

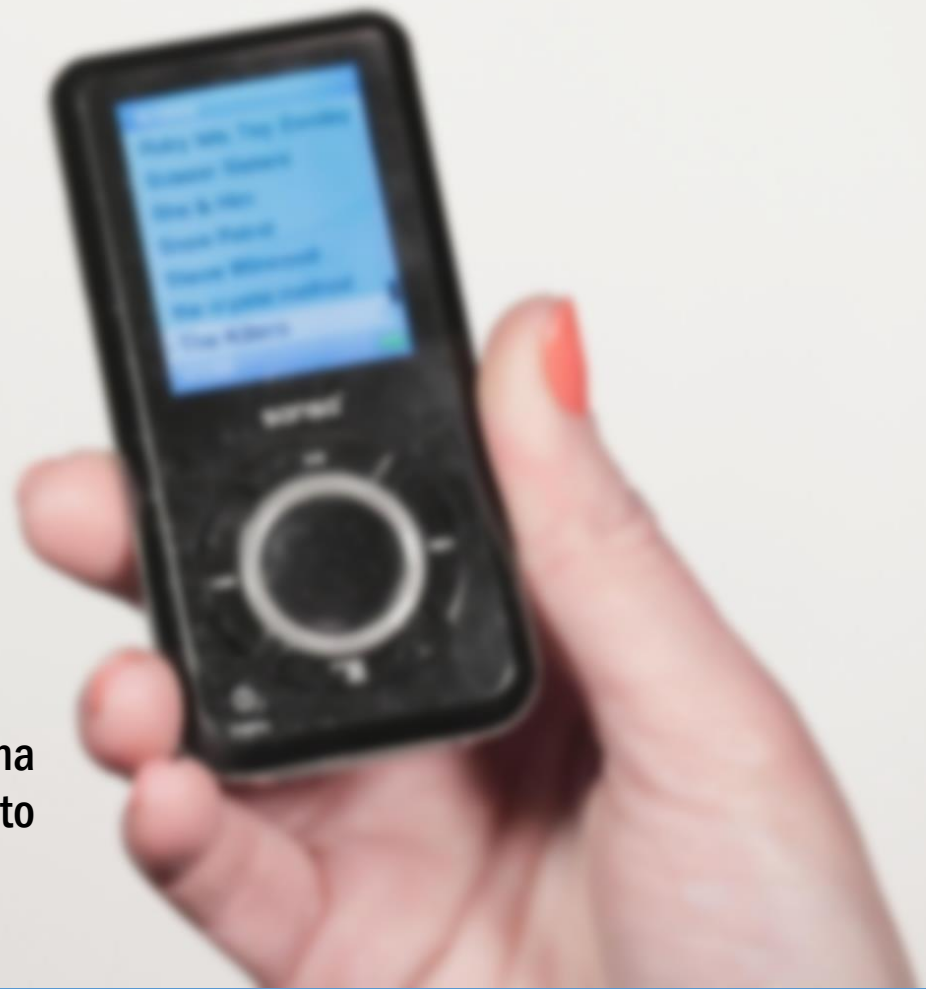
MODEL-VIEW-CONTROLLER (MVC)

Imagina que estás usando un reproductor de MP3.

Puedes usar su interfaz para:

- agregar nuevas canciones,
- administrar listas de reproducción,
- renombrar pistas ...

El reproductor se ocupa de mantener una base de datos de todas tus canciones junto con sus nombres y datos asociados.



También reproduce las canciones y, tal cual las reproduce, la interfaz de usuario se actualiza constantemente con el título de la canción actual, el tiempo de ejecución, etc.

**Detrás de este proceso está la
arquitectura MVC**

Observa que se actualiza la interfaz y se reproduce la canción

Se actualiza la interfaz de usuario



VISTA

Comunica a la Vista de un cambio de estado

El Modelo le dice a la Vista que el estado ha cambiado



“Reproduce canción”

Usa la interfaz de usuario y sus acciones van al Controlador



CONTROLADOR

Manipula el Modelo

Solicita al Modelo de Reproductor que se reproduzca la canción

```
class Reproductor {  
    play() {}  
  
    stop() {}  
  
    . . .  
}
```

MODELO

1 “El usuario hizo algo”



VISTA



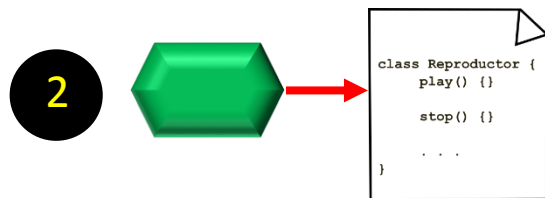
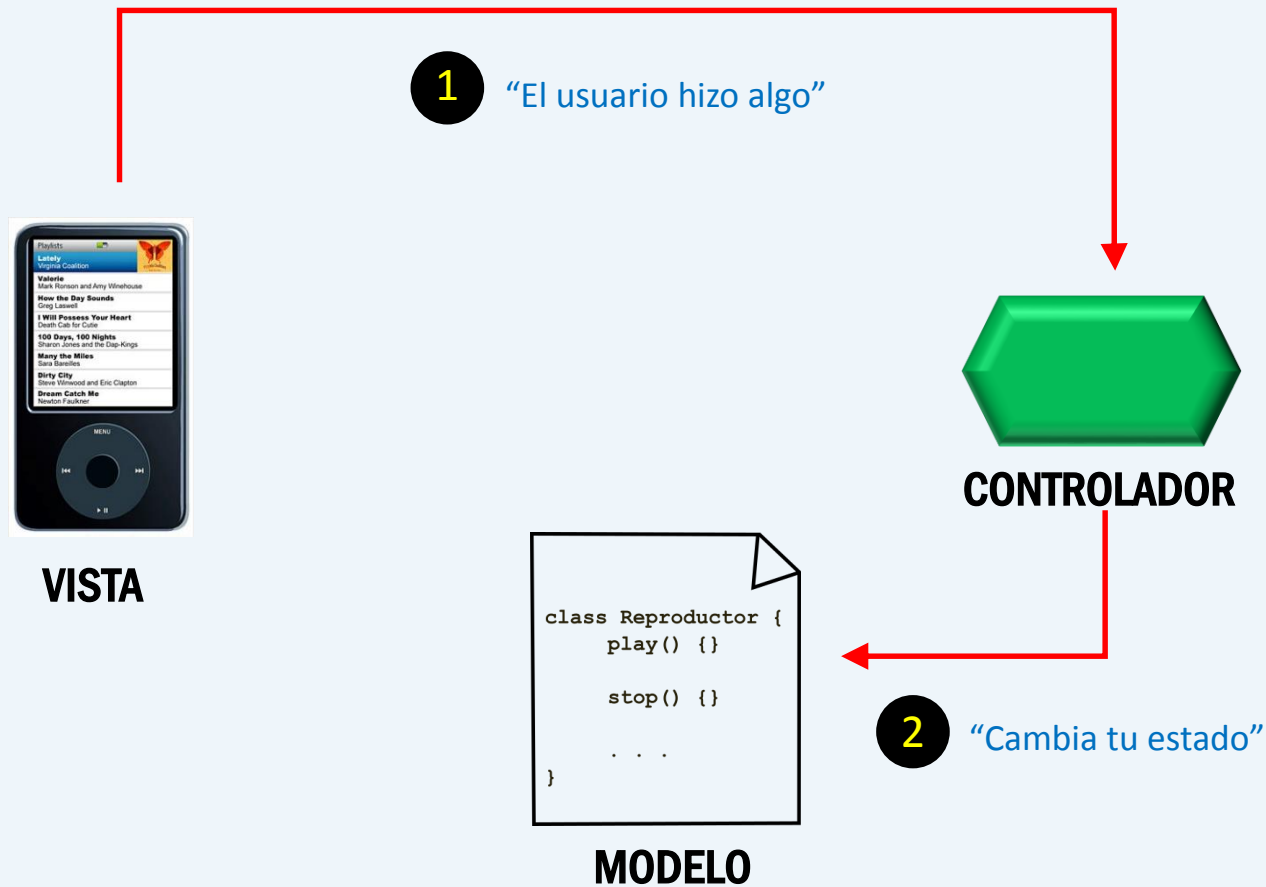
CONTROLADOR

1



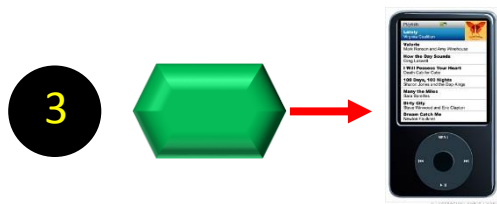
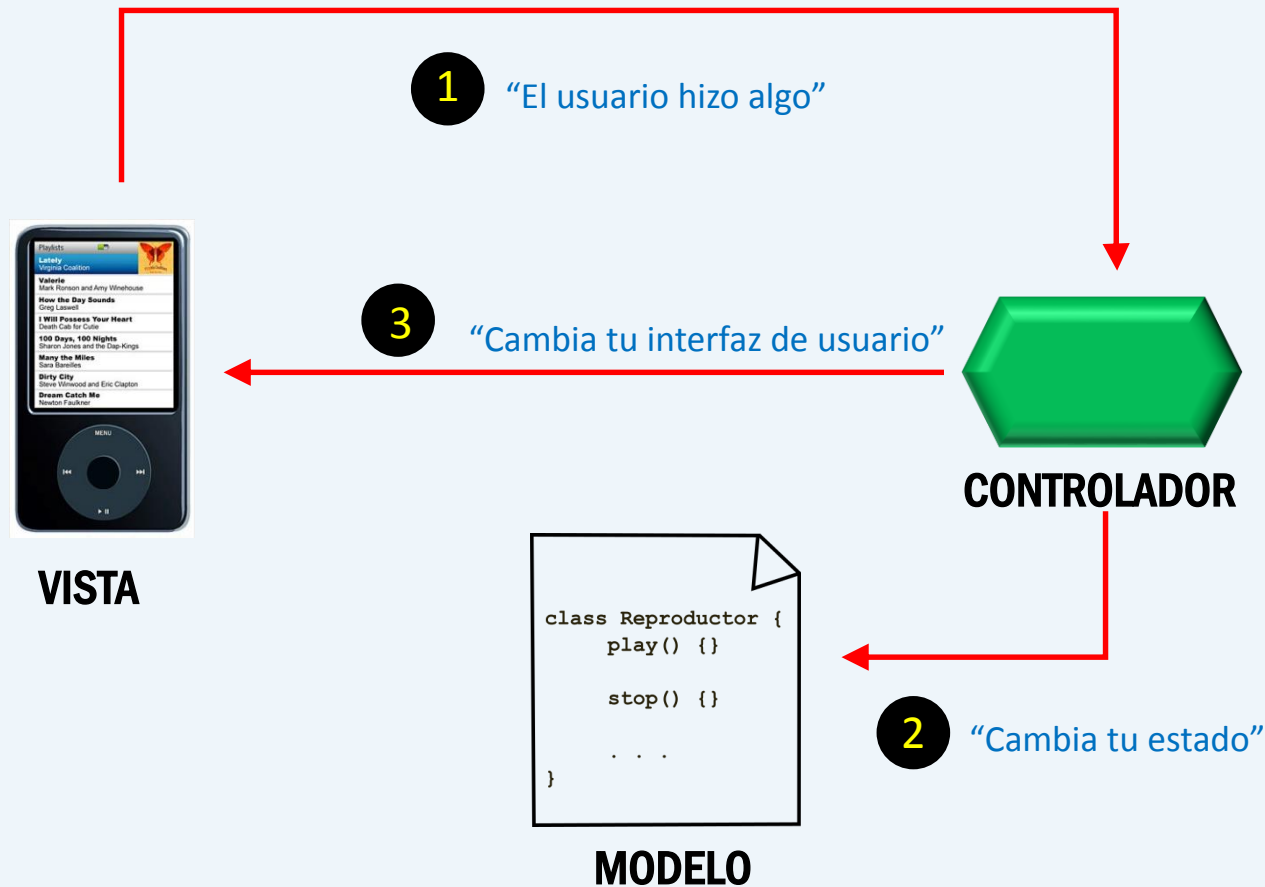
Eres el usuario e interactúas con la Vista.

La Vista es tu “conexión” con el Modelo. Cuando le haces algo a la Vista (p.ej. pulsas en *Play*), entonces la Vista le dice al Controlador lo que has hecho: es tarea del Controlador de manejarlo.



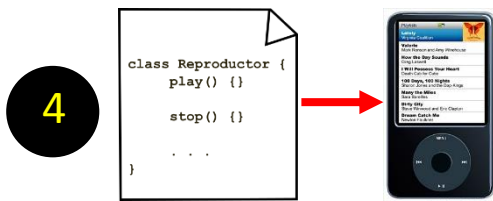
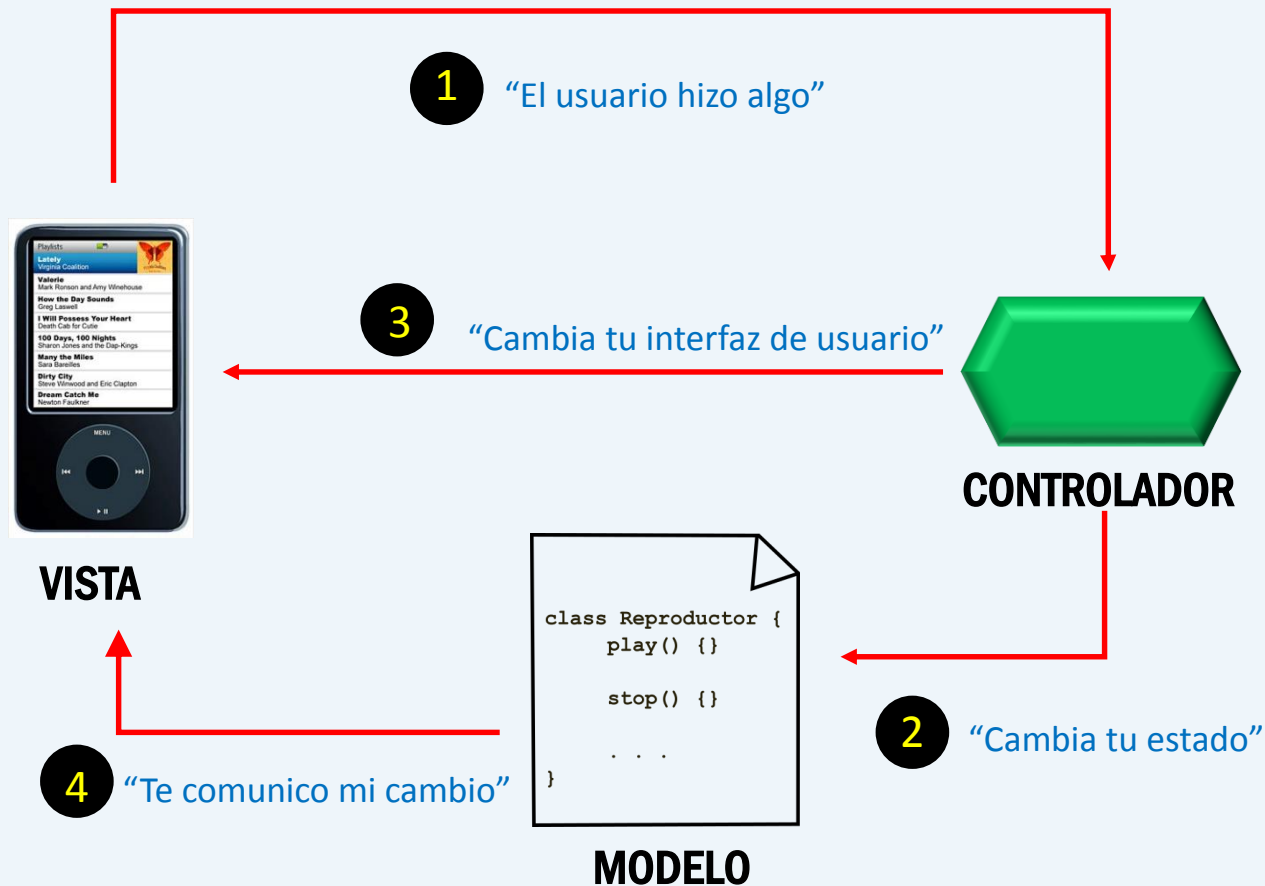
El Controlador solicita al Modelo que cambie su estado.

El Controlador toma tus acciones y las interpreta: si picas en un botón, es trabajo del Controlador averiguar lo que significa y cómo se debe manipular el Modelo dependiendo de esa acción.



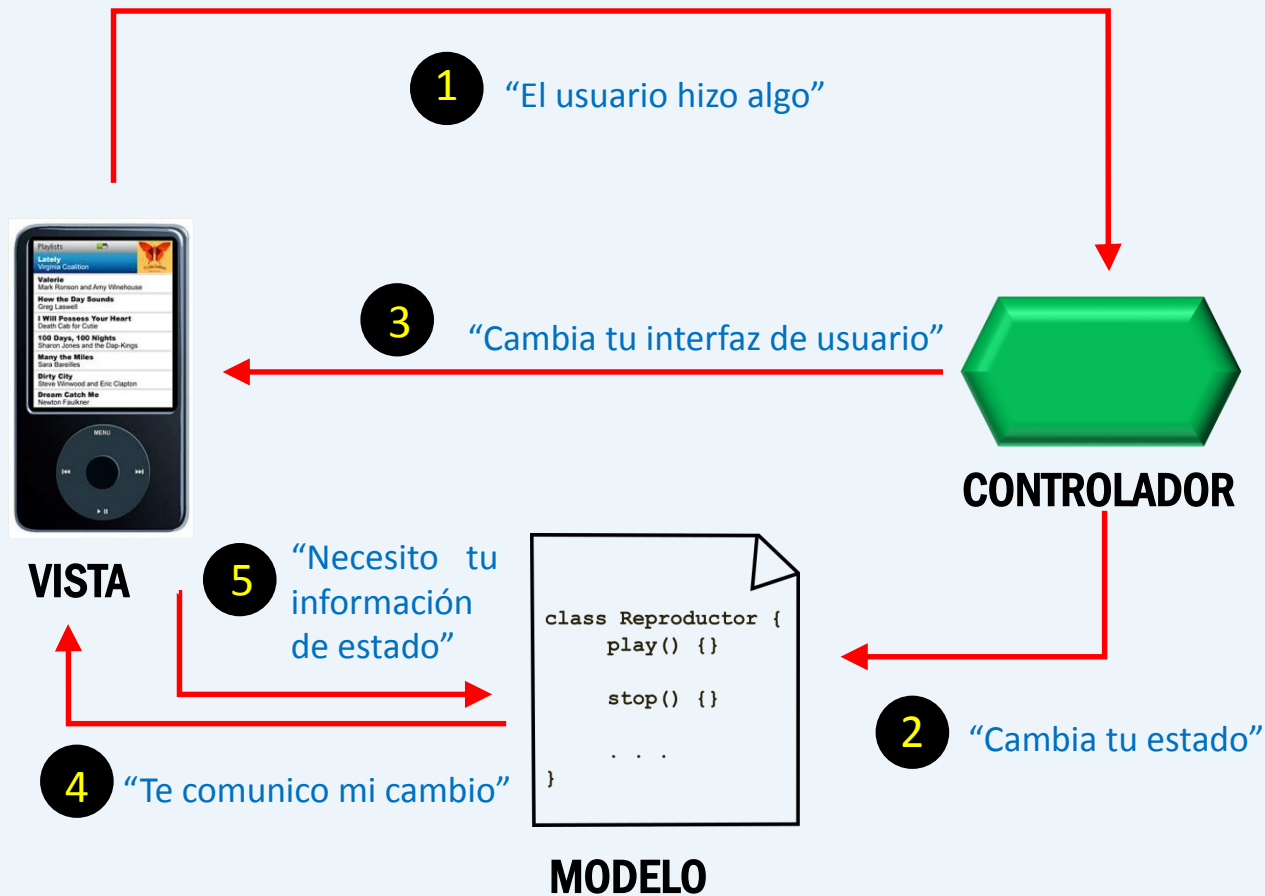
El Controlador también puede solicitar a la Vista que cambie.

Cuando el Controlador recibe una acción por parte de la Vista, como resultado, puede que le diga a la Vista que cambie: p. ej. El Controlador puede habilitar o deshabilitar algunos botones o elementos de menú de la interfaz.



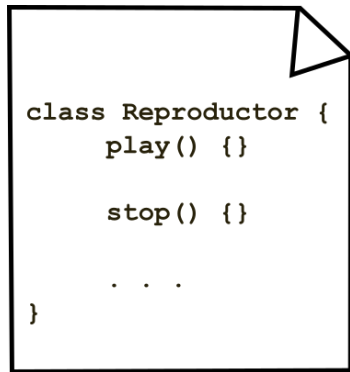
El Modelo notifica a la Vista cuando su estado ha cambiado.

Quando algo cambia en el Modelo basado en alguna acción tomada (p. ej. clic en un botón) o algún otro cambio interno (p. ej. comienza la siguiente canción en la lista de reproducción), el Modelo notifica a la Vista que su estado ha cambiado.



La Vista pregunta al Modelo el estado.

La Vista obtiene el estado que muestra directamente desde el Modelo: p. ej., cuando el Modelo notifica a la Vista que una nueva canción ha comenzado a reproducirse, la Vista solicita el nombre de la canción al Modelo y la muestra. La Vista también podría solicitar al modelo el estado como resultado de que el Controlador solicite algún cambio en la Vista.



```
class Reproductor {  
    play() {}  
  
    stop() {}  
  
    . . .  
}
```

MODELO

Contiene todos los datos, los métodos necesarios para manipularlos y proporciona acceso a los mismos. Es ajeno a la Vista y al Controlador.



VISTA

Te da una presentación del Modelo. Generalmente obtiene directamente desde el Modelo el estado y los datos que necesita para mostrar.



CONTROLADOR

Es el responsable de interpretar la entrada y manipular el Modelo en función de dicha entrada.

¿Qué hace MVC básicamente?

1. Separar, por un lado, todos los datos y cálculos (Modelo).
2. Por otro lado, la interfaz (Vista).
3. Y por otro lado, usa el Controlador como coordinador de todas las interacciones entre la Vista y el Modelo

¿En qué nos ayuda MVC?

Es un **estilo arquitectónico que ayuda a la gestión del cambio**. Al separar la representación de las entidades que la aplicación ha de manejar y sus relaciones (es decir, separa la lógica de negocio) de la interfaz de usuario:

- Facilita la evolución por separado de ambos aspectos.
- Incrementa reutilización y flexibilidad.

Por ejemplo, en una aplicación de gestión de bibliotecas tendremos clases de Modelo para: los libros, usuarios, préstamos, etc., así como métodos para prestar libros, devolverlos, y otros muchos.

```
class Libro {  
    String titulo;  
    String autor;  
    ...  
    public String getTitulo() {}  
    ...  
}
```

```
class Usuario {  
    String nombre;  
    String apellidos;  
    ...  
    public String getNombre() {}  
    ...  
}
```

```
class Prestamo {  
    String titulo;  
    boolean prestado;  
    ...  
    public boolean getPrestado() {}  
    ...  
}
```







La Vista lleva la lógica de presentación y cada Vista sabe mostrar unos datos que recibe y nada más.

JDAL Library Demo





Books
Authors
Categories

Book Filter

Title: Author Name: Author Surname:
Category: Published Before: Published After:

	Name	Author	Category ▾	ISBN
<input type="checkbox"/>	Pattern-Oriented Software Architect...	Michael Kircher	Patterns	978-0-4708-45...
<input type="checkbox"/>	Implementation Patterns	Kent Beck	Patterns	978-0-3214-13...
<input type="checkbox"/>	Test Driven Development: By Exam...	Kent Beck	Patterns	978-0-3211-46...
<input type="checkbox"/>	Expert One-on-One J2EE Design an...	Rod Johnson	Java	978-0-7645-43...
<input type="checkbox"/>	Expert One-on-One J2EE Developm...	Rod Johnson	Java	978-0-7645-58...
<input type="checkbox"/>	Analysis Patterns: Reusable Object ...	Martin Fowler	Java	978-0-2018-95...
<input type="checkbox"/>	Contributing to Eclipse: Principles, P...	Erich Gamma	Java	978-0-3212-46...
<input type="checkbox"/>	Pattern-Oriented Software Architect...	Frank Buschmann	Java	978-0-4700-59...
<input type="checkbox"/>	JUnit Pocket Guide	Kent Beck	Java	978-0-5960-07...
<input type="checkbox"/>	Spring in Action	Craig Walls	Java	978-1-9351-82...

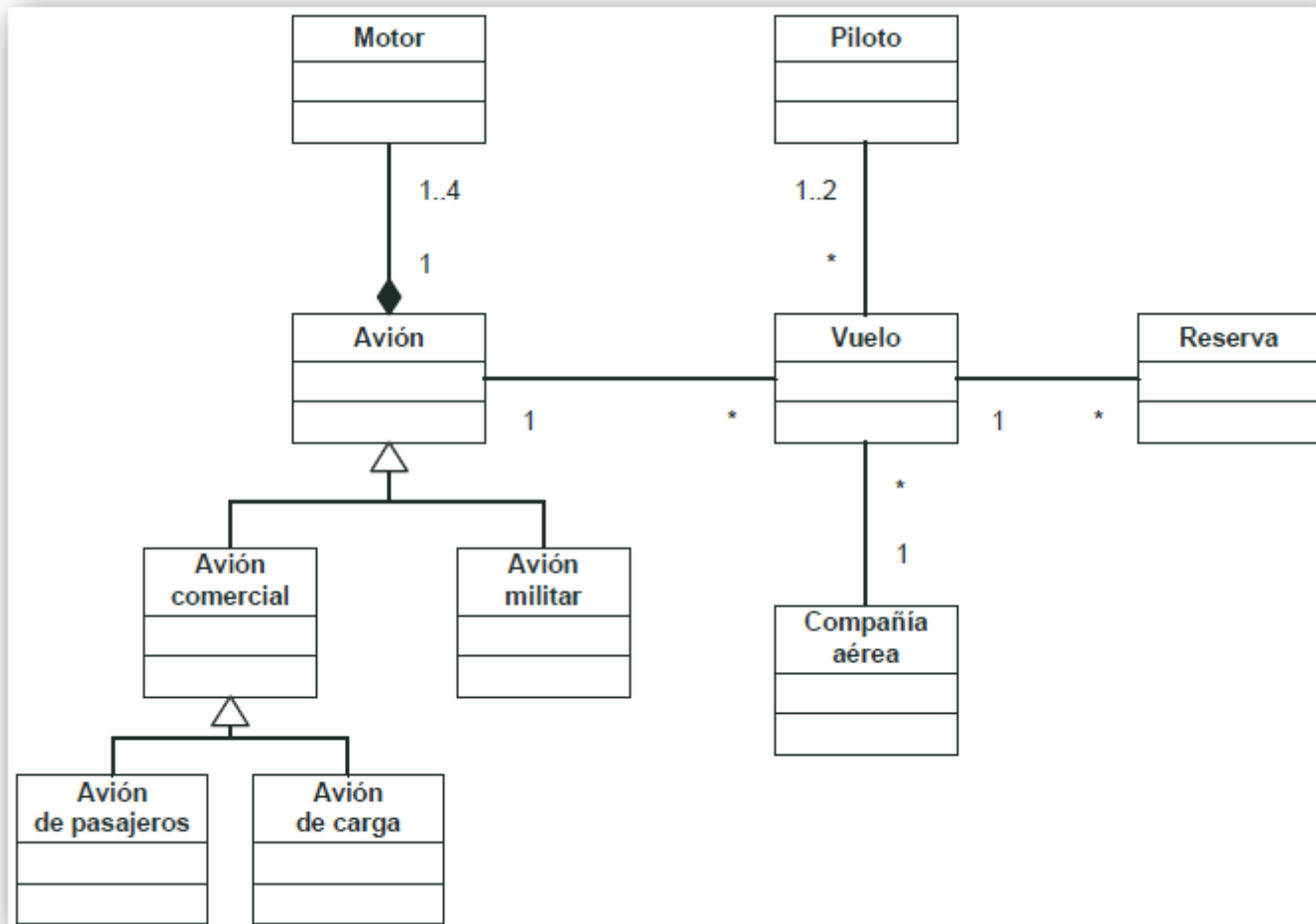
Records: 24   2 / 3   Page Size: 10 ▾

El Controlador:

- a) No solo pide datos al Modelo y se los pasa a la vista,**
- b) También se encarga también de la lógica de la aplicación (la que representa lo que ha de hacer el sistema para solicitar los datos adecuados al modelo y qué hacer con ellos).**

RECORDATORIO DE UML

Diagrama de Clases: es la pieza principal en el modelado orientado a objetos. Se usa para mostrar los diferentes objetos en un sistema, sus atributos, sus operaciones y las relaciones entre ellos.



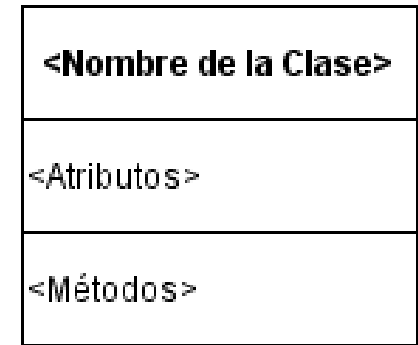
Un Diagramas de Clases está compuesto por los siguientes elementos:

1. **Clase.** Representada por un rectángulo que posee tres divisiones.
 - a) **Superior.** Contiene el nombre de la Clase. Los modificadores de niveles de acceso puede ser *public* (+) o no especificado (*package-private*).
 - b) **Intermedio.** Contiene los atributos. Los modificadores puede ser, además de los dos anteriores: *protected* (#), *private* (-)

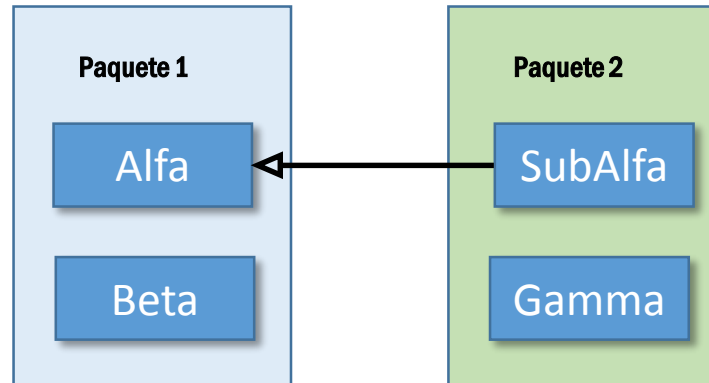
Acceso a los miembros permitido por cada modificador

Modificador	clase	paquete	subclase	todas
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
No especificado	✓	✓	✗	✗
private	✓	✗	✗	✗

- c) **Inferior.** Contiene los métodos u operaciones. Los modificadores son del mismo tipo que la de los atributos.



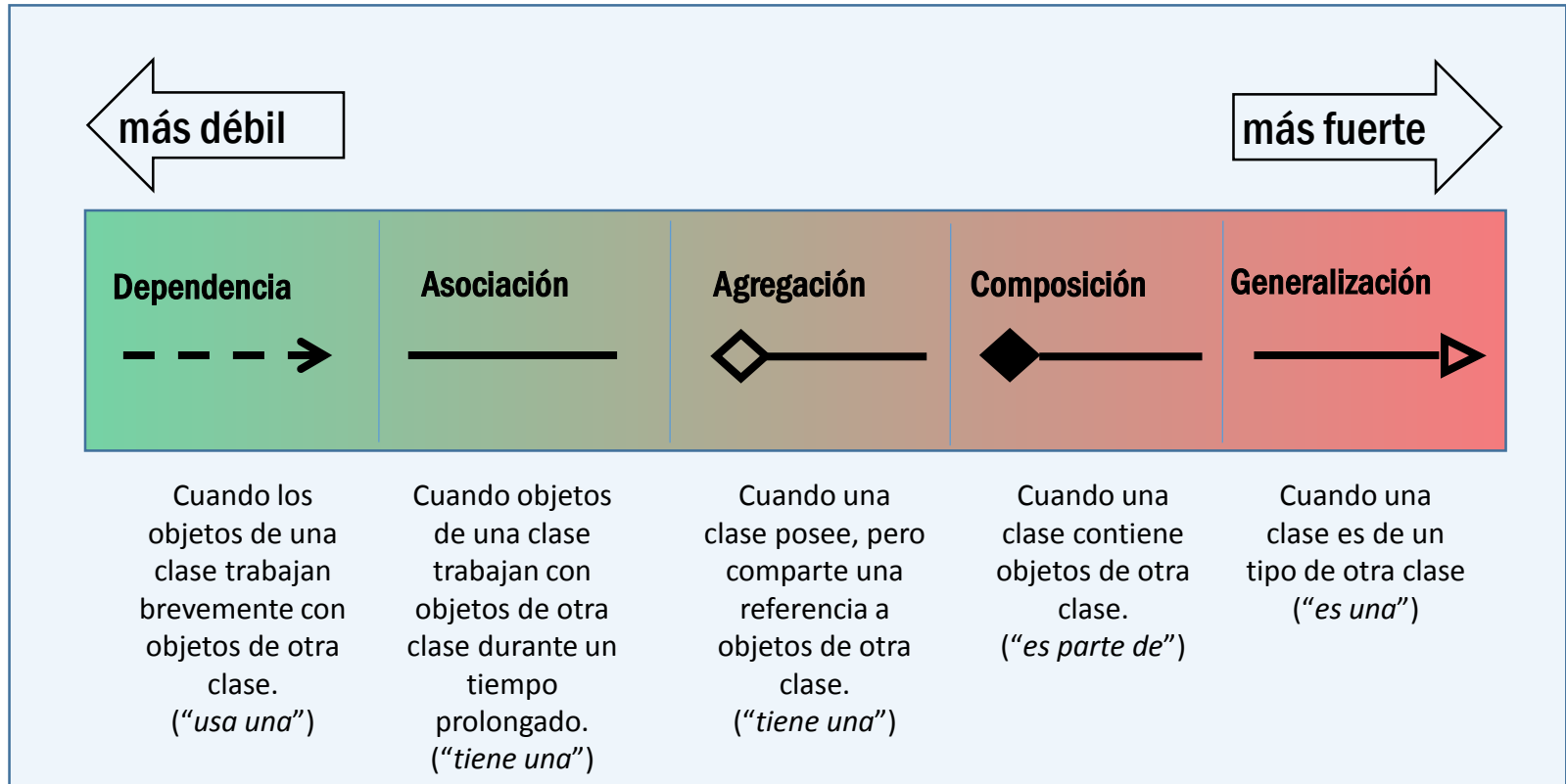
Consideremos varias clases en diferentes paquetes y veamos cómo afecta el nivel de acceso a la visibilidad:



La siguiente tabla muestra en dónde son visibles los miembros de la clase Alfa para cada uno de los modificadores de acceso que se pueden aplicar a ellos.

Visibilidad para los miembros de la clase Alfa				
Modificador	Alfa	Beta	SubAlfa	Gamma
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
No especificado	✓	✓	✗	✗
private	✓	✗	✗	✗

2. Relaciones entre Clases. Las clases trabajan juntas usando diferentes tipos de relaciones. Estas relaciones tienen diferentes “fortalezas”.



La fuerza de una relación de clase se basa en cuánto de dependientes son las clases involucradas en dicha relación.

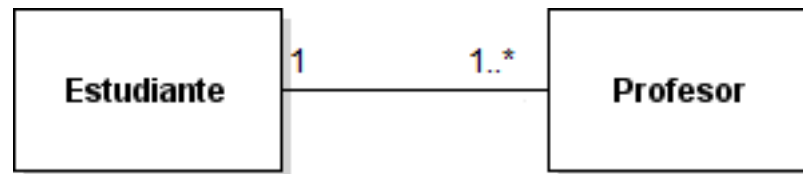
Dos clases que dependen fuertemente una de otra se dice que están estrechamente **acopladas**: los cambios en una clase probablemente afecten a la otra.

El acoplamiento no es en general algo bueno (aunque no siempre). Por lo tanto, cuanto más fuerte sea la relación entre clases, debemos ser más cuidadosos.

La **multiplicidad de las relaciones** indica el grado y nivel de dependencia. Se anota en cada extremo de la relación y básicamente pueden ser:

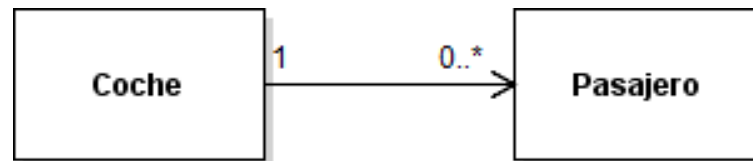
- **una o muchas:** 1..* (1..n)
- **0 o muchas:** 0..* (0..n)
- **número fijo:** m (m indica el número fijo).

Asociación. Si dos clases en un modelo necesitan comunicarse entre sí, debe haber un vínculo entre ellas, y eso se puede representar mediante una asociación (conector).



“Un estudiante puede asociarse con varios profesores”

La asociación se puede representar mediante una línea entre estas clases con una flecha (asociación directa) que indica la dirección de navegación. En caso de que la flecha esté en ambos lados, la asociación tendrá una asociación bidireccional.



“Un coche puede tener varios pasajeros”

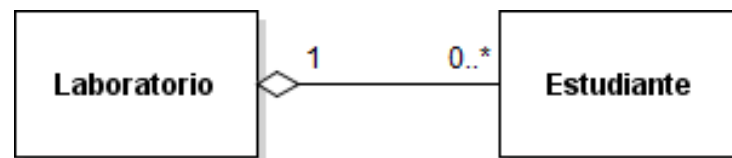
La asociación es una relación débil. Los objetos pueden vivir de forma independiente. Generalmente hay *setters* u otras formas de inyectar los objetos dependientes.

Asociación vs Agregación vs Composición. La Agregación y la Composición son casos específicos de Asociación, además la Composición es un caso específico de Agregación.

Tanto en la Agregación como en la Composición, el objeto de una clase "posee" el objeto de otra clase, pero hay una diferencia sutil:

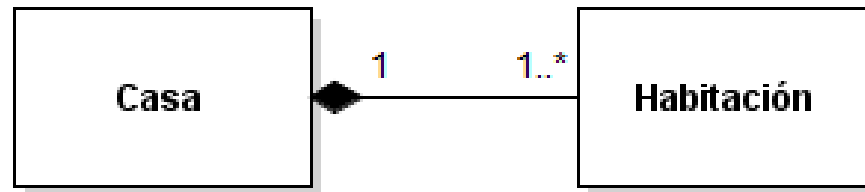
- **La Agregación implica que la clase hija puede existir independientemente de la clase madre.**

Por ejemplo, Laboratorio (madre) – Estudiante (hija). Si borramos Laboratorio, Estudiante todavía ‘vive’: el destruir un objeto Laboratorio no implica destruir un objeto Estudiante.



- La **Composición** implica una relación donde la clase hija no puede existir independientemente de la clase madre.

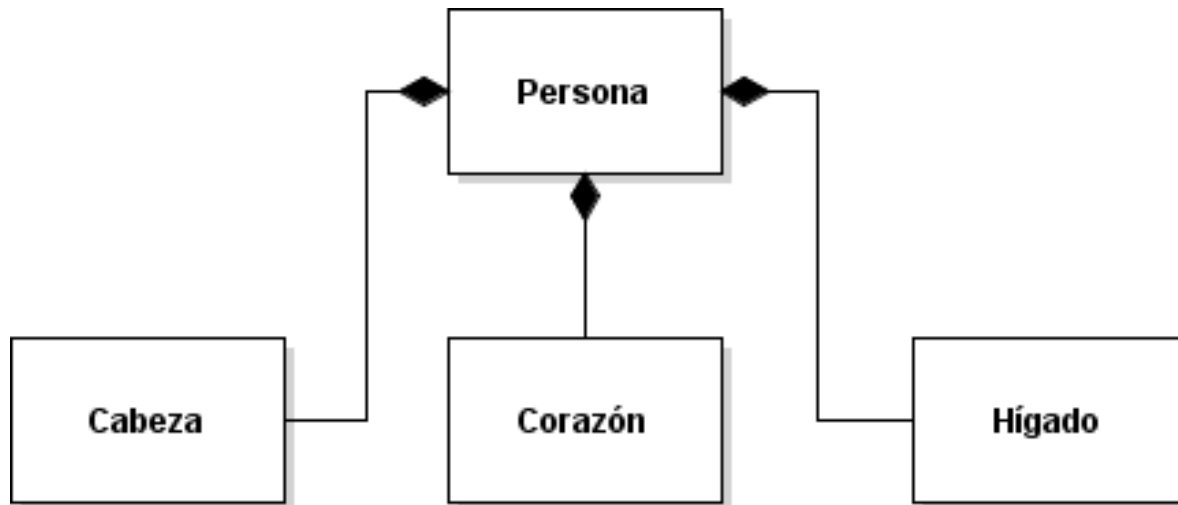
Por ejemplo: Casa (madre) y Habitación (hija). Las habitaciones no existen de forma separada a una casa: si destruimos el objeto casa, el objeto “contenido” habitación desaparece.



Nota: En la Agregación y en la Composición se puede seguir manteniendo la relación “*tiene una*”

¿Cuándo usar Composición? Cuando haya una Asociación entre clases, debemos ser más específicos y usar el enlace de Composición en aquellos casos en que:

- a) Además de la relación “*es parte de*” entre la clase A y la clase B,
- b) Hay una fuerte dependencia del tiempo de vida entre las dos, lo que significa que cuando se elimina la clase A, como resultado, también se elimina la clase B.

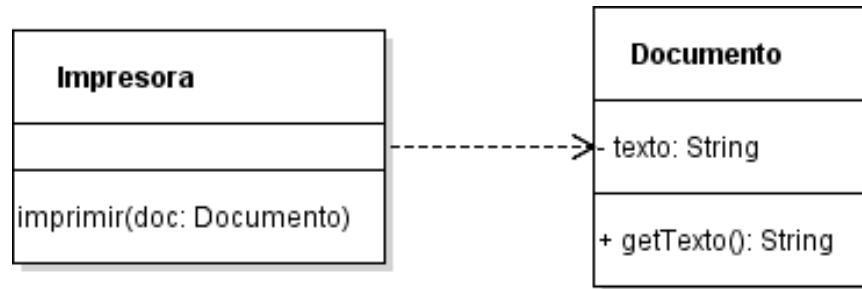


En resumen:

- **La Asociación es un término muy genérico que se usa para representar cuando en una clase se usan las funcionalidades proporcionadas por otra clase.**
- **Decimos que es una Composición si un objeto de la clase madre posee otro objeto de la clase hija y ese objeto de la clase hija no puede existir de manera significativa sin el objeto de clase madre.**
- **Si puede existir, entonces se llama Agregación.**

Además de las relaciones anteriores, también se tienen las siguientes:

Dependencia. Suele expresar el uso de una clase por otra. Es la más básica de las relaciones entre clases, y por lo tanto, la más débil.

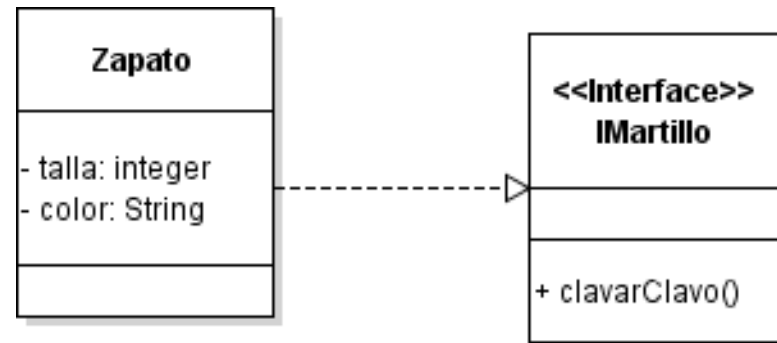


Generalización. La generalización (herencia) se usan para describir una clase que es un tipo de otra clase. Si decidimos que una clase A “*es una*” clase del tipo de otra clase B, deberíamos considerar usar la relación de generalización. Por ejemplo: un Tigre es un Animal.



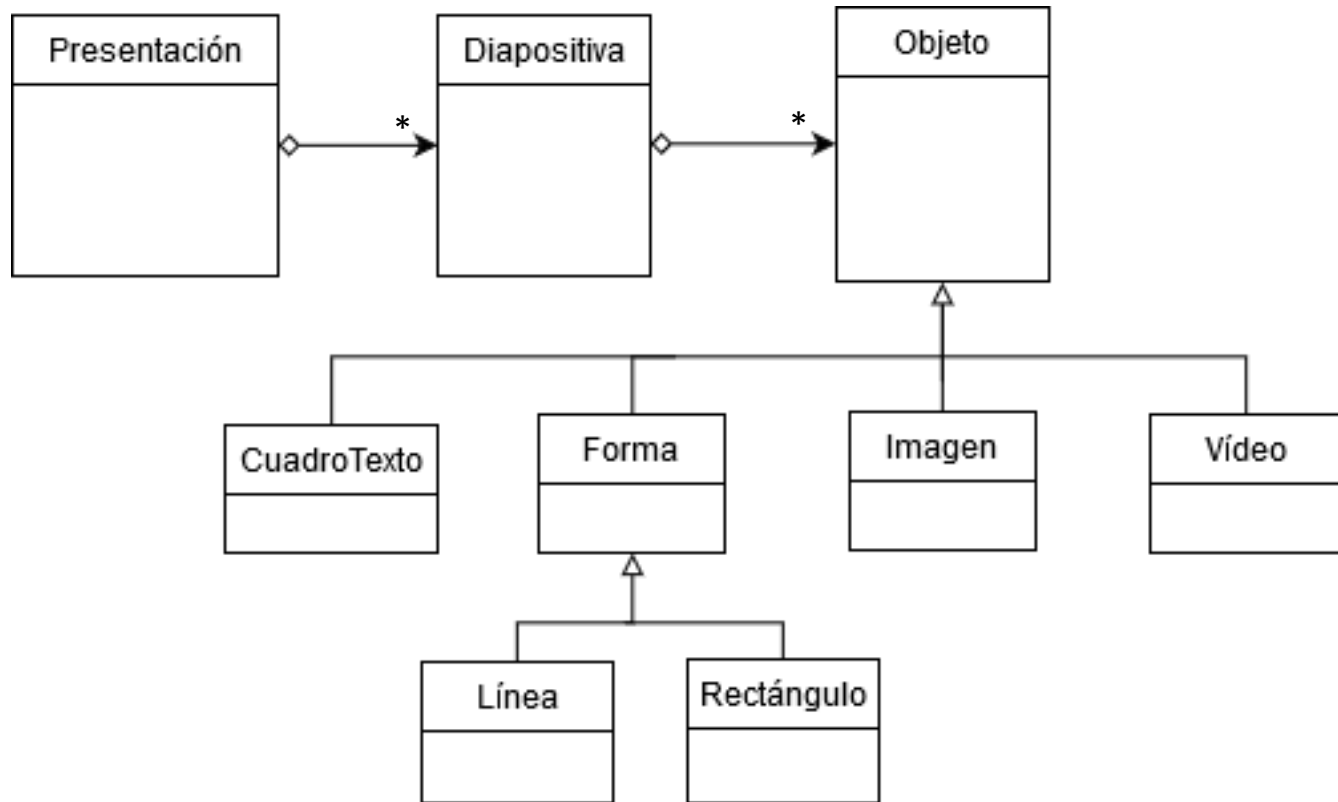
Realización. Este concepto se refiere a la implementación de un interfaz por parte de una clase. Este proceso tiene dos partes:

- a) En primer lugar la clase debe declarar la implementación del interfaz (`implements`).
- b) En segundo lugar la clase debe de definir el cuerpo de los métodos impuestos por la interfaz.



Por ejemplo, la clase Zapato implementa el interfaz IMartillo que permite al zapato utilizarlo como “Martillo”.

PowerPoint. Algunas Clases de Modelo y sus relaciones



RECORDATORIO DE SWING

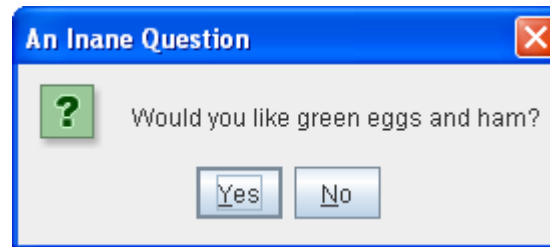
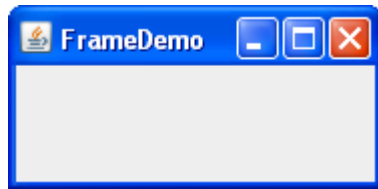
Swing es una biblioteca de interfaces gráficas de usuario (GUI) para Java.

Además del paquete `java.awt`, Java dispone del paquete `javax.swing` para crear interfaces gráficas.

Se dispone de una serie de Clases y de Interfaces, muchas de las cuales están disponibles en `awt` (excepto por la J inicial: `JFrame`, `JPanel`, etc.). Dichas Clases funcionan como `awt` pero con muchos más métodos y más útiles.

Contenedores: son componentes de las interfaces gráficas que pueden contener a otros componentes. Hay dos categorías:

- **Contenedores de alto nivel.** Dan origen a una interfaz gráfica y por tanto son la raíz de ella. Por ejemplo:
 - Ventanas → `JFrame`
 - Cuadros de Diálogo → `JDialog`
 - Applets → `JApplet`



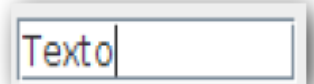
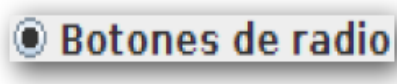
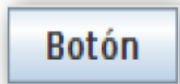
De izquierda a derecha: JFrame, JDialog, JApplet

- **Contenedores intermedios.** Van insertados un contenedor de alto nivel o de otro contenedor intermedio. Permiten estructurar la ubicación de los componentes de una GUI. Normalmente un contenedor de alto nivel irá seguido de uno o más contenedores intermedios.

Por ejemplo: Panel (JPanel), Barra de desplazamiento (JScrollBar), Barra de Herramientas (JToolBar), etc.

- **Componentes básicos:** le dan funcionalidad a una GUI.

Por ejemplo: *JButton, JLabel, JList, JComboBox, JRadioButton, JCheckBox, JTable, JPasswordField, JTextArea, JPasswordField.*



REFERENCIAS

Russ Miles, Kim Hamilton – *Learning UML 2.0*. O'Reilly Media (2006).

Eric Freeman, Elizabeth Freeman, Kathy Sierra, Bert Bates – *Head First Design Patterns*. O'Reilly (2008).

<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

<https://docs.oracle.com/javase/tutorial/uiswing/components/toplevel.html>