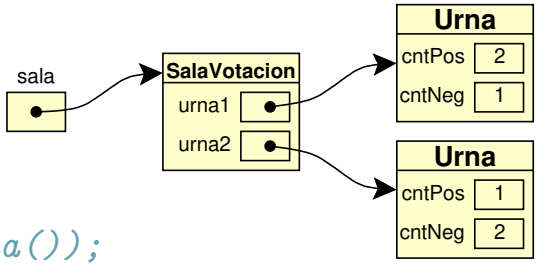


Ejemplo: Clase SalaVotacion

```java
// SalaVotacion.java
package votacion;                       // Paquete donde se define la clase
public class SalaVotacion {             // Abstracción sobre una sala de votación
    private Urna urna1;
    private Urna urna2;
    public SalaVotacion() {                 public SalaVotacion(Urna u1, Urna u2) {
        urna1 = new Urna();                     urna1 = u1;
        urna2 = new Urna();                     urna2 = u2;
    }                                       }
    public void introducirVoto(int numUrna, boolean voto) { // Añade un voto positivo o negativo a urna
        if (numUrna == 1) {
            urna1.introducirVoto(voto);       // Añade un voto a la primera urna
        } else if (numUrna == 2) {
            urna2.introducirVoto(voto);       // Añade un voto a la segunda urna
        } else {
            throw new RuntimeException("Número de urna erróneo " + numUrna);  // Notifica del Error
        }
    }
    public boolean resultado(int numUrna) {  // Devuelve el resultado de la votación en una urna
        boolean res;
        switch (numUrna) {
        case 1:  res = urna1.resultado(); break; // modo compacto por restricciones de espacio
        case 2:  res = urna2.resultado(); break; // modo compacto por restricciones de espacio
        default: throw new RuntimeException("Número de urna erróneo " + numUrna);  // Notifica del Error
        }
        return res;
    }
}
```

Ejemplo: Clase PruebaSalaVotacion

```java
// PruebaSalaVotacion.java
import votacion.SalaVotacion;

public class PruebaSalaVotacion {
    public static void main(String[] args) {
        //SalaVotacion sala = new SalaVotacion(new Urna(), new Urna());
        //SalaVotacion sala = new SalaVotacion(new Urna(), new UrnaOpaca());
        SalaVotacion sala = new SalaVotacion();
        votacion(sala);
        System.out.println("Resultado final 1: " + sala.resultado(1));
        System.out.println("Resultado final 2: " + sala.resultado(2));
    }
    private static void votacion(SalaVotacion sala) {
        sala.introducirVoto(1, false);
        System.out.println("Resultado intermedio 1: " + sala.resultado(1));
        sala.introducirVoto(1, true);
        sala.introducirVoto(1, true);
        sala.introducirVoto(2, true);
        System.out.println("Resultado intermedio 2: " + sala.resultado(2));
        sala.introducirVoto(2, false);
        sala.introducirVoto(2, false);
    }
}
```

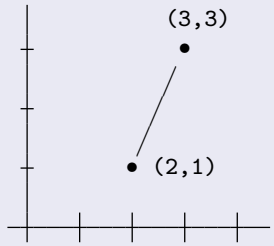


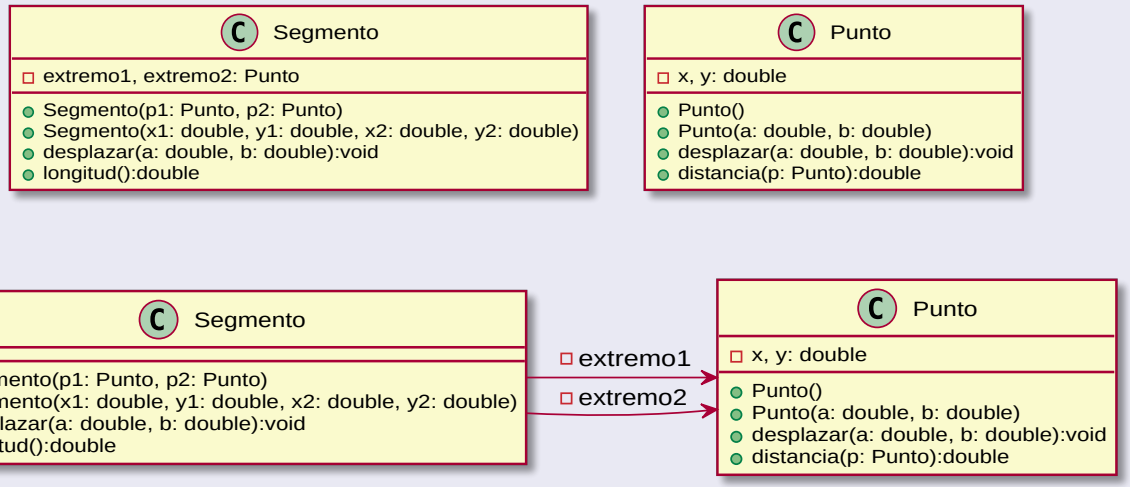
Ejemplo: Clase Segmento

Abstracción: Segmento del plano Cartesiano

Un *segmento* es un fragmento de una recta que está comprendido entre dos *puntos* en el plano Cartesiano.

- Comportamiento de un *segmento*:
  - Especificar el valor de las coordenadas X e Y de ambos puntos.
  - Calcular la longitud del segmento.
  - Desplazar el segmento según una distancia especificada en ambos ejes.
- Estado interno del *segmento*:
  - Ambos puntos (extremos) que delimitan el segmento.
- Podemos crear múltiples objetos de la *Clase Segmento*: `Segmento(2, 1, 3, 3)`



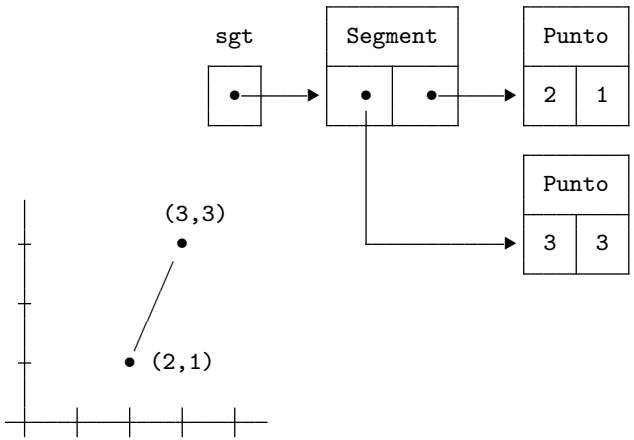


Ejemplo: Clase Segmento

```
package geometria;                  // paquete al que pertenece la clase

public class Segmento {
  // Atributos
  private Punto extremo1, extremo2;
  // Constructores
  public Segmento(Punto p1, Punto p2) {
    extremo1 = p1;
    extremo2 = p2;
  }
  public Segmento(double x1, double y1, double x2, double y2) {
    extremo1 = new Punto(x1, y1);  // creación de objetos de la clase Punto
    extremo2 = new Punto(x2, y2);  // creación de objetos de la clase Punto
  }
  // Métodos
  public void desplazar(double a, double b) {
    extremo1.desplazar(a, b);   // invocación a métodos de la clase Punto
    extremo2.desplazar(a, b);   // invocación a métodos de la clase Punto
  }
  public double longitud() {
    return extremo1.distancia(extremo2); // invocación a métodos de la clase Punto
  }
}
```

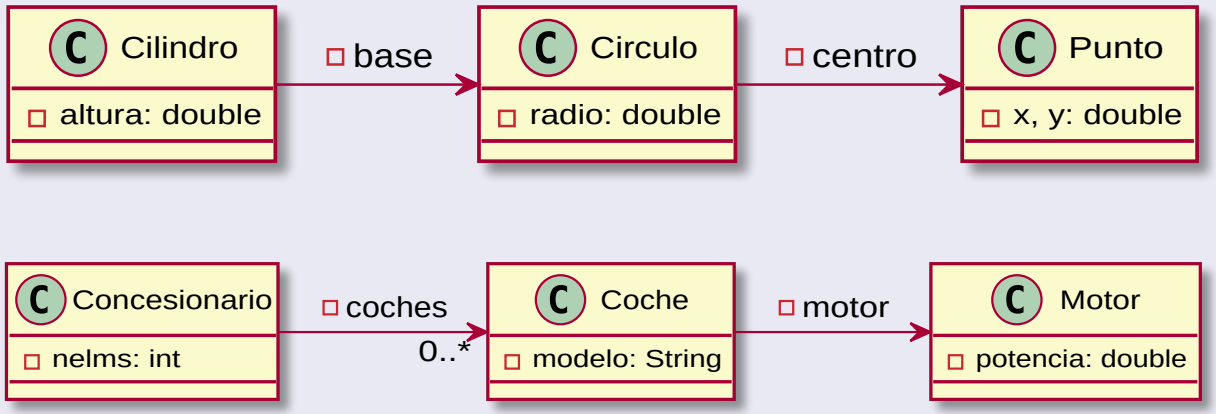
Ejemplo: Clase PruebaSegmento

```java
// PruebaSegmento.java
import geometria.Segmento;
public class PruebaSegmento {
  public static void main(String[] args) {
    Segmento sgt = new Segmento(2, 1, 3, 3);
    double lng = sgt.longitud();
  }
}
```



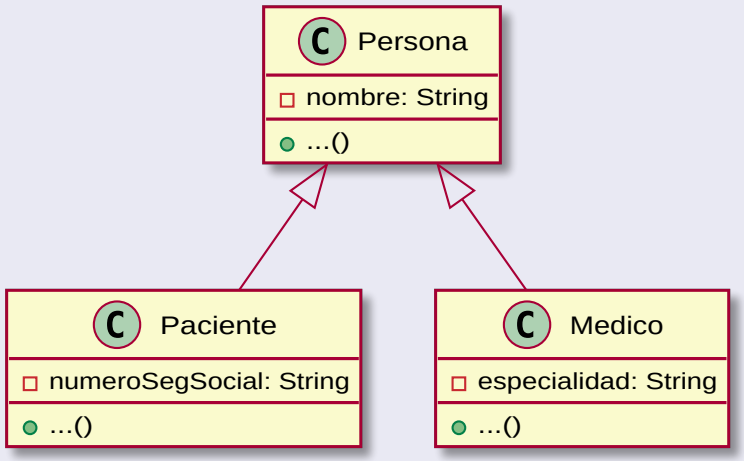
Composición

- Otros ejemplos de composición.
  - El objeto *contenido* forma parte de los **atributos** del objeto *contenedor*.
  - La composición se puede expresar en UML mediante una línea continua con flecha de **punta abierta** entre la *clase poseedora* y la *clase poseída*.
  - Esta línea continua con flecha de punta abierta se suele etiquetar con el nombre del atributo, su nivel de acceso, y opcionalmente su multiplicidad (donde el asterisco indica múltiples elementos).



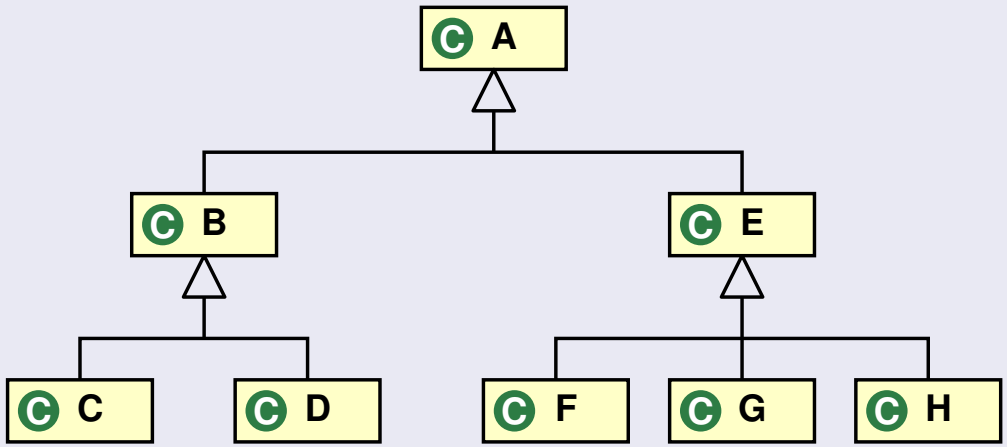
Herencia

- Representa una *relación* en la cual una Clase **es una** *especialización* o *extensión* de otra Clase.
- Permite definir una nueva **subclase** (o clase derivada) como una especialización o extensión de una **superclase** (o clase base) más general.
  - La *subclase* **hereda** tanto los **atributos** como los **métodos** definidos por la *superclase* (reusabilidad del código).
  - La subclase puede **añadir** *nuevos atributos* y *nuevos métodos* (extensibilidad), así como **redefinir** métodos de la superclase (especialización).
- La herencia se expresa en UML mediante una línea continua desde la *subclase* con un **triángulo hueco** en el extremo de la *superclase*.
- Por ejemplo, un `Paciente` **es una** `Persona`, y un `Médico` también **es una** `Persona`.



Herencia

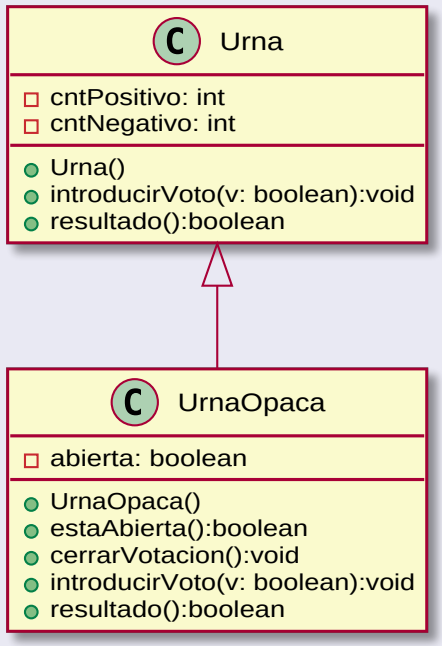
- Representa una *relación* en la cual una Clase **es una** *especialización* o *extensión* de otra Clase.
- Permite definir una nueva **subclase** (o clase derivada) como una especialización o extensión de una **superclase** (o clase base) más general.
  - La *subclase* **hereda** tanto los **atributos** como los **métodos** definidos por la *superclase* (reusabilidad del código).
  - La subclase puede **añadir** *nuevos atributos* y *nuevos métodos* (extensibilidad), así como **redefinir** métodos de la superclase (especialización).
- Permite definir **jerarquías** de clases (ascendientes, y descendientes).
- La relación de herencia es **transitiva**, si C hereda de B y B hereda de A, entonces C también hereda de A.



Ejemplo: Clase UrnaOpaca

Ejemplo: Abstracción urna opaca

- Una `UrnaOpaca` **es una** `Urna` en la cual sólo se puede consultar el resultado al final de la votación.
  - Por lo tanto, se podrán introducir votos mientras la votación está *abierta*, pero cuando se consulte el resultado, se cerrará la votación y no se permitirá introducir nuevos votos.
  - La `UrnaOpaca` hereda las características de `Urna` (atributos y métodos), y la extiende añadiendo un nuevo atributo, el estado de la votación (si la votación está abierta o cerrada). Además, permite consultar el estado de la votación, así como cerrar la votación. Así mismo, la `UrnaOpaca` también especializa la `Urna`, redefiniendo su comportamiento para evitar que se introduzcan votos cuando la votación está cerrada.



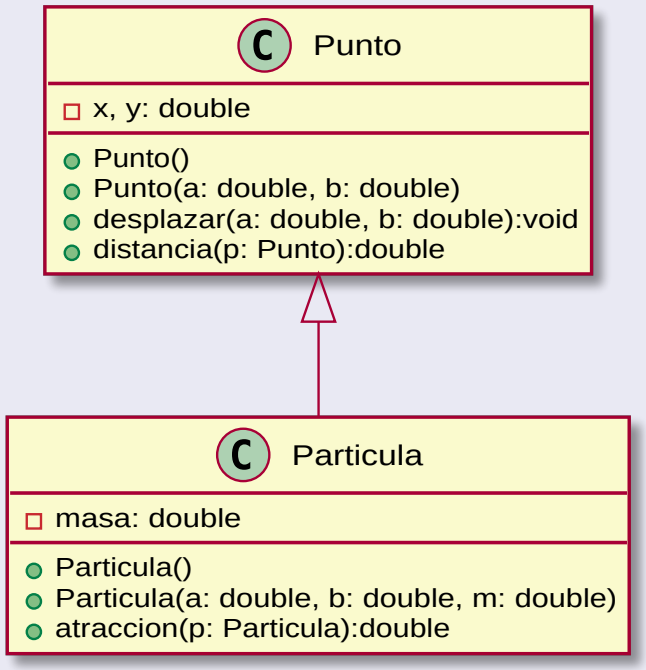
Ejemplo: Clase UrnaOpaca

```java
// UrnaOpaca.java
package votacion;                       // Paquete donde se define la clase

public class UrnaOpaca extends Urna { // Abstracción sobre una urna de votación
    private boolean abierta;            // Estado interno privado de cada objeto
    public UrnaOpaca() {                // Constructor del objeto
        super();                        // Inicializa el objeto-base de la superclase
        abierta = true;                 // Inicializa la urna como abierta
    }
    public boolean estaAbierta() {      // Devuelve el estado de la urna
        return abierta;
    }
    public void cerrarVotacion() {      // Cierra el estado de la urna
        abierta = false;                // Cierra la urna
    }
    @Override                           // Redefine el comportamiento del método
    public void introducirVoto(boolean voto) {  // Añade un voto positivo o negativo
        if (estaAbierta()) {              // Si está abierta, entonces se puede votar
            super.introducirVoto(voto); // Registra un voto positivo o negativo
        }                                 // En otro caso, se desecha el voto
    }
    @Override                           // Redefine el comportamiento del método
    public boolean resultado() {        // Devuelve el resultado de la votación
        this.cerrarVotacion();          // Cierra la urna
        return super.resultado();       // Devuelve el resultado de la votación
    }
}
```

Ejemplo: Clase Partícula

Ejemplo: Abstracción partícula en el plano Cartesiano

- Una `Particula` **es un** `Punto` con **masa**.
  - La `Particula` hereda las características de `Punto` (atributos y métodos), y la extiende añadiendo un nuevo atributo, la masa de la partícula. Además, las partículas tienen la capacidad de *atraerse* entre ellas.

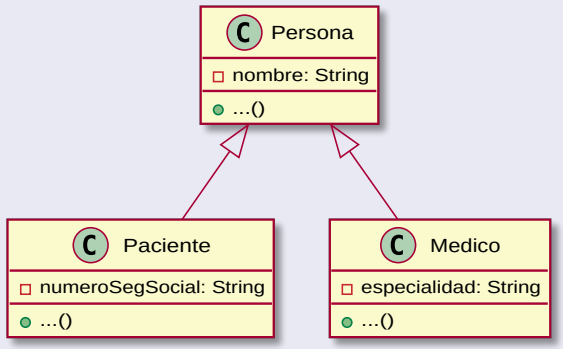


Ejemplo: Clase Particula

```java
package geometria;                  // paquete al que pertenece la clase

public class Particula extends Punto {
    // Atributos
    private final static double G = 6.67408e-11;
    private double masa; // ( + los atributos heredados de Punto )
    // Constructores
    public Particula(double m) {
        this(0, 0, m); // Se refiere a Partícula(double, double, double)
    }
    public Particula(double a, double b, double m) {
        super(a, b); // Se refiere a Punto(double, double)
        masa = m;
    }
    // Métodos ( + los métodos heredados de Punto )
    public void setMasa(double m) {
        masa = m;
    }
    public double getMasa() {
        return masa;
    }
    public double atraccion(Particula part) {
        // Nótese la invocación al método distancia heredado de Punto
        return G * this.masa * part.masa / Math.pow(this.distancia(part), 2);
    }
}
```

Polimorfismo

- Un lenguaje tiene capacidad polimórfica cuando una variable declarada de un determinado tipo (clase) (**tipo estático**) puede referenciar en **tiempo de ejecución** a valores (objetos) de tipo (clase) distinto (**tipo dinámico**).
- En POO el *polimorfismo* está restringido a la relación de herencia.
  - El tipo dinámico debe ser descendiente del tipo estático.
- El **polimorfismo** permite que un objeto de una **subclase** pueda *ser considerado* y *referenciado* como un objeto de la **superclase**. **Principio de sustitución**.
  - La dirección de correspondencia opuesta **no** se mantiene:
    Todos los Medicos son Personas, pero no todas las Personas son Médicos.
- El **polimorfismo** afecta a:
  - Asignaciones.
  - Paso de parámetros.
  - Devolución del resultado en una función.




```
Persona p1 = new Persona("María");
Persona p2 = new Medico("Juan", "Digestivo");
Persona p3 = new Paciente("Pepe", "1223334");

Urna u1 = new Urna();
Urna u2 = new UrnaOpaca();

Punto pt1 = new Punto(3, 5);
double d = pt1.distancia(new Particula(5, 7, 22));
```

Polimorfismo

- En *contextos polimórficos*, sólo es válido invocar a los métodos especificados por el *tipo estático* de la variable.

```
Urna urna1 = new Urna();
Urna urna2 = new UrnaOpaca();
urna1.introducirVoto(true);
urna2.introducirVoto(false);
// urna1.cerrarVotacion(); // ERROR
// urna2.cerrarVotacion(); // ERROR
```

```
Punto pt1 = new Particula(3, 5, 22);
Punto pt2 = new Particula(4, 6, 30);
double d = pt1.distancia(pt2);
// double f = pt1.atraccion(pt2); // ERROR
```

Vinculación Dinámica

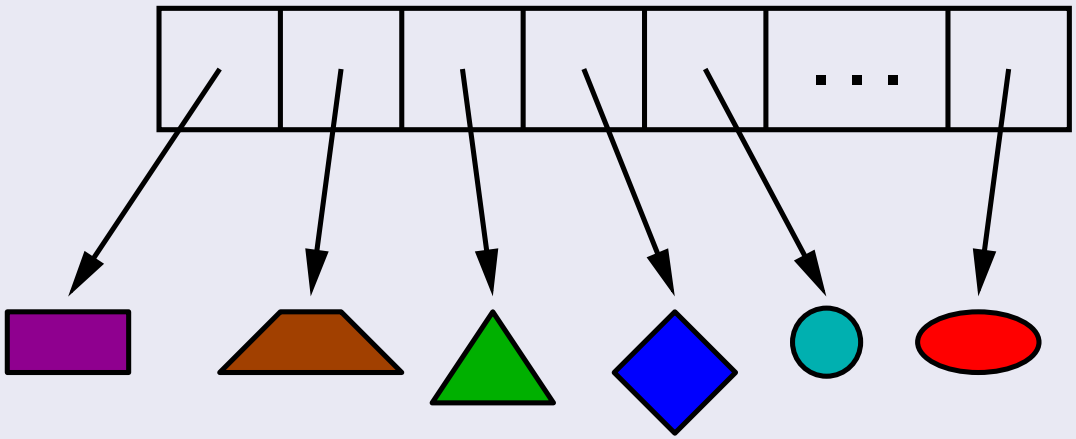
- La **vinculación dinámica** permite que las subclases puedan redefinir el comportamiento de los métodos definidos en la superclase.
  - En *contextos polimórficos*, los métodos invocados se seleccionan adecuadamente, en tiempo de ejecución, dependiendo del *tipo dinámico* del objeto, y no de su *tipo estático*.
    - La invocación del método que ha de resolver un mensaje se retrasa al tiempo de ejecución, y depende del tipo dinámico del objeto.
  - Se puede impedir que las subclases redefinan un determinado método especificando el calificador `final` en su definición.

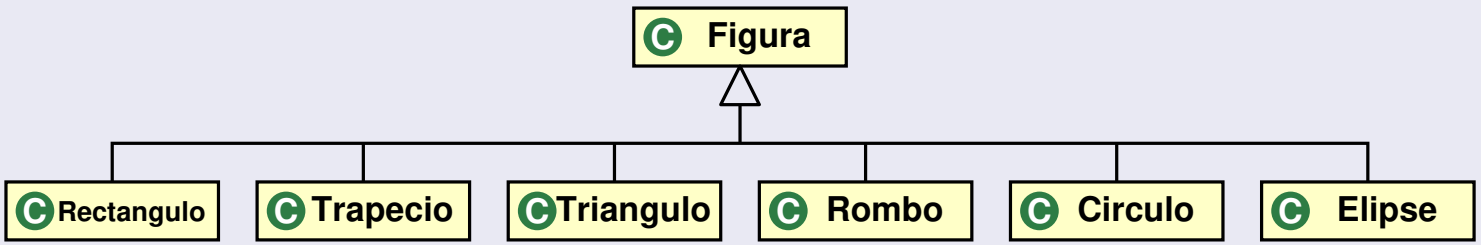
Ejemplo: Clase PruebaUrnaOpaca

```java
// PruebaUrnaOpaca.java
import votacion.*;      // Utilización de las clases del paquete votacion

public class PruebaUrnaOpaca {
    public static void main(String[] args) {
        Urna urna = new Urna();
        votacion(urna);
        System.out.println("res. final: " + urna.resultado()); // false

        UrnaOpaca urnaOpaca = new UrnaOpaca();
        votacion(urnaOpaca);
        System.out.println("estado:     " + urnaOpaca.estaAbierta()); // false
        System.out.println("res. final: " + urnaOpaca.resultado());   // true
    }
    private static void votacion(Urna urna) {
        urna.introducirVoto(true);
        urna.introducirVoto(true);
        urna.introducirVoto(true);
        urna.introducirVoto(false);
        urna.introducirVoto(false);
        System.out.println("res. intermedio: " + urna.resultado()); // true
        urna.introducirVoto(false);             // si es UrnaOpaca, este voto se desecha
        urna.introducirVoto(false);             // si es UrnaOpaca, este voto se desecha
        // System.out.println("estado: " + urna.estaAbierta());  // ERROR. No es posible
    }
}
```

Herencia, Polimorfismo y Vinculación Dinámica

- Gracias a la herencia, el polimorfismo, y la vinculación dinamica, se pueden construir estructuras con elementos dinámicos de distinta naturaleza, pero con un **comportamiento común**.





Símbolos en Eclipse para Diagramas de Clases UML