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.

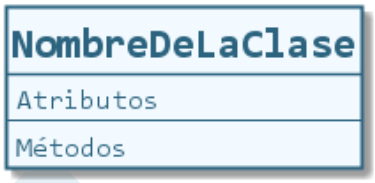
Por el momento, este apunta se va a centrar exclusivamente en los diagramas de clase UML, de los cuales hablaremos a continuación.

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`.

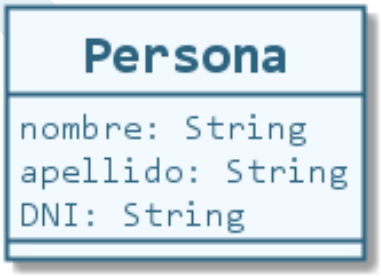
Atributos

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   }
```



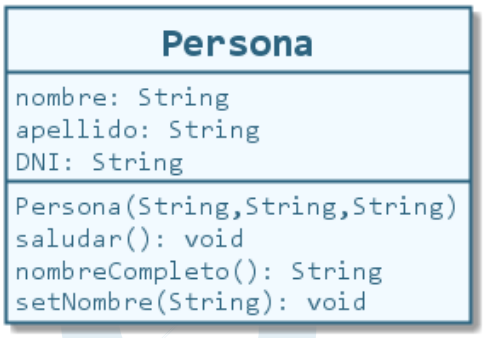
Constructores y métodos

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

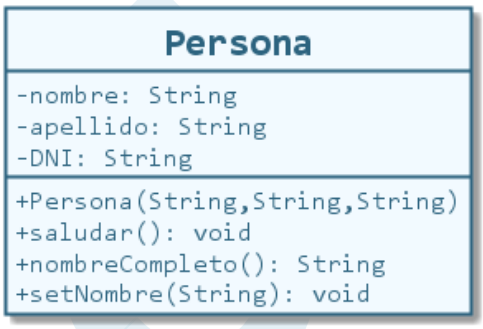
Cada uno de los miembros de una clase (atributos, constructores y métodos) puede tener diferente visibilidad, las cuales se modelan de la siguiente manera:

- **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;
  }
}
```

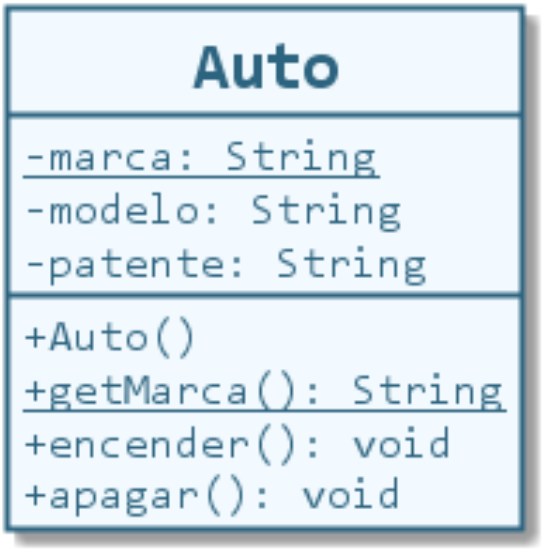


Miembros de clase

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

Supongamos un sistema para una concesionaria, cuyos autos comparten la misma marca. Si el fabricante cambia de nombre, todos los coches cambiarán el nombre de la marca. Tal variable `marca` podría ser de clase (`static` en Java) al igual que el método `getMarca()`, para evitar que se copie el mismo valor en cada 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.

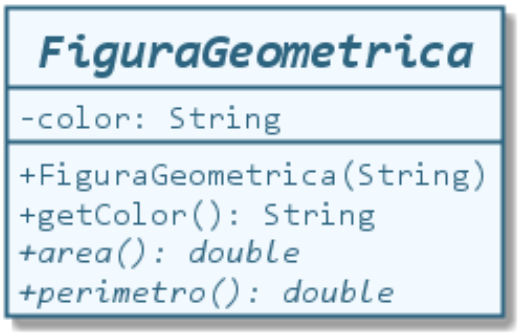
```
1   public enum EstadoCivil {
2     SOLTERO,
3     CASADO,
4     DIVORCIADO,
5     VIUDO;
6   }
```

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.

Por ejemplo, todas las figuras geométricas son capaces de devolver el valor de su área y perímetro, pero cada tipo de figura calcula estos valores de formas diferentes. Por lo tanto, los métodos **area** y **perimetro** en **FiguraGeometrica** son abstractos, al igual que la clase.

```
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   }
```