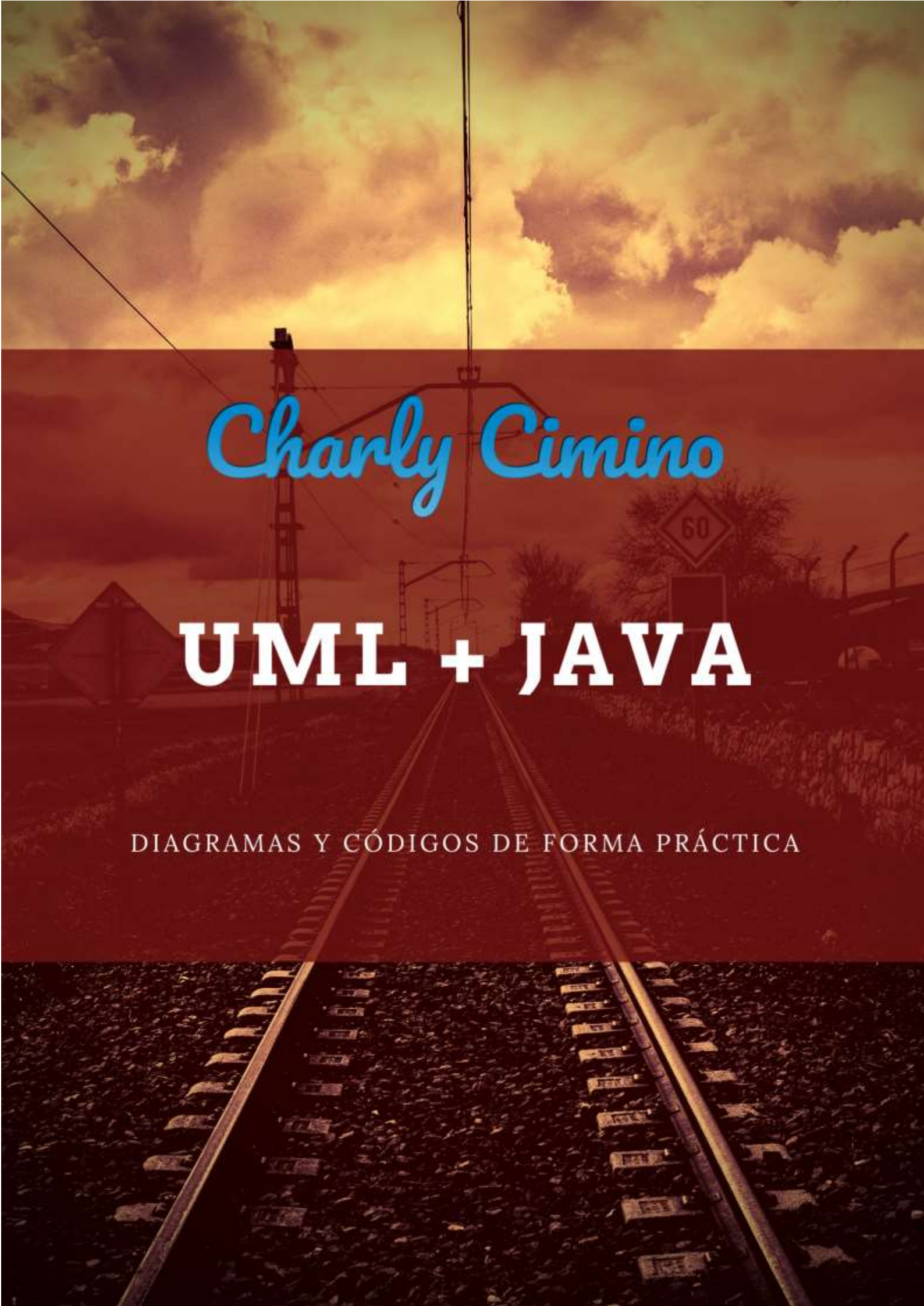

Charly Cimino
UML + JAVA
DIAGRAMAS Y CÓDIGOS DE FORMA PRÁCTICA

UML + Java

Charly Cimino

Versión 2023-01-19

Tabla de contenido

Introducción

Desarrollar software es mucho más que escribir código en un determinado lenguaje. A medida que escala una aplicación, es necesario contar con una documentación acorde que permita amenizar este proceso. Por ello nació UML, que permite graficar determinado sistema de forma general, sin depender de un lenguaje de programación.

Este apunte no pretender explicar todos los detalles y posibilidades que ofrece UML. Tampoco pretender ser una enciclopedia de Java. Ya existe variado material en la red sobre cada uno de éstos.

El objetivo entonces es enfocar las explicaciones a un carácter práctico, que permita hacer un paralelismo entre los conceptos del paradigma de programación orientada a objetos y el lenguaje Java, con ejemplo típicos de código y su correspondiente representación gráfica.

¿Qué es UML?

UML significa, en inglés, *"Unified Modeling Language"*, cuya traducción es **lenguaje unificado de modelado**. Se trata de un lenguaje gráfico estandarizado, que se utiliza para modelar software desde distintas perspectivas, por eso existen 13 tipos de diagramas diferentes, que se pueden enmarcar en dos grandes categorías: de estructura y de comportamiento.

Diagramas de estructura

Este tipo de diagramas son atemporales, es decir, grafican la estructura estática de un sistema, sin información sobre el ciclo de vida de sus componentes. A continuación, se listan los diagramas de estructura más comunes:

- **Diagrama de clases**: Muestra las clases intervinientes en un sistema y sus correspondientes relaciones.
- **Diagrama de paquetes**: Muestra los paquetes intervinientes en un sistema y sus correspondientes relaciones.
- **Diagrama de objetos**: Muestra el estado de los objetos intervinientes en un sistema en un determinado instante.

Diagramas de comportamiento

Este tipo de diagramas son temporales, es decir, grafican la estructura dinámica de un sistema, la cual sufre cambios de estado o de componentes en el tiempo. A continuación, se listan los diagramas de comportamiento más comunes:

- **Diagrama de casos de uso**: Muestra los diferentes actores intervinientes en un sistema junto a la funcionalidad requerida y sus interacciones.
- **Diagrama de secuencia**: Muestra la interacción en orden de los diferentes componentes de un sistema.

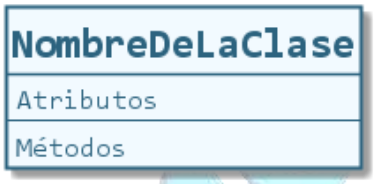
Por el momento, este apunta se va a centrar exclusivamente en los diagramas de clase UML, de los cuales hablaremos a continuación.

Diagrama de clases

Definición de una clase

Una clase se dibuja como un rectángulo dividido en tres compartimentos horizontales:

- Superior: Lleva el nombre de la clase.
- Central: Lleva la lista de atributos.
- Inferior: Lleva la lista de constructores y métodos.



Supongamos querer diagramar personas, las cuales poseen nombre, apellido y DNI y son capaces de saludar, devolver su nombre completo y cambiar de nombre. Veamos paso a paso como definir la clase `Persona`.

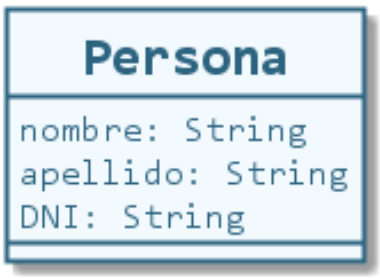
Atributos

Se inserta en cada línea el nombre del atributo seguido de un `:` y el tipo de dato.

En este caso, todas las personas tienen tres atributos: `nombre`, `apellido` y `DNI`.

El `DNI` sería una constante (en Java se antepone la palabra `final`). No hay un estándar para definir constantes en UML, aunque se puede inferir que `DNI` es constante por su nombre enteramente en mayúsculas.

```
1   public class Persona {
2         String nombre;
3         String apellido;
4         final String DNI;
5   }
```



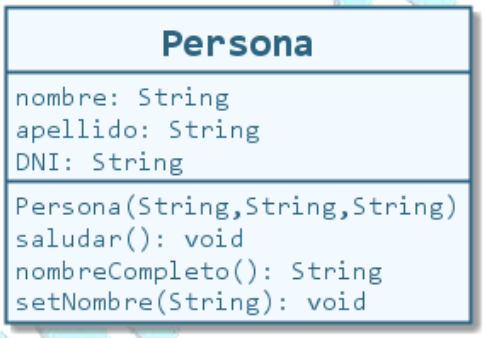
Constructores y métodos

Los constructores se suelen poner al principio de la lista. Recordá que deben llevar el mismo que la clase seguido de la lista de tipos de datos de sus parámetros entre paréntesis, separados por comas. Si no hay parámetros, se dejan los paréntesis vacíos. Para este ejemplo, las personas tienen únicamente un constructor con todos los parámetros.

Para los métodos, se escribe el nombre del método, seguido de la lista de tipos de datos de sus parámetros entre paréntesis, separados por comas. Si no hay parámetros, se dejan los paréntesis vacíos. A continuación se coloca un `:` y el tipo de dato de retorno (si no hay retorno, se escribe `void`). Para este ejemplo las personas tienen la capacidad de saludar, decir su nombre completo y cambiar de nombre.

(Se adjunta ejemplo en la página siguiente)

```java
public class Persona {
  String nombre;
  String apellido;
  final String DNI;
  Persona (String n, String a, String d) {
    nombre = n;
    apellido = a;
    DNI = d;
  }
  void saludar() {
    System.out.println("Hola!");
  }
  String nombreCompleto() {
    return nombre + " " + apellido;
  }
  void setNombre (String nombre) {
    this.nombre = nombre;
  }
}
```



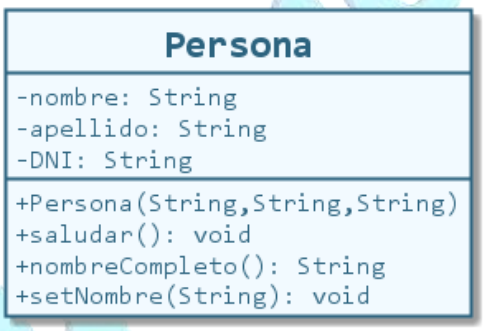
Visibilidad de los miembros

Cada uno de los miembros de una clase (atributos, constructores y métodos) puede tener diferente visibilidad, las cuales se modelan de la siguiente manera:

- **Público** (`public` en Java). Se representa con un **+**. Indica que ese miembro es accesible desde cualquier lugar.
- **Privado** (`private` en Java). Se representa con un **–**. Indica que ese miembro es accesible solo desde la propia clase.
- **Protegido** (`protected` en Java). Se representa con un **#**. Indica que ese miembro es accesible desde la propia clase, por otras clases dentro del mismo paquete y por sus subclases (sin importar el paquete donde se encuentren).
- **De paquete**. No se representa. Indica que ese miembro es accesible desde la propia clase y por otras clases dentro del mismo paquete.

Para el ejemplo que estamos tratando, los atributos serán privados y los constructores y métodos serán públicos. (Se adjunta ejemplo en la página siguiente)

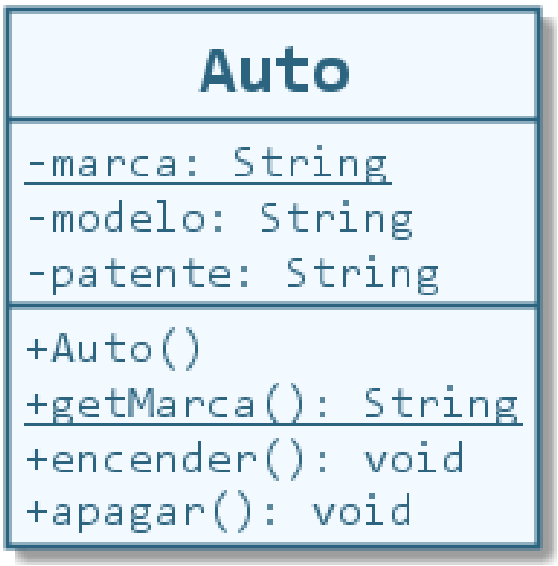
```java
public class Persona {
  private String nombre;
  private String apellido;
  private final String DNI;
  Persona (String n, String a, String d) {
    nombre = n;
    apellido = a;
    DNI = d;
  }
  public void saludar() {
    System.out.println("Hola!");
  }
  public String nombreCompleto() {
    return nombre + " " + apellido;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
}
```



Miembros de clase

A aquellos miembros que sean estáticos, es decir, de clase, se los debe subrayar para diferenciarlos con miembros de instancia.

```java
public class Auto {
  private static String marca;
  private String modelo;
  private String patente;
  public static String getMarca() {
    return Auto.marca;
  }
  public void encender() {…}
  public void apagar() {…}
}
```



Clases enumeradas

Las clases enumeradas permiten estandarizar ciertos valores, limitando las instancias de una clase exclusivamente a aquellas enumeradas dentro de la misma. Para este tipo de clases se utiliza el estereotipo **<<enum>>** justo arriba del nombre. Todos y cada uno de los valores enumerados son objetos de la propia clase enumerada, estáticos (**static**), constantes (**final**) y públicos.

Además, cuentan con algunos métodos de clase (los más usados, **values** y **valueOf**) y de instancia (los más usados, **name** y **ordinal**). El constructor de las clases enumeradas es y debe ser privado.

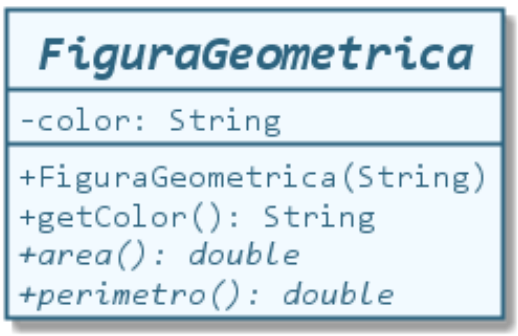
Para simplificar la representación de los enumerados en UML, por convención se suelen dejar únicamente los valores enumerados.

```
1   public enum EstadoCivil {
2     SOLTERO,
3     CASADO,
4     DIVORCIADO,
5     VIUDO;
6   }
```

Clases y métodos abstractos

Cuando un método o una clase sea abstracto se lo debe escribir en *itálica* para diferenciarlo con métodos o clases concretas.

```
1   public abstract class FiguraGeometrica {
2     private String color;
3     // Se omite constructor
4     public String getColor() {}
5     public abstract double area();
6     public abstract double perimetro();
7   }
```



Interfaces

Las interfaces son un tipo especial de clases abstractas que únicamente poseen métodos abstractos o constantes de clase. Para diferenciarlas de las clases abstractas que podrían tener además atributos y métodos concretos, se utiliza el estereotipo **<<interface>>** justo arriba de su nombre.

```
1   public interface Dibujable {
2     public abstract void dibujar ();
3   }
```

Tipos de relaciones entre clases

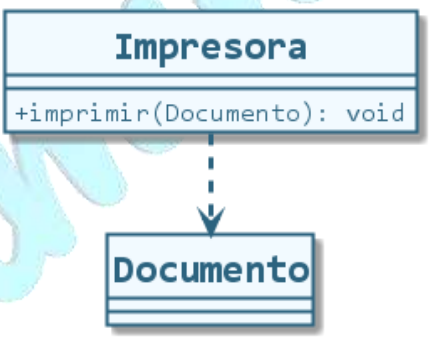
Uso o dependencia

Cuando una clase utiliza a otra clase hablamos de **uso** o **dependencia**. Es una relación débil, ya que la clase que utiliza no tiene un atributo del tipo de la clase utilizada, sino que simplemente instancia un objeto como variable local en un método o lo recibe por parámetro.

Una dependencia entre clases se representa con la siguiente flecha: 

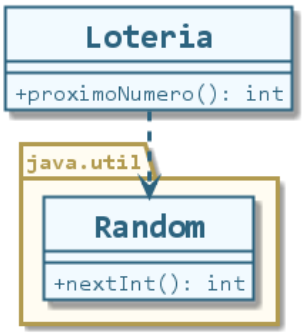
Por ejemplo, la clase `Impresora` depende de la clase `Documento` para hacer su tarea, pero no hay ningún atributo de tipo `Documento` en la clase `Impresora`, simplemente recibe una instancia de `Documento` como parámetro dentro del método `imprimir`.

```
1   public class Documento {}
2
3   public class Impresora {
4     public void imprimir (Documento doc) {
5       System.out.println("Imprimiendo " + doc);
6     }
7   }
```



La clase `Loteria` utiliza la clase `Random` para obtener un número entero al azar y devolverlo, pero no hay ningún atributo de tipo `Random` en la clase, simplemente se utiliza una instancia de `Random` dentro del método `proximoNumero`.

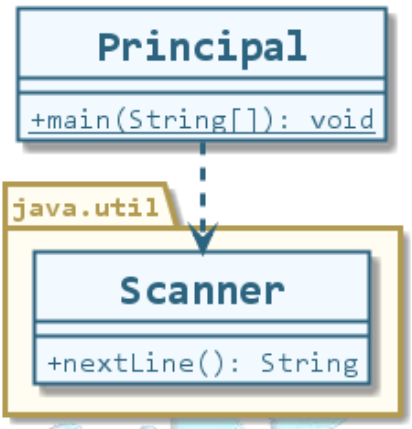
```
1   public class Loteria {
2     public int proximoNumero () {
3       Random r = new Random();
4       return r.nextInt();
5     }
6   }
```



La clase `Principal` utiliza la clase `Scanner` para obtener una cadena ingresada por el usuario desde la consola y mostrarla, pero no hay ningún atributo de tipo `Scanner` en la clase, simplemente se utiliza una instancia de Scanner dentro del método `main`.

(Se adjunta ejemplo en la página siguiente)

```java
public class Principal {
  public static void main (String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Escribí algo: ");
    String cad = sc.nextLine();
    System.out.println("Ingresó " + cad);
  }
}
```



Asociación

Cuando una clase tiene un atributo cuyo tipo es de otra clase, hablamos de **asociación**. Este tipo de relación puede tener cualquier significado y no especifica el ciclo de vida de los objetos intervinientes.

Una asociación entre clases se representa con la siguiente flecha: 

Por ejemplo, en un tren **viajan** pasajeros:

```java
public class Pasajero {}

public class Tren {
  private Pasajero[] pasajeros;
}
```

Una persona **tiene una** mascota:

```java
public class Animal {}

public class Persona {
  private Animal mascota;
}
```

Un profesor **tiene asignados** cursos:

```java
public class Curso {}

public class Profesor {
  private Curso[] cursos;
}
```

Una persona **es amiga de** otras personas:

```
1   public class Persona {
2     private Persona[] amigos;
3   }
```

Agregación

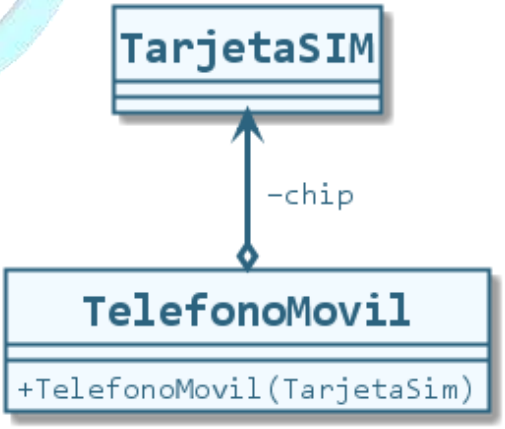
Un objeto puede estar compuesto de otro objeto. El ciclo de vida del objeto componente es independiente del objeto compuesto, esto significa que, si el objeto componente se destruye, el objeto compuesto de todas maneras puede existir (exista una referencia hacia él en otra clase). Este tipo de relación, se llama **agregación** y se cumple que un objeto **tiene un (has-a)** objeto.

Para que se cumpla esta relación, el objeto compuesto se recibe como parámetro (podría ser en un constructor o en un método setter). Esto asegura que tal objeto se creó en otro lugar, asegurando que exista una referencia a él si el objeto compuesto se destruye.

Una agregación entre clases se representa con la siguiente flecha: 

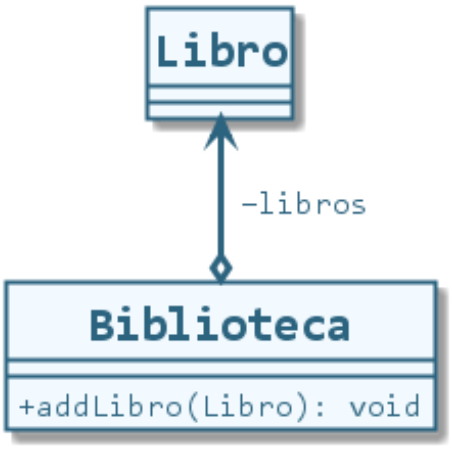
Por ejemplo, un teléfono móvil **tiene una** tarjeta SIM. Si el teléfono móvil se destruye, la tarjeta SIM puede ser utilizada en otro teléfono móvil.

```
1   public class TarjetaSIM {}
2
3   public class TelefonoMovil {
4     private TarjetaSIM chip;
5     // El chip "llega de afuera"
6     public TelefonoMovil (TarjetaSIM c) {
7       chip = c;
8     }
9   }
```



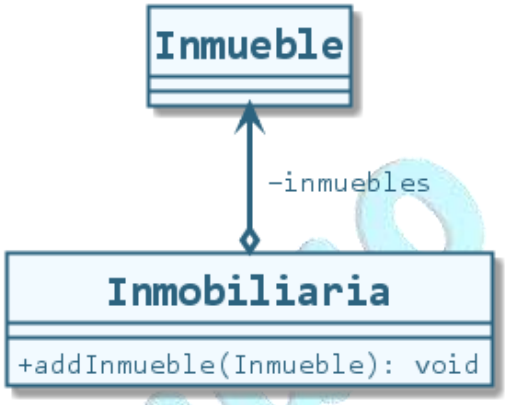
Una biblioteca **tiene** libros. Si la biblioteca deja de funcionar, sus libros pueden ofrecerse en otra.

```
1   public class Libro {}
2
3   public class Biblioteca {
4     private ArrayList<Libro> libros;
5     // El libro "llega de afuera"
6     public addLibro (Libro p) {
7       libros.add(p);
8     }
9   }
```



Una inmobiliaria **tiene** inmuebles. Si la inmobiliaria ya no existe, los inmuebles pueden ser ofrecidos por otras inmobiliarias.

```
1   public class Inmueble {}
2
3   public class Inmobiliaria {
4     private ArrayList<Inmueble> inmuebles;
5     // El inmueble "llega de afuera"
6     public addInmueble (Inmueble m) {
7       inmuebles.add(m);
8     }
9   }
```



Composición

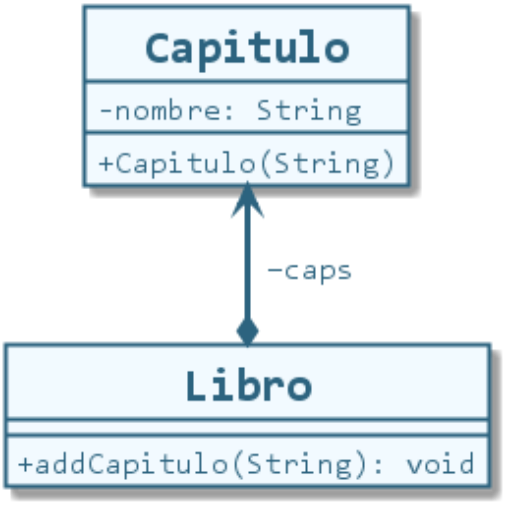
Un objeto puede estar compuesto de otro objeto. El ciclo de vida del objeto componente es totalmente dependiente del objeto compuesto, esto significa que, si el objeto componente se destruye, el objeto compuesto se destruye también (hay una única referencia hacia él en el objeto componente). Este tipo de relación, se llama **composición** y se cumple que un objeto **tiene un (has-a)** objeto.

Para que se cumpla esta relación, el objeto compuesto se instancia dentro de la clase del objeto componente. Esto asegura que ambos tengan el mismo ciclo de vida.

Una composición entre clases se representa con la siguiente flecha: 

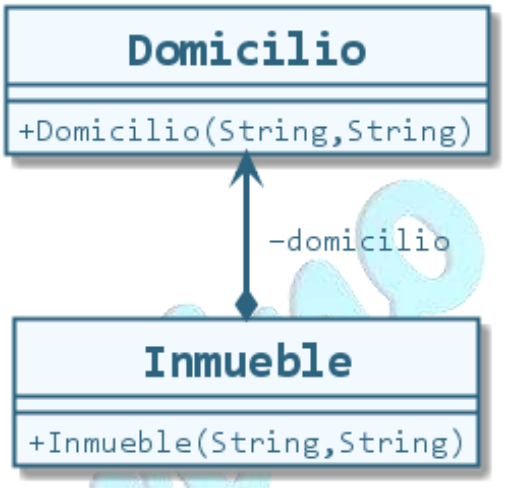
Por ejemplo, un libro **tiene** capítulos. Si el libro se destruye, sus capítulos no tienen razón de existir.

```
1   public class Capitulo {
2     private String nombre;
3     public Capitulo (String nombre) {
4       this.nombre = nombre;
5     }
6   }
7
8   public class Libro {
9     private ArrayList<Capitulo> caps;
10    // El capítulo "nace dentro"
11    public void addCapitulo (String nombre) {
12      caps.add( new Capitulo(nombre) );
13    }
14  }
```



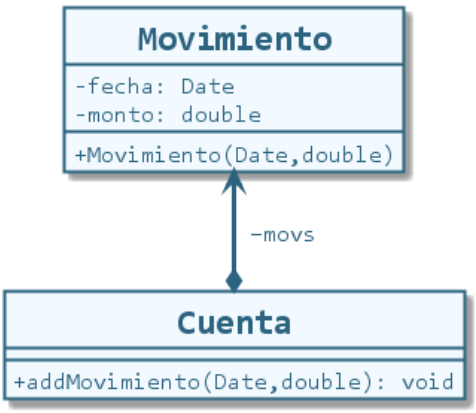
Un inmueble **tiene un** domicilio. Si se pierde la referencia a un inmueble, la de su domicilio también.

```
1   public class Domicilio {
2      public Domicilio (String calle, String num) {…}
3      // Se omite implementación del constructor
4   }
5
6   public class Inmueble {
7      private Domicilio domicilio;
8      // El domicilio "nace dentro"
9      public Inmueble (String calle, String num) {
10        domicilio = new Domicilio(calle,num);
11     }
12  }
```



Una cuenta bancaria **tiene** movimientos. Si la cuenta bancaria se cierra, sus movimientos se eliminan.

```
1   public class Movimiento {
2      private Date fecha;
3      private double monto;
4      public Movimiento (Date f, double m) {
5         fecha = f;
6         monto = m;
7      }
8   }
9
10  public class Cuenta {
11     private ArrayList<Movimiento> movs;
12     // El movimiento "nace dentro"
13     public void addMovimiento (Date fecha, double monto) {
14        movs.add( new Movimiento(fecha,monto) );
15     }
16  }
```



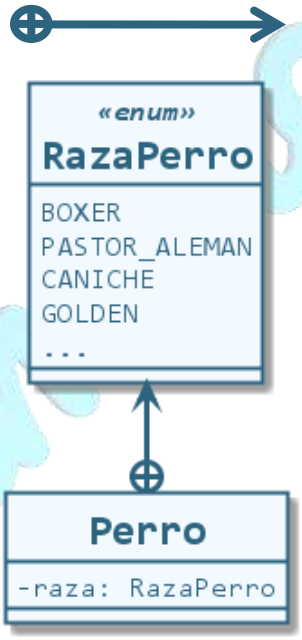
Anidamiento

Un tipo de objeto puede llegar a ser exclusivamente componente de otro. Este tipo de relación se llama **anidamiento**.

Para que se cumpla esta relación, la clase componente se define dentro de la clase compuesta, asegurando que solo se puedan instanciar objetos de la clase componente dentro de la clase compuesta.

Una clase anidando a otra se representa con la siguiente flecha:

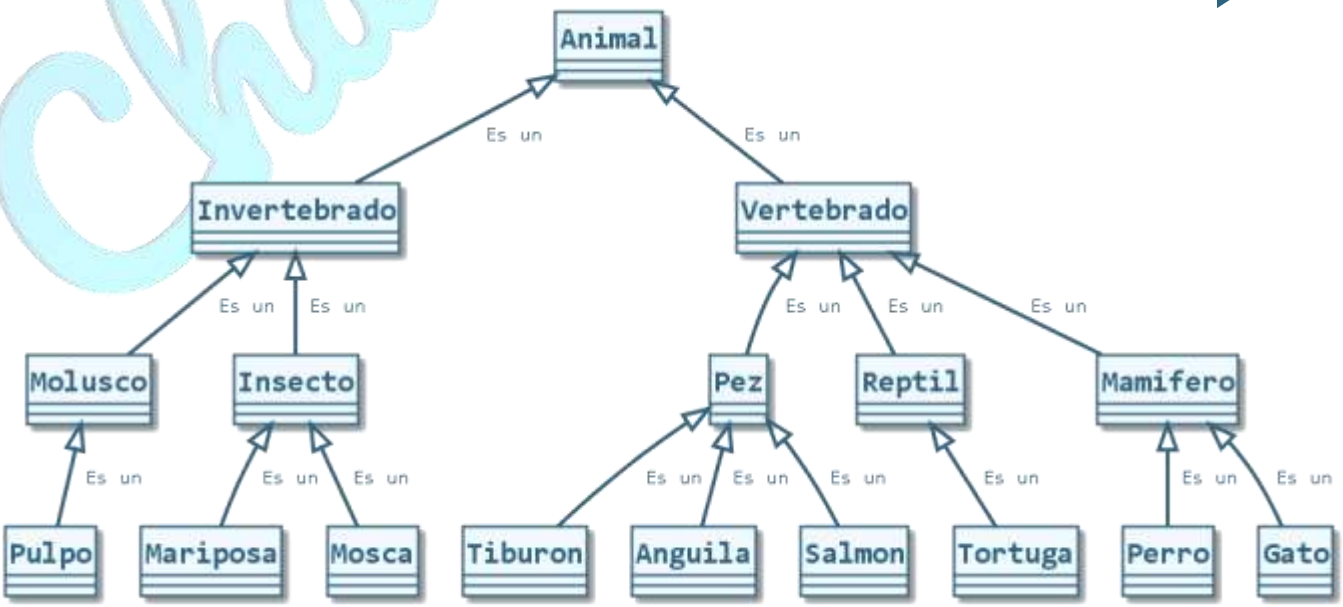
```
1   public class Perro {
2     private RazaPerro raza;
3
4     private enum RazaPerro {
5       BOXER,
6       PASTOR_ALEMAN,
7       CANICHE,
8       GOLDEN;
9     }
10  }
```



Generalización

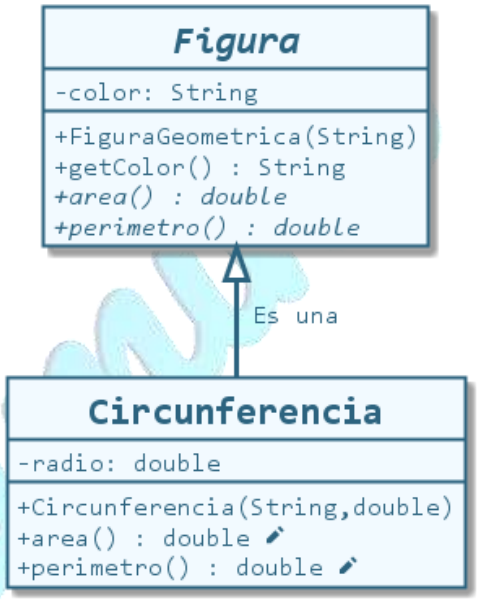
Cuando una subclase (también llamada clase hija o clase derivada) comparte estado y comportamiento con una superclase (también llamada clase padre o clase base), la relación es de **generalización**. Las subclases heredan todos los atributos y métodos de la superclase que no sean privados. **Los constructores no se heredan**. Se cumple que el objeto de la subclase **es un (is-a)** tipo especial de objeto de la superclase.

Una generalización entre clases se representa con la siguiente flecha:



Toda circunferencia es una figura geométrica, por lo tanto, se comportará como tal además de poder tener cierto estado y comportamiento propio de una circunferencia, como podría ser el radio. La subclase `Circunferencia` extiende (`extends`) de la superclase `Figura`.

```
1   public abstract class Figura {
2     private String color;
3     // Se omite constructor
4     public String getColor() {}
5     public abstract double area();
6     public abstract double perimetro();
7   }
8   public class Circunferencia extends Figura {
9     private double radio;
10    // Se omite constructor
11    @Override
12    public double area() { return … }
13    @Override
14    public double perimetro() { return … }
15  }
```



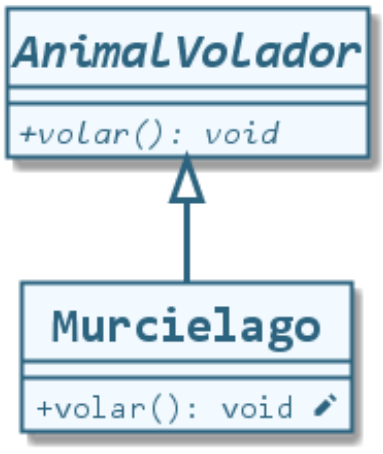
Realización

Cuando una clase cumple con un contrato especificado en una interfaz, la relación es de **realización**. La clase debe sobrescribir todos los métodos de la interfaz, los cuales son públicos y abstractos. En la realización se cumple también la relación **es un (is-a)**.

Una realización entre clases se representa con la siguiente flecha: 

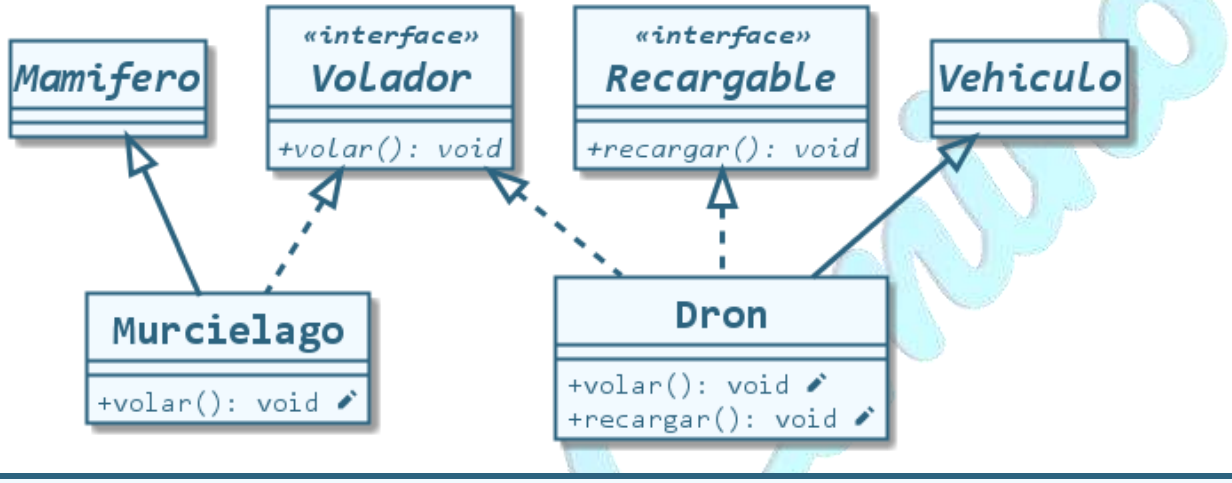
Por ejemplo, un murciélago tiene la capacidad de volar, por lo tanto, la subclase `Murcielago` podría extender (`extends`) de una superclase `AnimalVolador`.

```
1   public abstract class AnimalVolador {
2     public abstract void volar();
3   }
4
5   public class Murcielago extends AnimalVolador {
6     @Override
7     public void volar() {
8       System.out.println("Aleteo mis alas");
9     }
10  }
```



Sin embargo, los animales no son los únicos capaces de volar, por lo tanto, si en el modelo aparecen vehículos aéreos, éstos no serán animales. Las interfaces permiten emular la herencia múltiple: una subclase es hija (`extends`) como máximo de una superclase, pero puede implementar (`implements`) varias interfaces.

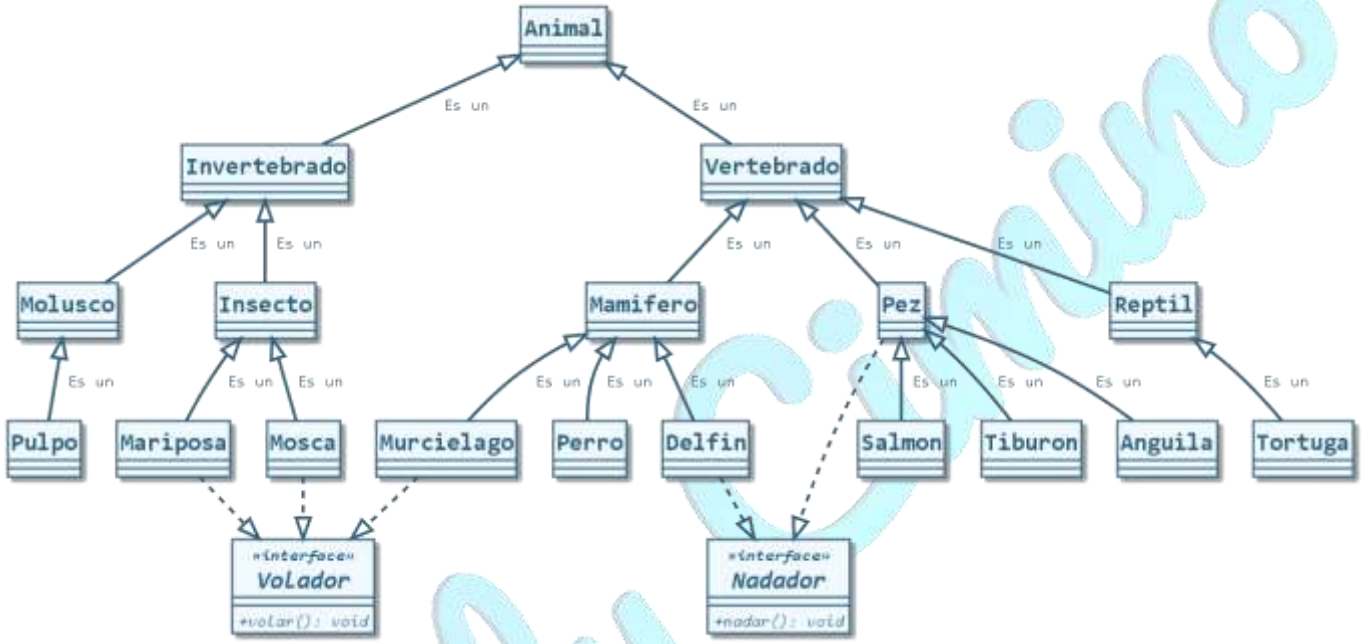
Tanto los murciélagos (que son un tipo especial de mamífero) como los drones (que son un tipo especial de vehículo) tienen la capacidad de volar, por lo tanto, las clases `Murcielago` y `Dron` realizan o implementan (`implements`) la interfaz `Volador`. Además, los drones pueden recargar sus baterías, por lo tanto, `Dron` implementa además (`implements`) la interfaz `Recargable`.




```
1   public class Mamifero {}
2   public class Vehiculo {}
3   public interface Volador {
4     public abstract void volar();
5   }
6   public interface Recargable {
7     public abstract void recargar();
8   }
9   public class Murcielago extends Mamifero implements Volador {
10    @Override
11    public void volar() {
12      System.out.println("Aleteo mis alas");
13    }
14  }
15  public class Dron extends Vehiculo implements Volador, Recargable {
16    @Override
17    public void volar() {
18      System.out.println("Giro mis hélices");
19    }
```

```
20    @Override
21    public void recargar() {
22      System.out.println("Recargo mis baterías");
23    }
24  }
```

En el ejemplo de la jerarquía de animales, el uso de interfaces ofrece flexibilidad al modelo:



Resumen de relaciones entre clases

Tipo de flecha	Nombre de la relación
	Uso o dependencia
	Asociación
	Agregación
	Composición
	Anidamiento
	Generalización
	Realización