

CFGS Desarrollo de aplicaciones web

Módulo profesional: Programación



**GENERALITAT
VALENCIANA**

Conselleria d'Educació,
Investigació, Cultura i Esport



Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Edu Torregrosa Llácer

Modificado por Joan Carrillo

Esta obra está licenciada bajo la licencia **Creative Commons
Atribución-NoComercial-CompartirIgual 4.0 internacional**. Para ver una
copia de esta licencia visita:
<https://creativecommons.org/licenses/by-nc-sa/4.0/>



**Attribution-NonCommercial-ShareAlike
4.0 International (CC BY-NC-SA 4.0)**

P00 en JAVA



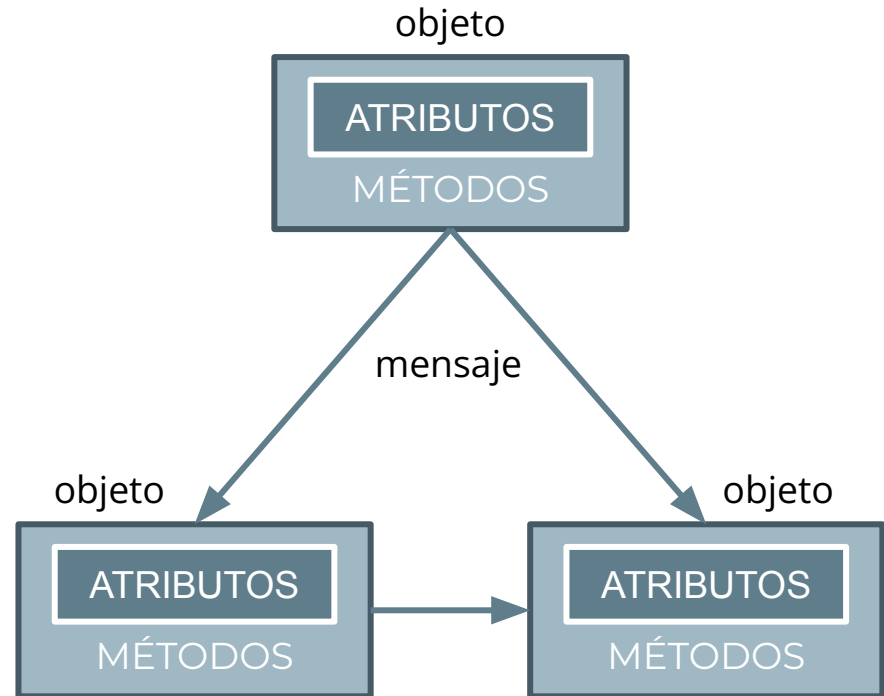
1. Elementos y características de la POO
2. Resolución de problemas con OO
3. Características de la POO
4. Diseño OO - UML
5. POO en Java
 1. Clases, instancias y objetos
 2. Constructores, métodos y this
 3. Visibilidad: public, private y protected
 4. Elementos estáticos (static)
 5. Clases inmutables y referencias de objetos.
 6. Arrays y ArrayList de objetos

Introducción

- Para empezar, todo parte del hecho de que el desarrollo de la programación de computadoras entró en crisis en los años 60 y 70 del s. XX
- Englobó a una serie de sucesos que se venían observando en los proyectos de desarrollo de software:
 - Los proyectos no terminaban en plazo.
 - Los proyectos no se ajustaban al presupuesto inicial.
 - Baja calidad del software generado.
 - Software que no cumplía las especificaciones.
 - Código inmantenible que dificulta la gestión y evolución del proyecto.
- Por todo ello surge un nuevo paradigma con el objetivo de resolver muchos de los problemas del desarrollo de SW. Surge la programación OO

Introducción

- La **programación orientada a objetos** gira alrededor del concepto de **objeto**.
- Así un **objeto** es una entidad que tiene unos **atributos** particulares, los datos, y unas formas de operar sobre ellos, los **métodos** o **procedimientos**.
- Durante la ejecución, los objetos reciben y envían **mensajes** a otros objetos para realizar las acciones requeridas.



Introducción

- La **POO** es una manera de diseñar y desarrollar software que trata de **imitar** la **realidad** tomando algunos conceptos esenciales de ella.
- El principal concepto es el **objeto**, cuyas características son la identidad, el estado y el comportamiento.
 - La **identidad** es el nombre que distingue a un objeto de otro.
 - El **estado** son las características que lo describen.
 - El **comportamiento** es lo que puede hacer.

Introducción

- Se debe tener presente que **los objetos, se abstraen en clases.**
- Por ejemplo:
 - De la clase Persona pueden existir dos objetos Pepe y Marta (esta es su identidad).
 - Pepe es un hombre que vive en Valencia, trabaja de profesor, y tiene 45 años de edad; mientras que Marta es una mujer de 25 años que vive en Madrid y es periodista(este es su estado).
 - De ambas personas podemos extraer información, y modificar sus estado (éste es su comportamiento).

Introducción

- Si nos pidieran que hiciéramos un programa orientado a objetos que simulara lo anterior haríamos:
 - La **clase** Persona que tendría las **variables** nombre, edad, ciudad de residencia y profesión.
 - Los **métodos** podrían ser obtenerDatos(), cambiarProfesion(), modificarCiudadResidencia().
 - Pepe y Marta serían los **identificadores** que podríamos usar en una aplicación que pretenda mostrar dos objetos (instancias) de la clase Persona. Aunque también podríamos usar nombres más genéricos como Persona1 y Persona2.

Introducción

- Identificadores:

Son los nombres que pueden tener las clases, los métodos y las variables y no pueden contener espacios ni caracteres especiales. Aunque no es obligatorio (el código funcionará igual), estos nombres deben respetar ciertas convenciones según el tipo de identificador:

Tipo de identificador	Convención	Ejemplo
Clase	Comienza con mayúscula	HolaMundo
Método	Comienza con minúscula	mostarSaludo()
Variable	Comienza con minúscula	saludo

Elementos básicos de la P00

1. Clases
2. Atributos
3. Métodos
4. Mensajes
5. Instanciación

Elementos básicos de la P00

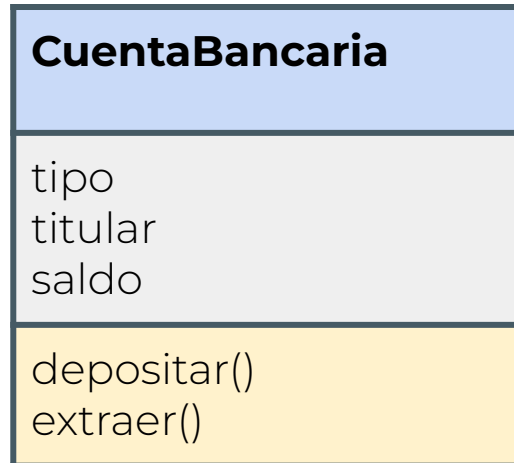
- **Clase:**

- Una clase es algo abstracto que define la "forma" del objeto, se podría hablar de la clase como el **molde de los objetos**.
- En el mundo real existen objetos del mismo tipo, por ejemplo tu bicicleta es solo una mas de todas las bicicletas del mundo. Entonces diríamos que tu bicicleta es una instancia de la clase Bicicleta.
- Todas las bicicletas tienen los **atributos**: color, cantidad de cambios, dueño y **métodos**: acelerar, frenar, pasar cambio, volver cambio.
- Las fábricas de bicicletas utilizan moldes para producir sus productos en serie, de la misma forma en POO utilizaremos la clase bicicleta (molde) para producir sus instancias (objetos).
- **Los objetos son instancias de clases.**

Elementos básicos de la P00

Clase (UML):

- Existe un lenguaje de modelado llamado UML mediante el cual podemos representar gráficamente todo un sistema orientado a objetos utilizando rectángulos, líneas y otro tipo de símbolos gráficos.
- Según UML, la clase "Cuenta Bancaria" se representará gráficamente como sigue:



Elementos básicos de la P00

Atributos

- Los atributos son las **características** individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades.
- Los atributos se guardan en **variables** denominadas de **instancia**, y cada objeto particular puede tener valores distintos para estas variables.
- Las variables de instancia, son **declaradas** en la clase, pero sus valores son fijados y cambiados en el objeto.

Métodos

- El **comportamiento** de los objetos de una clase se implementa mediante métodos.
- Un **método** es un conjunto de instrucciones que realizan una determinada tarea y son similares a las funciones de los lenguajes estructurados.

Elementos básicos de la P00

Mensajes

- La interacción entre objetos se produce mediante mensajes. **Los mensajes son llamadas a métodos de un objeto en particular.**
- Podemos decir que el objeto Persona envía el mensaje "retirar dinero" al objeto CuentaBancaria.
- Los mensajes pueden contener parámetros. Por ejemplo teniendo un método en la clase CuentaBancaria llamado "retirarDinero(double)" que recibe como parámetro la cantidad a retirar.
- Un mensaje está compuesto por los siguientes tres elementos:
 - **El objeto destino**, hacia el cual el mensaje es enviado.
 - **El nombre del método** a invocar.
 - **Los parámetros** solicitados por el método.

Elementos básicos de la P00

Instanciación:

- Podemos interpretar que una clase es el plano que describe cómo es un objeto de la clase, por tanto podemos entender que a partir de la clase podemos fabricar objetos. A ese objeto construido se le denomina instancia, y al proceso de construir un objeto se le llama **instanciación**.
- Cuando se construye un objeto es necesario dar un valor inicial a sus atributos, es por ello que existe **un método especial en cada clase, llamado constructor**, que es ejecutado de forma automática cada vez que es instanciada una clase.
- El constructor se llama igual que la clase y no devuelve ningún valor a través del **return**. La invocación al método constructor devolverá una instancia de dicho objeto

Elementos básicos de la P00

Ejemplo de clase:

Class Persona

Atributos

Nombre: Cadena;

Dirección: Cadena;

Fecha nacimiento: Fecha;

Teléfono: Número;

Métodos

mostrarDatos ();

modificarDireccion(Cadena);

CLASE Persona

ATRIBUTOS

Nombre

Cadena

Dirección

Cadena

Fecha nac

Fecha

Teléfono

Número

MÉTODOS

mostrarDatos()

modificarDireccion(Cadena)

Características de la P00

Características de la P00

- Son **4 las características básicas** que debe cumplir un **objeto** para denominarse como tal.
- Estas características son:
 - **Abstracción.**
 - **Encapsulamiento.**
 - **Herencia.**
 - **Polimorfismo**
- Estas características de POO nos van a permitir:
 - Aislar cada componente del resto de la aplicación.
 - Aprovechar nuestro esfuerzo, en su buen funcionamiento.
 - Controlar cada uno de los objetos,
 - Desarrollar código más breve y conciso
 - Reutilizar el código escrito.

Características de la P00

Abstracción:

- Es la capacidad de un objeto de cumplir sus funciones independientemente del contexto en el que se utilice.
- Ejemplo un objeto “Persona” siempre expondrá sus mismas propiedades y dará los mismos resultados a través de sus eventos, sin importar el ámbito en el cual se haya creado.

Encapsulamiento:

- Esta característica es la que denota la capacidad del objeto de responder a peticiones a través de sus **métodos** sin la necesidad de exponer los medios utilizados para llegar a brindar estos resultados.
- O sea, el método obtenerDatos() del objeto “**Persona**” antes mencionado, siempre nos va a mostrar los datos de una Persona, sin necesidad de tener conocimiento de cuáles son los recursos que ejecuta para llegar a brindar este resultado.

Características de la P00

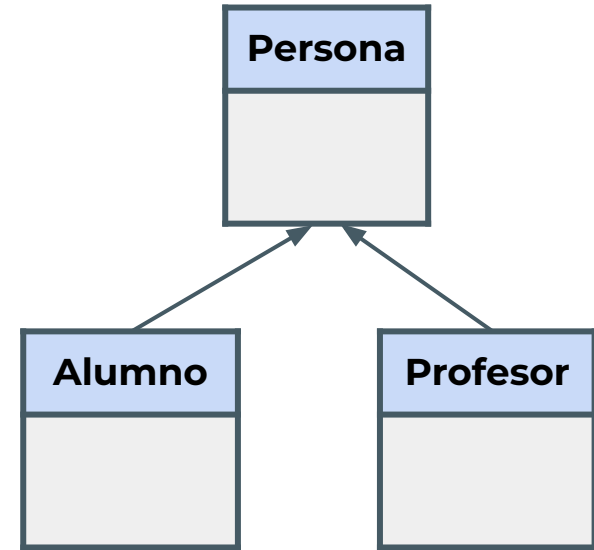
Herencia:

Es la característica por la cual los objetos para su creación se basan en una clase de base, heredando todas sus propiedades, métodos y eventos; los cuales a su vez pueden o no ser implementados y/o modificados.

Podemos crear la **clase Alumno** que hereda todos los atributos y métodos de la clase Persona, pero dicha clase, además puede incluir otros específicos.

Por ejemplo, del alumno puedo guardar datos adicionales y distintos a los de un profesor. Pero ambos pueden compartir datos en común, ya que ambos son personas. Ambos tienen un nombre y una edad. Sin embargo, el Alumno tiene un NIA y el profesor no.

En la terminología orientada a objetos "Alumno" y "Profesor" son subclases de la clase Persona. De forma similar Perro es la superclase de "Alumno".



Características de la POO

Herencia:

- Cada subclase hereda los atributos de la superclase. Tanto la clase "Alumno" como "Profesor" tendrán los atributos nombre, edad, teléfono, dirección, etc.
- Una subclase no está limitada únicamente a los atributos de su superclase, también puede tener atributos propios, o redefinir algunos definidos anteriormente en la superclase.
- No se está limitado tampoco a un solo nivel de herencia, se pueden tener todos los que se consideren necesarios.
- Gracias a la herencia, los programadores pueden reutilizar código una y otra vez.

Polimorfismo:

- Polimorfismo significa que la misma **operación** puede **comportarse diferentemente** sobre distintas clases.
- Por ejemplo, la operación "mover" puede comportarse diferentemente sobre una clase llamada Ventana y una clase llamada PiezasAjedrez.

D00 en JAVA

Introducción

- La orientación a objetos es una metodología con la que es posible resolver problemas mediante su descomposición en los elementos fundamentales que los componen y la especificación de cómo interactúan.
- En esta unidad se presenta una introducción general a esta metodología, adoptando la perspectiva del diseñador de SW. Principalmente por dos motivos:
 - Poder aplicar los conocimientos adquiridos sin atarnos a ningún lenguaje de programación concreto.
 - Plasmar gráficamente el diseño de software con el lenguaje **UML**
- **La orientación a objetos es una metodología**, con vistas al desarrollo del software. No se trata simplemente de una familia de lenguajes de programación

Introducción

- El UML se utiliza para establecer cómo se estructura la resolución de un problema mediante la orientación a objetos y de qué manera interactúan los diferentes componentes para lograr una tarea concreta.
- Estas bases son las siguientes:
 - Todo es un objeto, con una identidad propia.
 - Un programa es un conjunto de objetos que interactúan entre ellos.
 - Un objeto puede estar formado por otros objetos más simples.
 - Cada objeto pertenece a un tipo concreto: una clase.
 - Objetos del mismo tipo tienen un comportamiento idéntico.

UML

- **Unified Modelling Language** o Lenguaje de Modelado Unificado.
- Se utiliza para describir formalmente el diseño de una aplicación.
- Los orígenes del UML se remontan al año 1994, cuando Grady Booch y Jim Rumbaugh comenzaron a unificar diferentes técnicas de modelización orientada a objetos.
- El **UML** es un lenguaje estándar que permite especificar con notación gráfica software orientado a objetos.

Relaciones entre clases

- Una vez identificadas las clases de los objetos que componen el problema resolver, el siguiente paso es establecer qué relaciones hay entre ellas. Cada relación indica que hay una conexión entre los objetos de una clase y los de otra.
- **El tipo de relación más frecuente es la asociación.** Se considera que existe una asociación entre dos clases cuando se quiere indicar que los objetos de una clase pueden llamar operaciones sobre los objetos de otra.



Relaciones entre clases

- Dada una asociación, se debe especificar:
 - **En el centro, el nombre de la asociación.**
 - **Con una flecha se especifica la navegabilidad.** Partiendo del nombre de la asociación y los métodos de las clases, se debe poder establecer cuál es la clase **origen** y cuál el **destino**.
 - **La navegabilidad indica el sentido de las interacciones entre objetos:** a qué clase pertenecen los objetos que pueden llamar operaciones y a qué clase los objetos que reciben estas llamadas.

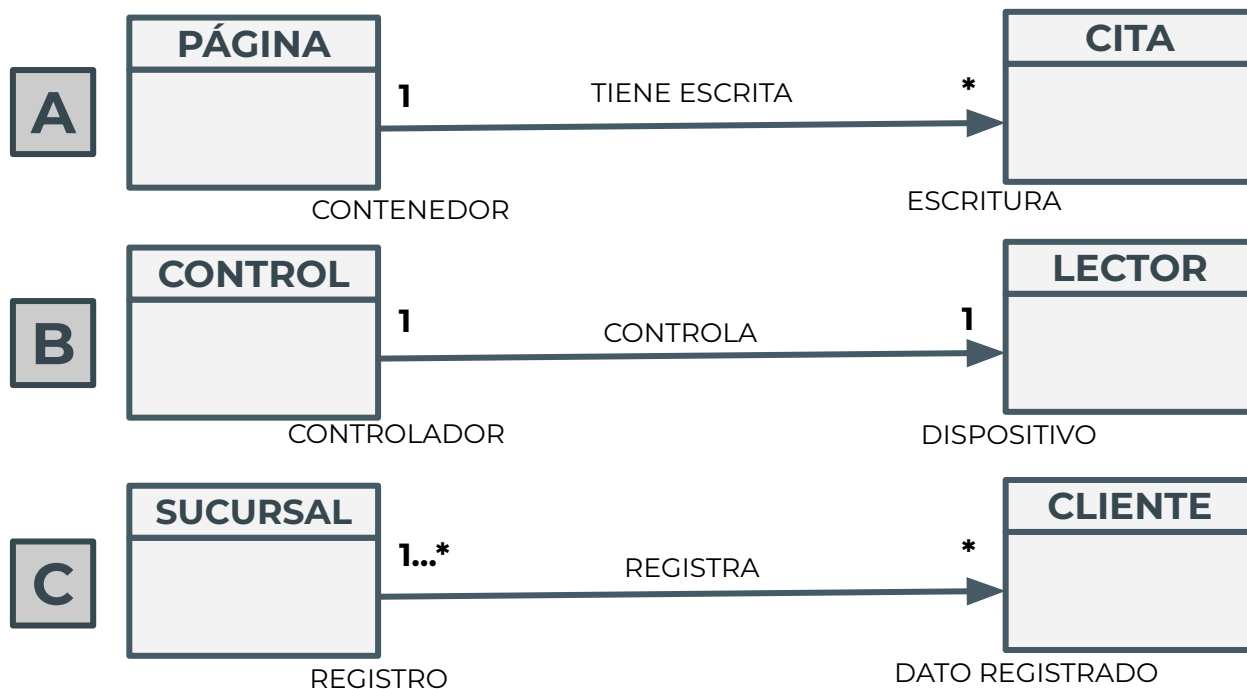
Relaciones entre clases

- En cada extremo se especifica la cardinalidad.
- La cardinalidad especifica con cuantas instancias de una de las clases puede estar **enlazada una instancia de la otra clase** en un momento determinado de la ejecución del programa.
- Para establecer rangos de valores posibles, **se usan los límites inferior y superior separados con tres puntos**.

Diferentes cardinalidades y su significado	
1	Sólo un enlace
0...1	Ninguno o un enlace
2,4	Dos o cuatro enlaces
1...5	Entre 1 y 5 enlaces
1...*	Entre 1 y un número indeterminado, es decir, más de 1
*	Un número indeterminado. Es equivalente a 0...*

Relaciones entre clases

- La navegabilidad y la cardinalidad son imprescindibles ya que la decisión que se tome en estos aspectos dentro de la etapa de diseño tendrá implicaciones directas sobre la implementación.



Relaciones entre clases

- Una instancia cualquiera de la clase Página puede enlazar hasta un número indeterminado de objetos diferentes de la clase Cita.
- Dado un objeto cualquiera de la clase Cita, sólo estará enlazado a un único objeto Página. Por lo tanto, no se puede tener una misma cita en dos páginas diferentes (pero sí tener dos citas diferentes y de contenido idéntico, con los mismos valores para los atributos, cada una a una página diferente).
- Tampoco puede haber citas que, a pesar de ser en la aplicación, no estén escritas en ninguna página.



Relaciones entre clases

- Un objeto de la clase Control siempre tiene un objeto de la clase Lector enlazado. Por tanto, un panel de control siempre controla un único lector de media.
- No se puede dar el caso de que un panel de control no controle ningún lector. La inversa también es cierta, todo lector es controlado por algún panel de control.
- El panel de control puede llamar operaciones sobre el lector, pero no al revés. Esto tiene sentido, ya que es el panel de control que controla el lector.



Relaciones entre clases

- Dado un cliente, éste puede estar registrado en más de una sucursal, pero al menos siempre lo estará en una. Nunca se puede dar el caso de un cliente dado de alta en el sistema pero que no esté registrado en ninguna sucursal.

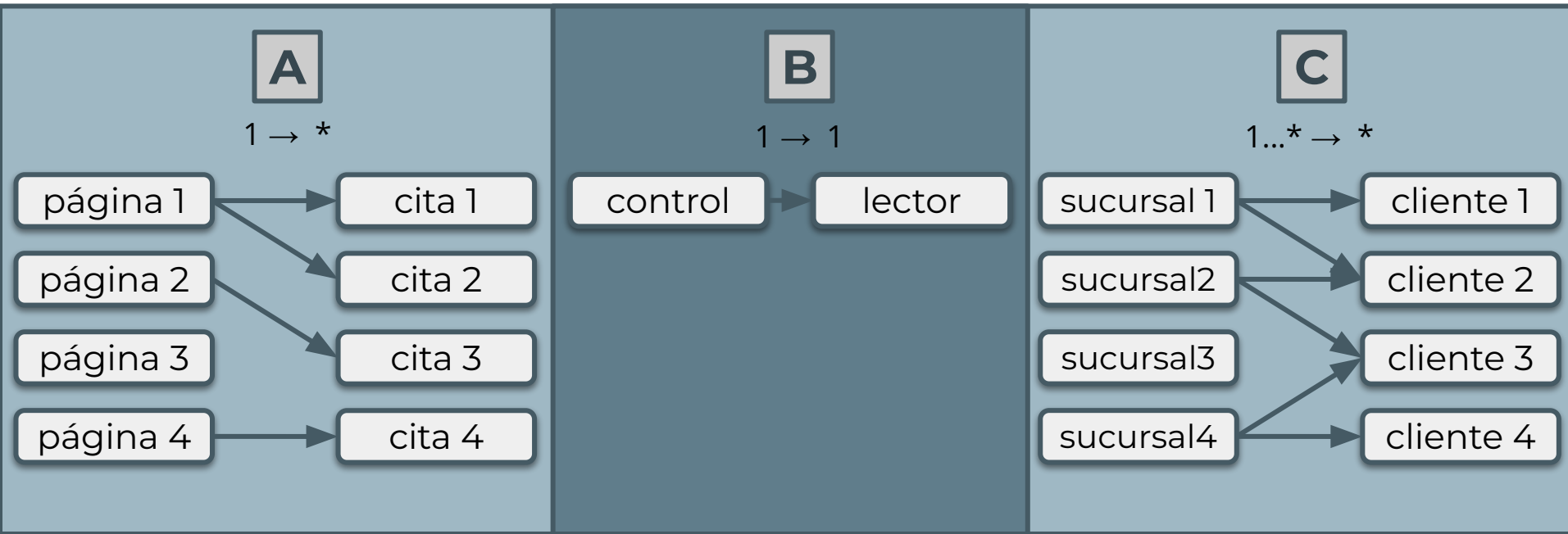


Mapa de objetos

- Se utilizan para reflexionar sobre si una cardinalidad representa lo que el diseñador quiere.
- Se trata de esquemas que representan los diferentes estados posibles de la aplicación.
- Los mapas de objetos sólo son una herramienta de apoyo, y no se utilizan como mecanismo formal para representar el diseño.
- En un **mapa de objetos** se muestran todos los objetos instanciados y los enlaces que hay entre ellos en un momento determinado de la ejecución, la aplicación de acuerdo con lo que se ha representado en el diagrama estático UML.
 - Dado un cliente, éste puede estar registrado en más de una sucursal, pero al menos siempre lo estará en una. Nunca se puede dar el caso de un cliente dado de alta en el sistema pero que no esté registrado en ninguna sucursal.

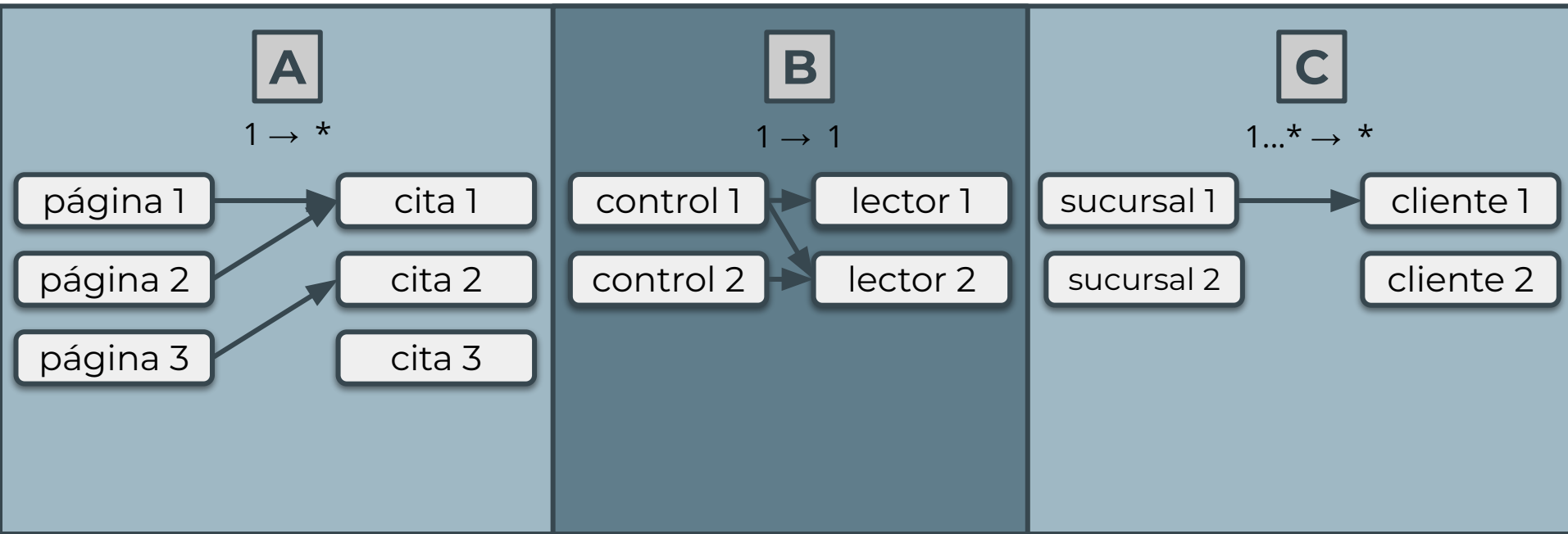
Mapa de objetos

- La figura representa una serie de mapas de objetos, uno por cada caso, con diferentes objetos enlazados correctamente según la cardinalidad especificada en la asociación.
- Los enlaces se representan con flechas según la navegabilidad de las asociaciones.



Mapa de objetos

- La figura muestra algunos casos de estados que se considerarían incorrectas según las cardinalidades especificadas en las asociaciones.



Agregación

- Asociación especial mediante la cual los objetos de cierta clase forman parte de los objetos de otra.
- En el diagrama estático UML, esto se representa gráficamente añadiendo un rombo blanco en el extremo de la asociación donde está la clase que representa el todo.
- Como con este símbolo ya se dice cuál es la relación entre los objetos de ambas clases, se pueden omitir el nombre y la función en los descriptores de la asociación.



- Una página contiene citas escritas en su interior y se puede considerar que lo escrito en una página es parte de la misma
- Expresa **contiene o es parte de**.

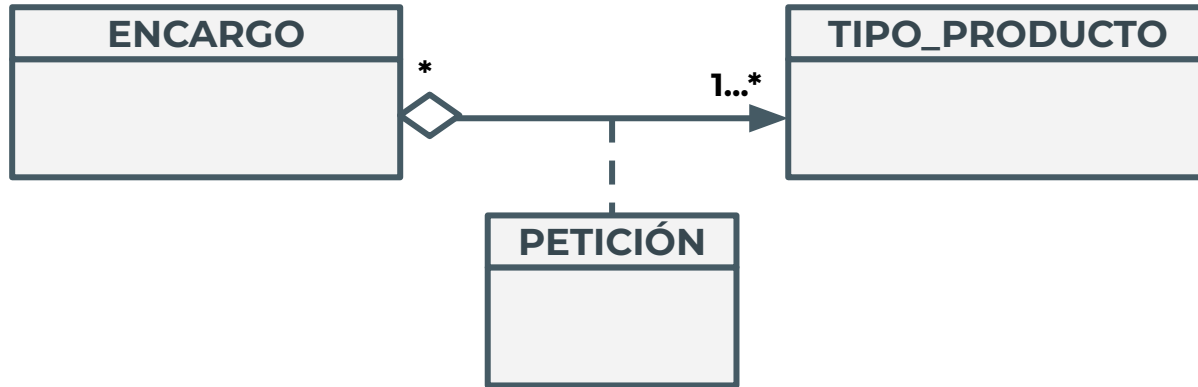
Composición

- Expresa el concepto de **es parte de**, pero la clase compuesta no tiene sentido sin sus componentes.
- En contraposición, en una agregación, el agregado sí tiene sentido sin ninguno de sus componentes.
- Una composición es una forma de agregación que requiere que los objetos componentes sólo pertenezcan a un único objeto agregado y que, además, este último no exista si no existen los componentes.
- Cualquier asociación en que el diseñador pondría un verbo de el estilo tiene, contiene, es parte de, etc. es una agregación o una composición.
- Este tipo de asociación se representa de forma idéntica a una agregación, sólo que en este caso el rombo es de color negro.
 - No tiene sentido una agenda sin páginas.
 - Tampoco puede ser que una misma página esté en más de una agenda.
 - En cambio, sí tiene sentido una página en blanco sin ningún cita, por lo que el caso Página-Cita es una agregación pero no una composición.



Clases asociativas

- Se utilizan cuando, el diseñador quiere especificar atributos en una asociación.
- Una clase asociativa se representa con el mismo formato que una clase: es una nueva clase que hay que especificar dentro de la descomposición del problema.
- Una línea discontinua une la nueva clase con la asociación que representa.



- Las clases asociativas representan asociaciones que se pueden considerar clases.
- Java no soporta directamente clases asociativas.

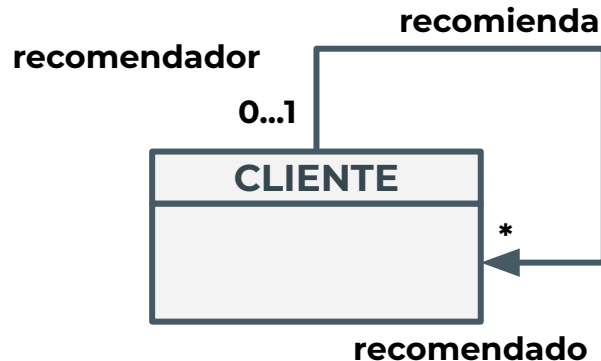
Clases asociativas

- Una particularidad de las clases asociativas es que se pueden representar con clases y asociaciones normales. Este hecho es importante, ya que es la única manera de representarlas en un mapa de objetos y la mayoría de lenguajes de programación no soportan las asociaciones con este tipo de clases vinculadas.
- El desarrollo es el siguiente:
 - Eliminar la asociación original.
 - Generar una asociación de la clase origen a la clase asociativa. El tipo de la nueva asociación y la navegabilidad son idénticos al original.
 - Generar otra asociación de la clase asociativa al destino. La navegabilidad es de la clase asociativa en destino.



Asociación reflexiva

- Una asociación reflexiva es aquella en que la clase origen y el destino son la misma.
- No se puede aplicar una asociación reflexiva a una composición.
- Ocurre cuando hay una asociación entre instancias de una clase.
- Un ejemplo de este caso se muestra en la figura, en el que los clientes de la aplicación de gestión recomiendan otros clientes. Se trata de una asociación reflexiva, ya que tanto quien recomienda como quien es recomendado, un cliente, pertenecen a la misma clase.



Implementación de asociaciones en JAVA

- Las asociaciones entre clases se captan mediante atributos codificados en la clase origen de la relación, el tipo de los cuales es la clase destino.
 - Habrá tantos atributos de cada tipo como la cota superior de la cardinalidad en el extremo destino de la relación.
 - Si cardinalidad es 2 habrán 2 atributos en clase origen.
 - Si cardinalidad 1, sólo se codificaría un único atributo.
 - Las cardinalidades indeterminadas (caso *), se suele usar la clase parametrizada ArrayList, y se aplica como parámetro la clase en el extremo opuesto de la relación.

Implementación de asociaciones en JAVA

- Las operaciones definidas en cada clase se implementan mediante la definición de métodos en el código fuente de las clases.
 - Cada método contiene el conjunto de instrucciones del lenguaje de programación necesarias para efectuar la tarea asociada.
 - Aunque muchas veces se usan los términos operación y método indistintamente, formalmente describen cosas diferentes.

Implementación de asociaciones en JAVA

- Las operaciones de clase tienen la siguiente sintaxis:

visibilidad nomOperació (llistaParàmetres): tipusRetorn

- Las operaciones de clase no están vinculadas a objetos y, por tanto, no se pueden llamar sobre ellos, ya que no tienen ámbito sobre los atributos de ningún objeto concreto.
- En cambio, una operación de clase sí que puede manipular atributos de clase.
- Este tipo de operación se suele usar para tareas de propósito general a las que se debe acceder directamente desde cualquier objeto dentro del programa (de la clase que sea) o para manipular fácilmente atributos de clase.
 - ejemplo: contarObjetosInstanciados (): entero
 - Este es un caso en que puede tener sentido disponer de una función fácilmente accesible desde cualquier clase

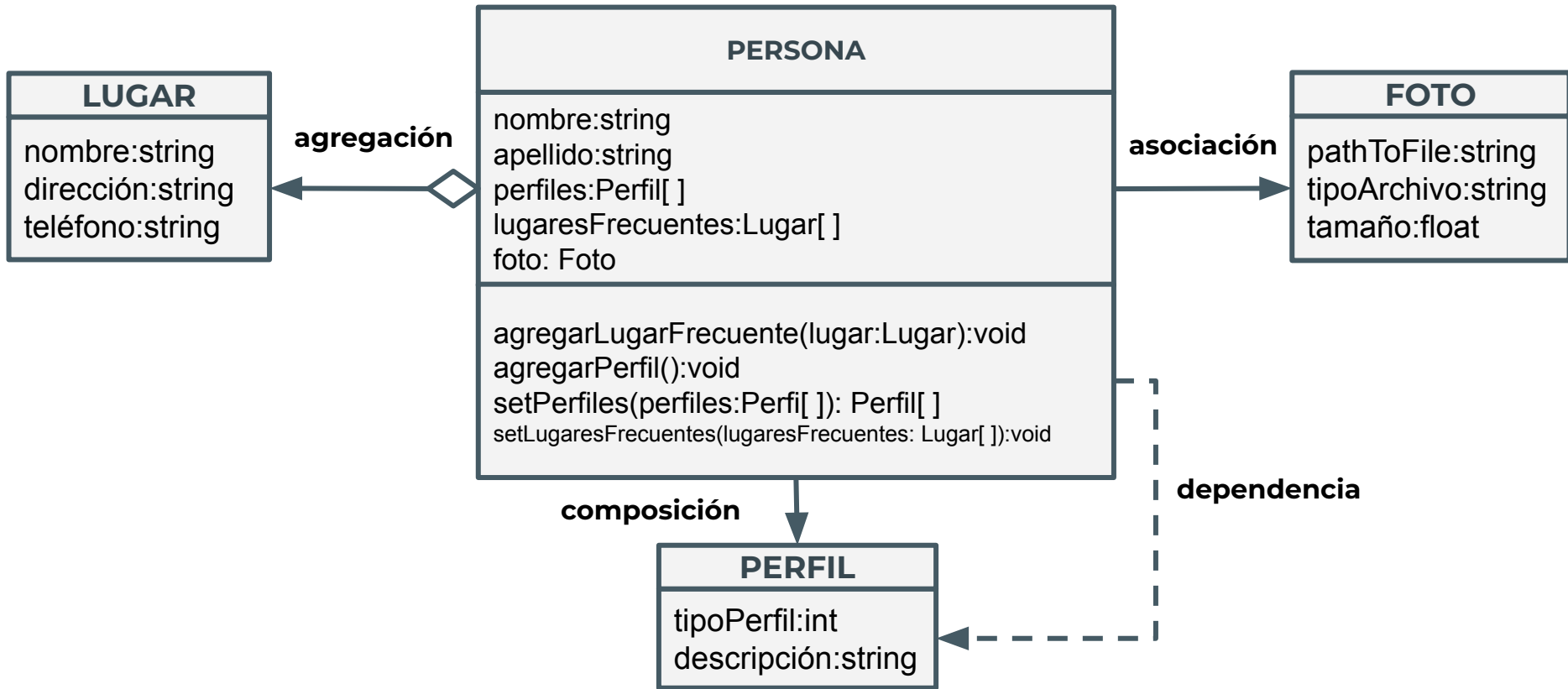
Implementación de asociaciones en JAVA

- Ubicación de operaciones
 - Uno de los momentos que puede resultar dificultoso dentro de todo el proceso de diseño es decidir qué operaciones hay que especificar en cada clase.
 - Uno de los principios a seguir en ubicar operaciones es el de la cohesión: obtener al final del diseño clases con una cierta coherencia y que aporten una idea muy clara de cuál es su papel dentro del problema que se está descomponiendo.
 - Las diferentes clases del diseño deben coexistir en armonía y cada una debe tener un objetivo muy claro.
 - Sólo hay que asignar los atributos y las operaciones mínimas e imprescindibles para realizar esta tarea exclusivamente.
 - Otro aspecto importante para lograr una correcta cohesión es asignar las operaciones en la clase correcta.

Ejemplo 1: Perfiles

Ejemplo: Perfiles

- Tenemos el siguiente diagrama de clases:



Ejemplo: Perfiles

- Una ASOCIACIÓN es una relación entre objetos de diferentes clases.

```
public class Persona {  
    String nombre;  
    String apellidos;  
    ...  
    Foto foto;  
}
```

- En el ejemplo una persona tiene una foto, pero la Vida de la persona no depende de la foto ni viceversa.
- Se dibuja una flecha de Persona->Foto, pues es la Persona la que tiene la relación.
- Si pudiera ser bidireccional se representaría con una recta

Ejemplo: Perfiles

- Una ASOCIACIÓN se implementa en Java introduciendo referencias a objetos en una clase como atributos en la otra
 - Si la relación tiene una cardinalidad 1,2... se define un o(dos) atributos de esa clase.
 - Si la relación tiene una cardinalidad superior a dos entonces será necesario utilizar un Array de referencias.
 - También es posible utilizar un ArrayList para almacenar las referencias.
 - Normalmente la conexión entre los objetos se realiza recibiendo la referencia de uno de ellos en el constructor o una operación ordinaria del otro

Ejemplo: Perfiles

```
public class Persona {
```

```
    private String nombre;
```

```
    private String apellido;
```

```
    private List perfiles = new ArrayList();
```

```
    private List lugaresFrecuentes = new ArrayList();
```

```
    //Setters and Getters
```

```
    public String getNombre() {return nombre;}
```

```
    public void setNombre(String nombre) { this.nombre = nombre; }
```

```
    public String getApellido() { return apellido; }
```

```
    public void setApellido(String apellido) {this.apellido = apellido;}
```

Ejemplo: Perfiles

- Una AGREGACIÓN es un tipo especial de asociación donde se añade el matiz semántico de que la clase de donde parte la relación representa el “todo” y las clases relacionadas “las partes”
- En nuestro caso una persona puede visitar muchos lugares, pero los lugares pueden existir con independencia de que hayan personas que los visitan.

Ejemplo: Perfiles

- Ahora veamos cuales son los métodos que caracterizan a la relación de

AGREGACIÓN

```
public void agregarLugarFrecuenta(Lugar lugar){  
    if(!lugaresFrecuentes.contains(lugar)){  
        lugaresFrecuentes.add(lugar);  
    }  
}  
  
public void removerLugarFrecuenta(Lugar lugar){  
    if(lugaresFrecuentes.contains(lugar)){  
        lugaresFrecuentes.remove(lugar);  
    }  
}
```

Ejemplo: Perfiles

- Ahora veamos cuales son los métodos que caracterizan a la relación de AGREGACIÓN
- Los objetos son pasados por parámetro, no han sido instanciados dentro del método, es decir no hemos realizado el new del objeto. Ha "nacido" en cualquier otra parte y se lo hemos pasado por parámetro al método para ser agregado a la lista lugaresFrecuentes.
- El objeto Persona podría morir, y el objeto aún podría mantener una referencia activa en alguna otra parte de nuestro código por lo tanto sus ciclos de vida no estarían atados.
- No nace ni muere, dentro de la Persona.

Ejemplo: Perfiles

- La composición es un tipo de agregación que añade el matiz de que la clase “todo” controla la existencia de las clases “parte”.
- La clase “todo” creará al principio las clases “parte” y al final se encargará de su destrucción.
- La clase Persona creará el objeto perfil dentro del método agregar y la referencia no se devuelve (es void o boolean), la variable de referencia local va a dejar de existir una vez que el método se termine de ejecutar
- El ciclo de vida de esa instancia en particular va a quedar atada a la lista, y por ende a la Persona.
- Como el método no es estático, se deberá crear primero una instancia de Persona, para después poder agregar un Perfil. Empezando así a "atar" el ciclo de vida de un Perfil, al de una Persona.

Ejemplo: Perfiles

- ¿Cual es la Diferencia con la **COMPOSICIÓN**?

```
public void agregarPerfil(){
    Perfil p = new Perfil();
    perfiles.add(p);
}
//sobrecarga
public void agregarPerfil(String nombre){
    Perfil p = new Perfil(nombre);
    perfiles.add(p);
}
public void removerPerfil(int index){
    perfiles.remove(index); // aquí lo quitamos de la lista
}
```

Ejemplo: Perfiles

- ¿Cuál es la diferencia con la **COMPOSICIÓN**?
 - La composición también tiene los métodos para agregar y borrar. Pero...el new del objeto se crea dentro del método agregar“
 - La instanciación del objeto p se realiza dentro del método constructor y la referencia no se devuelve (es void o boolean).
 - Una vez que el objeto Persona no se reference más, el objeto lista, quedará sin referencia, y por lo tanto sus elementos también.
 - Además como el método no es estático, se deberá crear primero una instancia de Persona, para después poder agregar un Perfil. Empezando así a "atar" el ciclo de vida de un Perfil, al de una Persona.

Ejemplo: Perfiles

- Una variable local de una clase es una composición.
- Un objeto pasado como parámetro en el constructor es agregación.

```
public class Aplicacion {  
    private Codigo codigo;  
    private BaseDeDatos bd;  
    //No puede haber Aplicación sin Codigo  
    public Aplicacion(Codigo codigo){  
        this.codigo=codigo;  
    }  
    //podria tener o no una base de datos;  
    public void setDatabase(BaseDatos bd) { ... }  
}
```

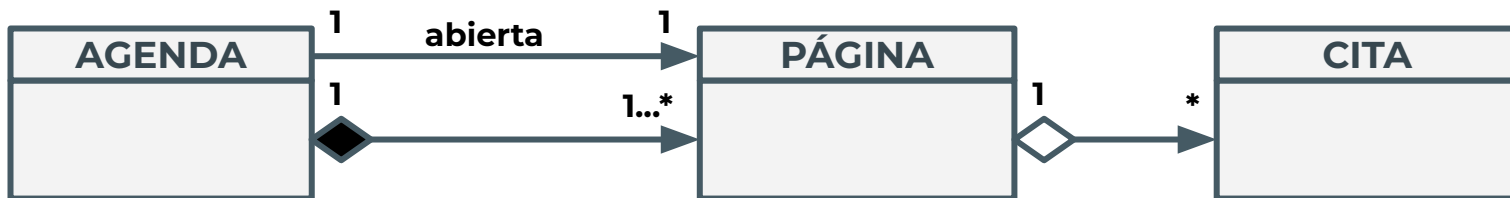

Ejemplo 2: Agenda

Ejemplo: Agenda

- Se quiere diseñar una agenda que permita consultar las fechas de un calendario para un año concreto y apuntar citas a unas horas concretas.
- Se puede pensar en la agenda como un libro en el que se van pasando páginas adelante o atrás, cada una de las cuales corresponde a un día.
- En cada página se pueden escribir citas establecidas para unas horas de inicio y de finalización determinadas.

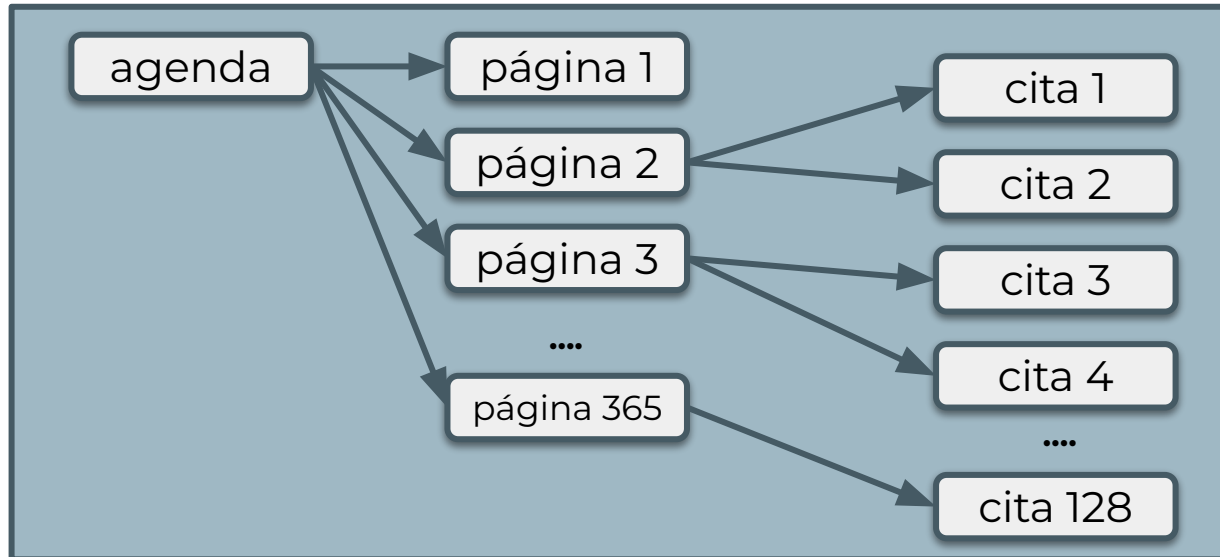
Ejemplo: Agenda

- La instancia de Agenda puede tener dos tipos de relaciones con las páginas.
- Por un lado, la de composición: entre todas las páginas forman la agenda.
- Además, una agenda también sabe por qué página está abierta, que sería la página que se puede leer en este momento.



Ejemplo: Agenda

- De este mapa de objetos se puede deducir, por ejemplo, que en este momento el usuario tiene un total de 128 citas apuntadas en la agenda.
- No tiene ninguna cita para el 1 de enero (la primera página de la agenda), y tiene dos para el día 2 y el 3. El 31 de diciembre tiene otra cita.



Ejemplo: Agenda

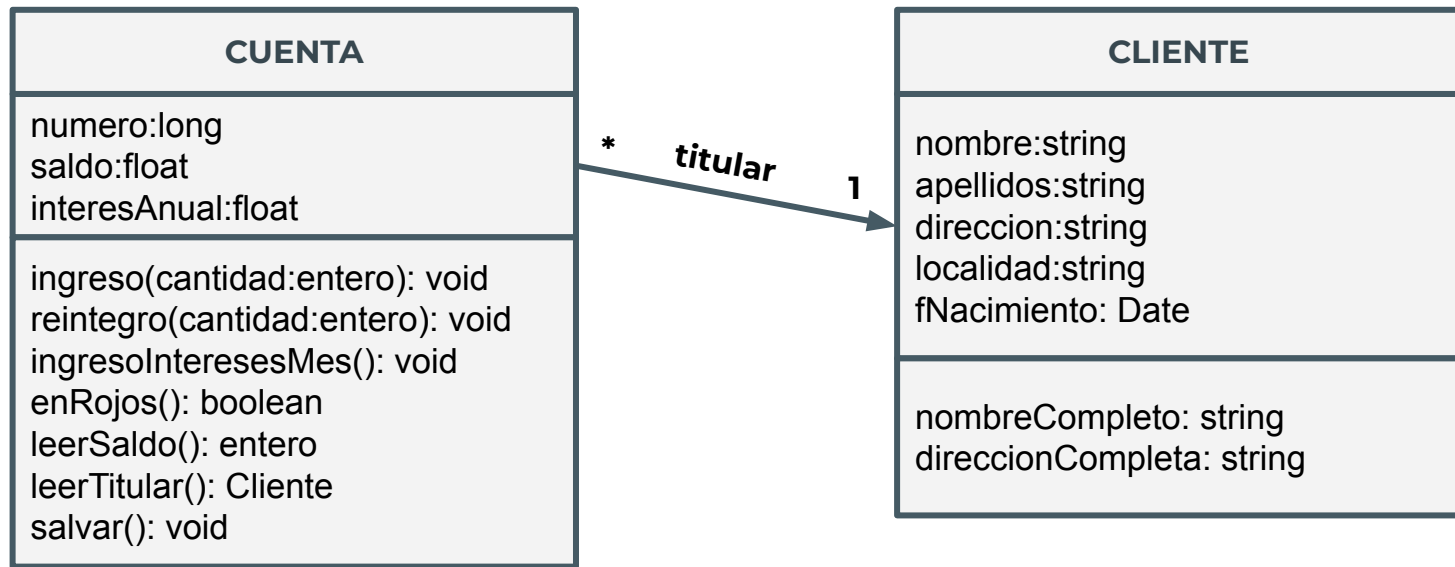
- La agenda. La figura presenta la especificación total de las clases de la aplicación de la agenda, con todos los atributos y operaciones.

AGENDA	PÁGINA	CITA
anyo:entero	dia:entero mes:entero	hora:entero minutos:entero titulo:string texto:string
avanzarPagina() : void retrocederPagina() : void leerPagina(): Pagina	agregarCita(c: Cita) : void borrarCita(c: Cita) : void buscarCita(título:string): Cita verCitas(): List<Cita>	modificarTexto(texto:string): void

Ejemplo 3: Cuentas

Ejemplo: Cuentas

- Partimos del siguiente diagrama de clases.



Ejemplo: Cuentas

- Se especifica la clase Cliente

```
public class Cliente {  
    private String nombre, apellidos;  
    private String direccion, localidad;  
    private Date fNacimiento;  
  
    Cliente (String aNombre, String aApellidos, String aDireccion,  
            String a Localidad, Date aFNacimiento) {  
        nombre = aNombre;  
        apellidos = aApellidos;  
        direccion = aDireccion;  
        localidad = alocalidad;  
        fNacimiento = aFNacimiento;  
    }  
    String nombreCompleto () { return nombre + " " + apellidos; }  
    String direccionCompleta () { return direccion + ", " + localidad; }
```

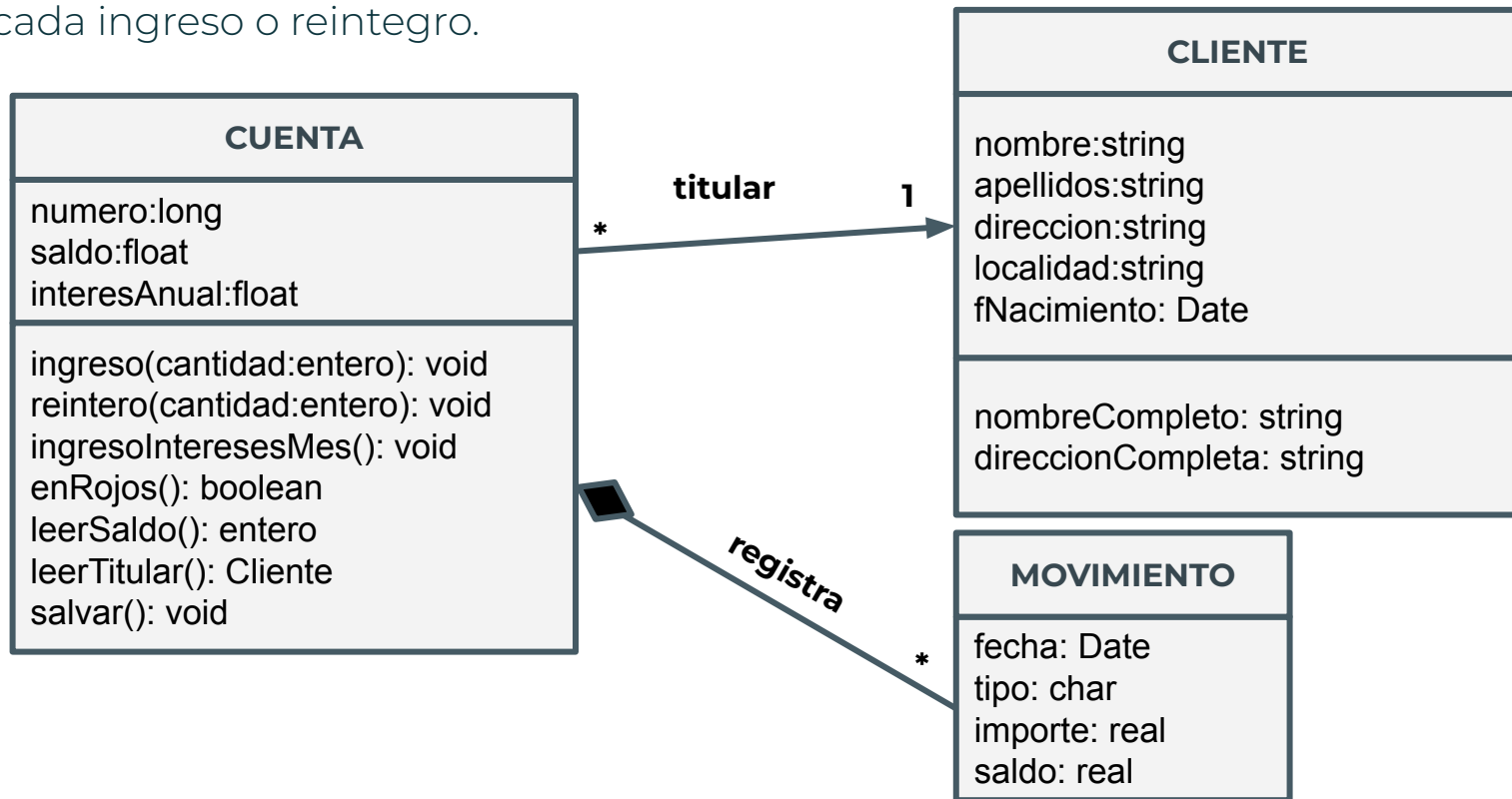

Ejemplo: Cuentas

- Se especifica la clase Cuenta:

```
public class Cuenta {  
    private long numero;  
    private Cliente titular;  
    private float saldo;  
    private float interesAnual;  
  
    // Constructor general  
    public Cuenta (long aNumero, cliente atitular, float aInteresAnual) {  
        numero = aNumero;  
        titular = aTitular;  
        saldo = 0;  
        interesAnual = aInteresAnual;  
  
        Cliente leerTitular() { return titular; }  
        // Resto de operaciones de la clase Cuenta a partir de aquí
```

Ejemplo: Cuentas

- Añadimos un registro de movimientos a la clase Cuenta, de forma que quede constancia tras cada ingreso o reintegro.



Ejemplo: Cuentas

- Se especifica la clase Movimiento:

```
import java.util.Date
class Movimiento
{
    Date fecha;
    char tipo;
    float importe;
    float saldo;

    public Movimiento (Date aFecha, char aTipo, float aImporte, float
aSaldo) {
        fecha = aFecha;
        tipo = a Tipo;
        importe = a Importe;
        saldo = a Saldo;
    }
}
```

Ejemplo: Cuentas

- Redefinimos la clase Cuenta

```
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo;
    private float interesAnual;
    private LinkedList movimientos; // Lista de movimientos

    // Constructor general
    public Cuenta (long aNumero, Cliente a Titular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = a InteresAnual;
        movimientos = new LinkedList(); }

    // Nueva implementación de ingreso y reintegro
    public void ingreso (float cantidad) {
        movimientos.add(new Movimiento (new Date(), 'I', cantidad, saldo += cantidad)); }

    public void reintegro (float cantidad) { movimientos.add(new Movimiento (new Date(), 'R', cantidad, saldo -=
        cantidad)); }

    public void ingreso Intereses () { ingreso (interesAnual * saldo / 1200); }
    // Resto de operaciones de la clase Cuenta a partir de aquí
```

Ejemplo: Cuentas

- Vamos a definir la clase Movimiento en el **interior** de Cuenta.
- Al ser declarada como privada, se impediría su utilización desde el exterior.

```
import java.util.Date
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo;
    private float interes Anual;
    private LinkedList movimientos; // Lista de movimientos

    static private class Movimiento {
        Date fecha;
        char tipo;
        float importe;
        float saldo;

        public Movimiento (Date aFecha, char aTipo, float a Importe, float aSaldo) {
            fecha = aFecha; tipo = aTipo; importe = a Importe; saldo = a Saldo; } }

    // Constructor general
    public Cuenta (long aNumero, cliente a Titular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0; interesAnual = a InteresAnual; movimientos = new LinkedList(); }

    // Resto de operaciones de la clase Cuenta a partir de aquí
```

Ejemplo: Cuentas

- Cuando la clase anidada no es estática, se denomina clase interior y tiene características especiales
 - Pueden ser creadas únicamente dentro de la clase continente
 - Tiene acceso completo y directo a todos los atributos y operaciones del objeto que realiza su creación
 - Los objetos de la clase interior quedan ligados permanentemente al objeto concreto de la clase continente que realizó su creación
 - No debe confundirse este elemento con la relación de composición, aunque en muchos casos es posible utilizar clases interiores para la implementación de este tipo de relaciones

Ejemplo: Cuentas

- Movimiento como una clase interior permite copiar el valor del saldo de la cuenta que realiza el movimiento

```
import java.util.Date
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo, interes Anual;
    private LinkedList movimientos; // Lista de movimientos

    private class Movimiento {
        Date fecha;
        char tipo;
        float importe, saldoMov;

        public Movimiento (Date aFecha, char a Tipo, float a Importe) {
            fecha = aFecha;
            tipo = aTipo;
            importe = a Importe;
            saldoMov = saldo; // Copiamos el saldo actual
        }
    }
}
// Sigue la implementación de la clase Cuenta
```

Ejemplo: Cuentas

- Se desea realizar una aplicación para manejar las cuentas de una sucursal bancaria. En la sucursal habrá:
 - Clientes que se almacenarán en un Array (máximo 5)
 - Cuentas que se guardarán en un Vector (Como máximo 10). En la creación se asignarán números de cuenta correlativos, saldo a 0, cliente a nulo e interés del 1%.
 - En cada Cuenta puede haber un número indeterminado de Movimientos que se almacenarán en un ArrayList.
 - Para trabajar con la aplicación siempre habrá una cuenta activa que es aquella en la que se realizan las operaciones.

Ejemplo: Cuentas

- La aplicación tendrá un menú principal :

MENÚ PRINCIPAL

- 1.- Mantenimiento de Clientes (Altas, Bajas, Modificaciones)
- 2.- Mantenimiento de Cuentas
- 0.- Salir

- Si el usuario selecciona la opción 1 se mostrará un submenú con las siguientes opciones:

CLIENTES

- 1.- Altas
- 2.- Bajas
- 3.- Modificaciones
- 4.- Listado

Ejemplo: Cuentas

- Si el usuario selecciona la opción 2, se solicitará que introduzca nombre y apellidos del cliente y se buscará en el Vector para ver si existe alguna Cuenta de ese Cliente.
 - Si existe se pondrá esa cuenta como activa y se mostrará el submenú.
 - Si no existe se avisará al usuario de que no tiene Cuenta y si quiere crearla.
 - En caso afirmativo se pone esa cuenta como activa y se muestra submenú

Ejemplo: Cuentas

- El submenú tiene las siguientes opciones:

1.- Ingresar (cantidad)

2.- Hacer reintegro (cantidad)

3.- Ingresar interés mensual

4.- En rojos

5.- Leer Saldo

6.- Datos titular

7.- Salvar

8.- Listar movimientos

P00 en JAVA

P00 en JAVA

- Java es un lenguaje **orientado a objetos** y programar en Java consiste en escribir las definiciones de las clases y utilizar esas clases para crear objetos de forma que representen correctamente el problema que queremos resolver.
- Las clases son **predefinidas** y **definidas** por el programador.
- En función de la estructura de la clase y del uso tenemos dos tipos básicos:
 - **Clase – Tipo de datos**: definen el conjunto de posibles valores que tomarán los objetos y las operaciones que se realizarán en estos.
 - **Clase – Programa**: son los que inician la ejecución del código, la clase que contiene el main

Ejemplo de clase

```
public class Circulo {  
    private double radio;  
    private String color;  
    private int centroX, centroY;  
    public Circulo() {                                //crea un círculo de radio 50, negro y centro en (100,100)  
        radio = 50;  
        color = "negro";  
        centroX = 100;  
        centroY = 100;  
    }  
    public double getRadio() {                        //consulta el radio del círculo  
        return radio;  
    }  
    public void setRadio(double nuevoRadio) {        //actualiza el radio del círculo a nuevoRadio  
        radio = nuevoRadio;  
    }  
    public void decrece() {                          //decrementa el radio del círculo  
        radio = radio / 1.3;  
    }  
    public double area() {                          //calcula el área del círculo  
        return Math.PI * radio * radio;  
    }  
    public String toString() {                      //obtiene un String con las componentes del círculo  
        return  
        "Círculo de radio " + radio + ", color " + color + " y centro (" + centroX + ", " + centroY + ")";  
    }  
}
```

Ejemplo de clase

```
public class PrimerPrograma {  
  
    public static void main(String[] args) {  
  
        // Crear un círculo con el constructor del círculo  
        Circulo c1 = new Circulo();  
        c1.setRadio(2.9);  
        System.out.println("Los datos del círculo: " + c1.toString());  
    }  
}
```

Estructura de una clase

- A continuación se especifica el esquema de definición de una clase:

```
[ámbito] class NombreDeLaClase {           [opcional ambito]clase  
    // Definición de atributos  
    [ámbito] tipo nombreVar1;  
    [ámbito] tipo nombreVar2;  
    .....  
    // Definición de métodos  
    // Constructores  
    ...  
    // Otros métodos  
}
```

atributos/características

metodos/comportamiento

Ámbito de declaración: private y public

- Toda la información declarada **private** es exclusiva del objeto e inaccesible desde fuera de la clase.
 - Cualquier intento de acceso a las variables de instancia radio o color que se realice desde fuera de la clase Circulo (p.e., en la clase PrimerPrograma) dará un error de compilación.

```
private double radio;  
private String color;
```

- Toda la información declarada **public** es accesible desde fuera de la clase.
 - En el caso de los métodos getRadio() o area() de la clase Circulo.

```
public double getRadio() {  
    return radio;  
}
```

Atributos de una clase

Los atributos o variables de instancia representan información de cada objeto de la clase y se declaran de un tipo, en la mayoría de ocasiones su acceso será privado.

```
// Definición de atributos  
[ámbito] tipo nombreVar1;  
[ámbito] tipo nombreVar2;
```

```
public class Circulo {  
    // Definición de atributos  
    private double radio;  
    private String color;  
    private int centroX, centroY; ...  
}
```

Métodos de una clase

- Los **métodos** definen las operaciones que se pueden hacer sobre los objetos de la clase y se describen indicando:
 - **La cabecera:** nombre, tipo de resultado y lista de parámetros necesarios para hacer el cálculo.
 - **El cuerpo:** contiene la secuencia de instrucciones necesarias.

```
public class Circulo {  
    ...  
    public double area() {  
        return 3.14 * radio * radio;  
    }  
}
```

Métodos de una clase

- Los clasificamos según su función:

- **Constructores:** permiten crear el objeto
- setters ○ **Modificadores:** permiten modificar el estado (valores de los atributos)
- getter ○ **Consultores:** permiten conocer, sin cambiar, el estado del objeto

```
public class Circulo {  
    //Constructor vacío  
    public Circulo() {    radio = 50; color = "negro"; }  
    //Constructor con 3 parámetros  
    public Circulo(double r, String c, int px, int py)  
    { radio = r; color = c; centroX = px; centroY = py; }  
    //Consultor: getter  
    public double getRadio() { return radio; }  
    //Consultor: setter  
    public void setRadio(double nuevoRadio) { radio = nuevoRadio; }  
}
```

P00 en JAVA

- **Tipos** de clases

- Debemos tener en cuenta que según la estructura de la clase y el uso que hagamos de ésta tenemos 2 tipos de clases:
 - **Clase - Tipo de datos:** definen el conjunto de posibles valores que pueden tomar los objetos y las operaciones que se pueden realizar sobre éstos.
 - **Clase - Aplicación:** son los que inician la ejecución del código.
- Así, en nuestro ejemplo tenemos la clase 'Television' que será del primer tipo y la clase aplicación que será del segundo tipo.

P00 en JAVA

- A esta clase le vamos a añadir un atributo llamado canal que va a ser de tipo int

```
public class Televisor { int canal; }
```

- A la Clase Televisor le vamos a añadir los métodos el constructor por defecto, subirCanal(), bajarCanal() y getCanal()

```
class Televisor {  
    int canal;  
    // El constructor por defecto, aunque no es necesario  
    public Televisor(){}  
    public void subirCanal() { canal++; }  
    public void bajarCanal() { canal--; }  
    public int getCanal()    { return canal; }  
}
```

P00 en JAVA

- Añadimos a la clase Aplicación el siguiente código:

```
Televisor tv;
```

- en este punto estamos declarando una variable de referencia tv. De momento su valor es null ya que todavía no apunta a ninguna Instancia u Objeto

```
tv = new Televisor();
```

- el operador new nos indica que se acaba de crear un nuevo Objeto, que es una Instancia de la Clase Televisor
- ahora la variable de referencia tv contiene la dirección de memoria de dicha Instancia

P00 en JAVA

- Para invocar los métodos de un Objeto, tenemos que tener primeramente una referencia a ese objeto y después escribir un punto "." y finalmente el nombre del método que queremos llamar.
- Este código se lo añadimos a la Clase **Aplicación**

```
tv.subirCanal();  
System.out.println("El canal seleccionado es el: " + tv.getCanal());  
tv.bajarCanal();  
System.out.println("El canal seleccionado es el: " + tv.getCanal());
```


P00 en JAVA

- Los Constructores se declaran de la siguiente forma:

```
nombreDelConstructor(listaDeParámetros){  
    cuerpoDelConstructor  
}
```

- **tipoValorDevuelto:** un Constructor no devuelve ningún valor, ni siquiera void.
 - **nombreDelConstructor:** el nombre del Constructor es el mismo que el nombre de la Clase.
- Y siguiendo la convención de nombres en Java, este nombre tendría que tener
 - la primera letra de la primera palabra compuesta en mayúsculas.
 - la primera letra de la segunda y restantes palabras compuesta en mayúsculas

P00 en JAVA

Sobrecarga de métodos: tenemos dos constructores uno sin argumento y el otro con un argumento de tipo int, es una de las tres formas de implementar el Polimorfismo.

```
public class Televisor {  
    int canal;  
    public Televisor() {};  
    public Televisor(int valorCanal) {  
        canal = valorCanal;  
    }  
}
```

P00 en JAVA

Ahora vamos a crear dos objetos de tipo Televisor. Uno de ellos estará referenciado por una variable de referencia llamada tv1 y el otro por otra variable de referencia llamada tv2, en clase **Aplicacion**.

```
public static void main(String[ ] args) {  
    ...  
    Televisor tv1 = new Televisor();  
    System.out.println("El canal por defecto es : "+ tv1.canal);  
    tv1.canal = 8;  
    System.out.println("El canal del primer televisor es el: " + tv1.getCanal());  
    Televisor tv2=new Televisor(6);  
    System.out.println("El canal del segundo televisor es el: " +tv2.getCanal());  
}
```

P00 en JAVA

- Encapsulación
 - A través de la Encapsulación se puede **controlar** qué partes de un programa pueden acceder a las variables y métodos de un Objeto.
 - La encapsulación se basa en el **control de acceso o ámbito**.
- Los ámbitos pueden ser:
 - **public** : puede ser accedido por cualquier parte del programa
 - **private** : sólo puede ser accedido por otros miembros de su Clase

P00 en JAVA

A la Clase Televisor le tenemos que hacer las siguientes modificaciones:

```
public class Televisor {
    private int canal;
    public Televisor() {}
    public Televisor(int valorCanal) {
        canal=valorCanal;}
    public void subirCanal() {
        setCanal(canal + 1); }
    public void bajarCanal() {
        setCanal(canal - 1); }
    public int getCanal() {
        return canal; }
    public void setCanal(int valorCanal) {
        if (valorCanal < 0){
            canal = 0;    }
        else {
            canal = valorCanal; }
    }}
}
```

P00 en JAVA

- **private int canal;**

- Al indicar que el ámbito es private estamos diciendo que sólo desde dentro de la Clase Televisor se puede acceder al atributo canal
- setCanal(canal + 1); setCanal(canal - 1);
- Cuando el usuario baja el canal, se podría dar el caso que llegara al canal 0 y si sigue bajando el canal llegaría al canal -1. Para evitar esto invocamos al método setCanal(...) pasándole como argumento el resultado de la operación de bajar el canal.
 - si más adelante nos dijeran que por ejemplo el canal más alto no puede superar el número 99, simplemente tendríamos que ampliar el filtro en el método setCanal(...)

- **public void setCanal(int valorCanal) { ... }**

- public, este método podrá ser llamado tanto desde el Constructor de su Clase como desde cualquier otra parte del programa

P00 en JAVA

Subir y bajar la intensidad del color del televisor : Ahora queremos que el Televisor estándar además de ofrecer la operativa de cambiar los canales, también nos permita aumentar y disminuir la intensidad del color. Para ello vamos a crear cuatro nuevos métodos

```
public void subirColor(){
    System.out.println("Televisor - subirColor(): estoy subiendo el color");
    subirColorAyuda();
}
private void subirColorAyuda() {
    System.out.println("Televisor - subirColorAyuda(): sigo subiendo el color");
}
public void bajarColor(){
    System.out.println("Televisor - bajarColor(): estoy bajando el color");
    bajarColorAyuda();
}
private void bajarColorAyuda() {
    System.out.println("Televisor - bajarColorAyuda(): sigo bajando el color");
}
```

P00 en JAVA

Compilamos la Clase Televisor y seguidamente modificamos la clase Aplicacion

```
package paqtvestandar;
    public class Aplicacion {
        public static void main(String[] args) {
            ...
            tv.subirColor();
            //tv.subirColorAyuda();
        }
    }
```

Si descomentamos la invocación al método subirColorAyuda() apuntado por la variable de referencia tv, vamos a poder ver que el compilador se queja (y con razón!) porque estamos intentando acceder a un método de ámbito private

P00 en JAVA

- Implementa las Clases Televisor de tal forma que cuando creemos una instancia de Televisor, ésta ya tenga el volumen por defecto en posición 5.
- Por lo que respecta a la implementación de los métodos subirVolumen() y bajarVolumen(), no tendremos en cuenta los valores negativos ni tampoco el valor máximo del volumen, así que tendremos que implementar estos métodos con el siguiente código
 - `volumen = volumen + 1`
 - `volumen = volumen - 1`
- Desde la Clase Aplicacion tenemos que:
 - crear una instancia de Televisor
 - seguir cambiando los canales y subiendo el color como en el ejemplo anterior
 - subir el volumen una posición
 - mostrar un mensaje diciendo que "La posición del volumen es: "....

Ejercicios

Ejercicio 1: Modificar el ejemplo anterior para añadirle el atributo color.

- Cuando se cree el objeto por defecto se inicializara a 7.
- El método subirColor será el que incremente la variable color.
- El método bajarColor será el que decremente la variable color.
- Los métodos subirColorAyuda y bajarColorAyuda deben mostrar el valor del color.

Ejercicio 2: Modifica el ejercicio anterior para que los canales sean del 0 al 10, y en el caso de que estés en el canal 10 y subas el canal nos de el 0 y en el caso de que estés en el canal 0 y bajes el canal nos de el 10.

Ejercicio 3: Modifica el ejercicio anterior para que la intensidad de color sea de 1 a 7, y en el caso de que la intensidad sea 7 y subas la intensidad se quede en 7 y en el caso de que la intensidad sea 1 y la bajes se quede en 1.

Ejercicio 4: Modifica el ejercicio anterior para que el volumen sea de 0 a 15, y en el caso de que el volumen sea 15 y lo subas se quede en 15 y en el caso de que el volumen sea 0 y lo bajes se quede en 0.

Métodos con clases

Métodos con clases

- Los métodos definen las operaciones que se pueden realizar sobre la clase
- Declaración de métodos:

[acceso][static][final] tipo_retorno nombreMetodo ([args])

{/*Cuerpo método*/}

- Donde:
 - **acceso**: public, private, protected, package
 - **static**: se puede acceder a los métodos sin necesidad de instanciar un objeto de la clase
 - **final**: constante, evita que un método sea sobrescrito
 - **tipo_retorno**: tipo primitivo, referencia o void
 - **nombreMetodo**: identificador del método
 - **args**: lista de parámetros separados por comas

Métodos con clases

- Devolución de valores:
 - En la declaración se debe especificar el **tipo de dato** devuelto por el método.
 - Si el método no devuelve nada, el tipo de retorno será **void**.
 - Si el método devuelve algo:
 - Se pueden devolver tipos primitivos u objetos, ya sean objeto predefinidos u objetos de clases creadas por el propio usuario.
 - El cuerpo del método deberá contener una instrucción similar a `→ return valorDevuelto;`

Métodos con clases

- Los modificadores de acceso se utilizan para controlar el acceso a clases, atributos y métodos.
- Tipos de modificadores de acceso:
 - public
 - private
 - protected
 - package
- Clases
 - public
 - package
- Variables y métodos
 - public, private, protected y package

Métodos con clases

```
Class Punto {
```

```
    public int x;
```

```
    public int y;
```



ATRIBUTOS

```
    public double CalcularDistanciaCentro(){
```

```
        double z;
```

```
        z = Math.sqrt((x*x)+(y*y));
```

```
        return z;
```

```
    }
```

```
}
```



MÉTODO

Métodos con clases

- Los métodos se aplican siempre a un objeto de la clase usando el operador punto (.) salvo los métodos declarados como static. Dicho objeto es un argumento implícito:


```
nombreObjeto.metodo ( [args] ) ;
```

- Argumentos explícitos: van entre paréntesis, a continuación del nombre del método. Los tipos primitivos se pasan por valor, para cambiarlos hay que encapsularlos dentro de un objeto y pasar el objeto.
- Ejemplo:

```
double d = p1.CalcularDistanciaCentro();  
System.out.println("distancia:  " + d);
```


Métodos con clases

- Los métodos pueden definir variables locales:
 - Visibilidad limitada al propio método
 - Variables locales no se inicializan por defecto

```
public double calcularDistanciCentro() {  
    double z;   
    z = Math.sqrt((x*x)+(y*y));  
    return z;  
}
```

Métodos con clases

- **Método constructor**
- Método que se llama automáticamente cada vez que se llama un objeto de una clase.
- Características:
 - Reservar memoria e inicializar las variables de la clase
 - No tienen valor de retorno (ni siquiera void)
 - Tiene el mismo nombre que la clase
- Sobrecarga:
 - Una clase puede tener varios constructores, que se diferencia por el tipo y su número de sus argumentos

Métodos con clases

- Constructores por defecto:
 - Constructor sin argumentos que inicializa:
 - Tipos primitivos a su valor por defecto
 - Strings y referencias a objetos a null
- Creación de objetos:

```
Punto p = new Punto(1, 1);
```

- Declaración: Punto p;
- Instanciación: new Punto(1, 1); crea el objeto
- Inicialización: new Punto(1, 1); llama al constructor que es el que llama al objeto.

Métodos con clases

```
public class Punto {  
    public int x;  
    public int y;  
    public Punto(int a) { x=a; y=a; }  
    public Punto(int a, int b) { x=a; y=b; }  
    public double CalcularDistanciaCentro(){  
        double z;  
        z = Math.sqrt((x*x)+(y*y));  
        return z; }  
}
```

this en JAVA

- La variable this
- Definida implícitamente en el cuerpo de los métodos
- Dentro de un método o de un constructor, this hace **referencia al objeto actual**


```
public class Punto {  
    public int x = 0;  
    public int y = 0;  
    public Punto(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

```
public class Punto {  
    public int x = 0;  
    public int y = 0;  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

Métodos con clases

- Un constructor puede llamar a otro constructor de su propia clase si:
 - El constructor al que se llama está definido
 - Se llama utilizando la palabra `this` en la primera sentencia

```
public Cliente(String n, long dni){  
    nombre = n;  
    DNI = dni;}  
public Cliente(String n, long dni, long tel){  
    nombre = n;  
    DNI = dni;  
    telefono = tel;}  
public Cliente(String n, long dni, long tel){  
    this(n,dni);  
    telefono = tel;}
```

A red line originates from the first parameter 'n' in the call 'this(n,dni);' within the third constructor, extends vertically upwards, and then horizontally to the left, pointing towards the first constructor's definition.

Métodos con clases

- Puede haber dos o más métodos que se llamen igual en la misma clase
- Se tienen que diferenciar en los parámetros (tipo o número)
- El tipo de retorno es insuficiente para diferenciar dos métodos

Métodos con clases

- Modificadores de acceso
 - Los modificadores de acceso permiten al diseñador de una clase determinar quién accede a los datos y métodos miembros de una clase.
 - Los modificadores de acceso preceden a la declaración de un elemento de la clase (ya sea dato o método), de la siguiente forma:

```
[modificadores] tipo_variable nombre;
```

```
[modificadores] tipo_devuelto nombre_Metodo (lista_Argumentos);
```


Métodos con clases

- Existen los siguientes modificadores de acceso:
- **public** - Todo el mundo puede acceder al elemento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.
- **private** - Sólo se puede acceder al elemento desde métodos de la clase, o sólo puede invocarse el método desde otro método de la clase.
- **protected** - Se explicará en el capítulo dedicado a la herencia.
- **sin modificador** - Se puede acceder al elemento desde cualquier clase del package donde se define la clase

Pueden utilizarse estos modificadores para cualquier tipo de miembros de la clase, incluidos los constructores (con lo que se puede limitar quien puede crear instancias de la clase).

Métodos con clases

Modificadores de acceso para clases

Las clases en sí mismas pueden declararse:

- **public** - Todo el mundo puede usar la clase. Se pueden crear instancias de esa clase, siempre y cuando alguno de sus constructores sea accesible
- sin modificador - La clase puede ser usada e instanciada por clases dentro del package donde se define

Las clases no pueden declararse ni **protected**, ni **private**

Métodos con clases

- Los elementos (atributos y métodos) definidos como **static** son independientes de los objetos de la clase
- Atributos estáticos - variables de clase:
 - Un atributo static es una variable global de la clase
 - Un objeto de la clase no copia los atributos static → todas las instancias comparten la misma variable
 - Uso de atributos estáticos:
 - nombreClase.nombreAtributoEstático

Métodos con clases

```
public class MiClase {  
    String nombre;  
    static int contador;  
    public MiClase(String n) { nombre = n; contador++; }  
    public void imprimeContador()  
    { System.out.println("Contador: " + contador); }  
}  
  
...  
MiClase o1 = new MiClase("Primero");  
MiClase o2 = new MiClase("Segundo");  
o2.imprimeContador(); //2  
MiClase.contador = 1000;  
o2.imprimeContador(); //1000
```

Métodos con clases

- Métodos estáticos:

- Un método static es un método global de la clase
- Un objeto no hace copia de los métodos static
- Suelen utilizarse para acceder a atributos estáticos
- No pueden hacer uso de la referencia this
- Llamada a métodos estáticos:
- Ejemplo:

```
nombreClase.nombreMetodoEstático()
```

```
public static void actualizaContador( ) {  contador = 0;  }  
//Principal:  
...  
MiClase.actualizaContador( );  
o.imprimeContador( );
```

Métodos con clases

```
public class Persona{
    //Atributo estático de la clase
    private static int nPersonas;
    //Cada instancia incrementa este atributo
    public Persona ( ) {
        nPersonas++;
    }
    //Método estático que retorna un atributo estático
    public static int getNPersonas( ) {
        return nPersonas;
    }
    public static void main (String[] args) {
        //Se crean instancias
        Persona p1 = new Persona( );
        Persona p2 = new Persona( );
        Persona p3 = new Persona( );
        //Accedemos al método estático para ver el número
        //de instancias de tipo Personas creadas
        System.out.println(Persona.getNPersonas( ));
    }
}
```

Métodos con clases

- **Un Atributo static:**

- No es específico de cada objeto. Solo hay una copia del mismo y su valor es compartido por todos los objetos de la clase.
- Podemos considerarlo como una variable global a la que tienen acceso todos los objetos de la clase.
- Existe y puede utilizarse aunque no existan objetos de la clase.
- Para acceder a un atributo de la clase se escribe:
- `NombreClase.atributo`

- **Un método static:**

- Tiene acceso solo a los atributos estáticos de la clase.
- No es necesario instanciar un objeto para poder utilizarlo.
- Para acceder a un método de la clase se escribe:
- `NombrClase.metodo()`

Métodos con clases

- La palabra reservada **final** indica que su valor no puede cambiar
- Si se define como final:
 - **Una clase** → No puede tener clases hijas (seguridad y eficiencia del compilador)
 - **Un método** → No puede ser redefinido por una subclase
 - **Una variable** → Tiene que ser inicializada al declararse y su valor no puede cambiarse
- Suele combinarse con el modificador **static**
- El identificador de una variable final debe escribirse en mayúsculas

Métodos con clases

```
public class Constantes{  
    //Constantes públicas  
    public static final float PI = 3.141592f;  
    public static final float E = 2.728281f;  
    //main  
    public static void main (String[] args) {  
        System.out.println("PI = " + Constantes.PI);  
        System.out.println("E = " + Constantes.E);  
    }  
}
```

Arrays de objetos

Arrays de objetos

La clase precio

```
public class Precio {  
    public double euros;  
  
    public double getPrecio() {  
        return euros;  
    }  
    public void setPrecio(double x) {  
        euros=x;  
    }  
}
```

Arrays de objetos

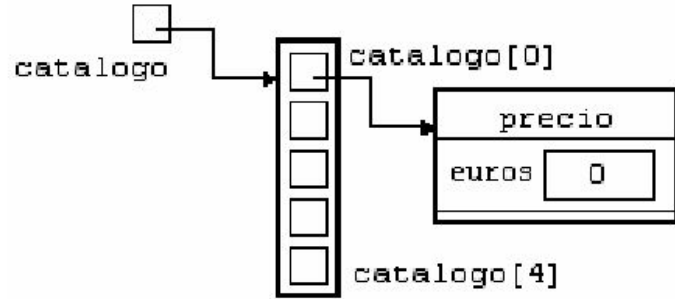
La clase PruebaPrecio

```
public class PruebaPrecio {  
    public static void main (String [] args ) {  
        Precio p = new Precio();  
        p.setPrecio(56.8);  
        System.out.println("Valor = " + p.getPrecio());  
        Precio q = new Precio(); // Crea una referencia y el objeto  
        q.euros=75.6;             // Asigna 75.6 al atributo euros  
        System.out.println("Valor = " + q.euros);  
    }  
}
```

Arrays de objetos

El uso de vectores no tiene por qué restringirse a elementos de tipo primitivo. Por ejemplo, pueden crearse referencias a arrays de la clase Precio o de la clase String:

```
Precio [ ] catalogo;  
catalogo = new Precio [5];  
catalogo[0] = new Precio();
```



Primero se crea la referencia al array de punteros, luego se crea el array de punteros y, finalmente, se crea la instancia de la clase precio y se almacena su dirección de memoria en el primer elemento del array de punteros.

Arrays de objetos

```
public class ArrayPrecios {  
    public static void main (String [] args) {  
        Precio [ ] catalogo;  
        catalogo = new Precio [5];  
        for (int i=0; i<catalogo.length; i++) {  
            catalogo[i] = new Precio();  
            catalogo[i].setPrecio(10*Math.random());  
            System.out.println("Producto "+ i + " : " + catalogo[i].getPrecio());  
        }  
        //Busqueda del máximo precio  
        double maximo=catalogo[0].getPrecio();  
        for (int i=1; i<catalogo.length; i++) {  
            if (maximo<catalogo[i].getPrecio())  
                maximo=catalogo[i].getPrecio();  
        }  
        System.out.println("El mas caro vale "+ maximo + " euros");  
    } }  
}
```

ArrayList de objetos

ArrayList de objetos

La clase **ArrayList** permite almacenar elementos en memoria de manera dinámica.

La principal diferencia con los arrays es que el número de elementos que almacena no está limitado por un número prefijado.

ArrayList de objetos

La declaración de un ArrayList se hace según el siguiente formato:

ArrayList<nombreClase> nombreDeLista;

- Entre <> indicamos la clase o tipos básicos de los objetos que se almacenarán.
- Ejemplo:
 - ArrayList<String> listaPaises;
 - ArrayList<Persona> listaPersonas;

Creación del ArrayList

La creación de un ArrayList se hace según el siguiente formato:

```
nombreDeLista = new ArrayList();
```

También se puede declarar a la vez que se crea:

ArrayList<nombreClase> nombreDeLista = new ArrayList();

La clase ArrayList forma parte del paquete java.util por lo que hay que incluir en la parte inicial del código el paquete

```
import java.util.ArrayList
```

Insertar elementos al final

El método **add** de la clase ArrayList posibilita añadir elementos.

Se colocan después del último elemento que hubiera en el ArrayList

```
boolean add(Object elementoAInsertar);
```

```
ArrayList<String> listaPaises = new ArrayList();  
listaPaises.add("España");    //Ocupa la posición 0  
listaPaises.add("Francia");   //Ocupa la posición 1  
listaPaises.add("Portugal");  //Ocupa la posición 2
```

Insertar elementos en una determinada posición

Es posible insertar un elemento en una determinada posición desplazando el elemento que se encontraba en esa posición, y todos los siguientes, una posición más. Se utiliza el método `add` indicando como primer parámetro el número de la posición.

```
void add(int posicion, Object elemento);
```

```
ArrayList<String> listaPaises = new ArrayList();  
listaPaises.add("España");  
listaPaises.add("Francia");  
listaPaises.add("Portugal");  
//El orden hasta ahora es: España, Francia, Portugal  
listaPaises.add(1, "Italia");  
//El orden ahora es: España, Italia, Francia, Portugal
```

Eliminar elementos

Para eliminar un determinado elemento se emplea el método **remove** al que se le puede indicar por parámetro un valor int con la posición a suprimir o bien , se puede especificar el elemento a eliminar si es encontrado en la lista.

```
Object remove(int posicion)
boolean remove(Object elementoASuprimir)
```

```
ArrayList<String> listaPaises = new ArrayList();
listaPaises.add("España");
listaPaises.add("Francia");
listaPaises.add("Portugal");
//El orden hasta ahora es: España, Francia, Portugal
listaPaises.add(1, "Italia");
//El orden ahora es: España, Italia, Francia, Portugal
listaPaises.remove(2);
//Eliminada Francia, queda: España, Italia, Portugal
listaPaises.remove("Portugal");
//Eliminada Portugal, queda: España, Italia
```

Consultar un elemento

El método **get** permite obtener el elemento almacenado en una determinada posición:

```
Object get(int posicion)
```

```
System.out.println(listaPaises.get(3));
```

```
//Siguiendo el ejemplo anterior, mostraría: Portugal
```

Modificar y buscar un elemento

El método **set** permite modificar el elemento almacenado en una determinada posición.

```
Object set(int posicion, Object nuevoElemento)
```

```
listaPaises.set(1, "Alemania");
```

```
//Se modifica el país que había en la posición 1 por Alemania
```

El método **indexOf** retorna un valor int con la posición que ocupa el elemento que se indica por parámetro

```
int indexOf(Object elemento)
```

```
String paisBuscado = "Francia";
```

```
int pos = listaPaises.indexOf(paisBuscado);
```

```
if(pos!=-1)
```

```
    System.out.println(paisBuscado + " encontrado en la posición: " + pos);
```

```
else System.out.println(paisBuscado + " no se ha encontrado");
```

Recorrer ArrayList

Podemos recorrer un ArrayList con un bucle como lo haríamos con los arrays convencionales. Para obtener el número de elementos se puede utilizar el método **size()**.

Ejemplo:

```
for(int i = 0; i < listaPaises.size(); i++) {  
    System.out.println(listaPaises.get(i));  
}
```

El `foreach` nos permite recorrer los elementos de un ArrayList sin utilizar un índice. Aunque la flexibilidad es menor que el bucle **for** normal, es muy útil para recorrer de forma normal un ArrayList.

Ejemplo:

```
for (String s : listaPaises) {  
    System.out.println(s);  
}
```


Otros métodos de interés

- **void clear():** Borra todo el contenido de la lista
- **Object clone():** Retorna una copia de la lista
- **boolean contains(Object elemento):** Retorna true si se encuentra el elemento en la lista, y false en caso contrario.
- **boolean isEmpty():** Retorna true si la lista está vacía
- **Object[] toArray():** Convierte la lista a un array

Bibliografía:

Allen Weiss, M. (2007). *Estructuras de datos en JAVA*. Madrid: Pearson

Froufe Quintas, A. (2002). *JAVA 2: Manual de usuario y tutorial*. Madrid: RA-MA

J. Barnes, D. *Programación orientada a objetos en JAVA*. Madrid: Pearson

Desing Patterns. Elements of Reusable. OO Software

JAVA Limpio. Pello Altadi. Eugenia Pérez

Apuntes de Programación de Anna Sanchis Perales

Apuntes de Programación de Lionel Tarazón Alcocer



Ilustraciones:

<https://pixabay.com/>

<https://freepik.es/>

<https://lottiefiles.com/>

Preguntas

