IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones web

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

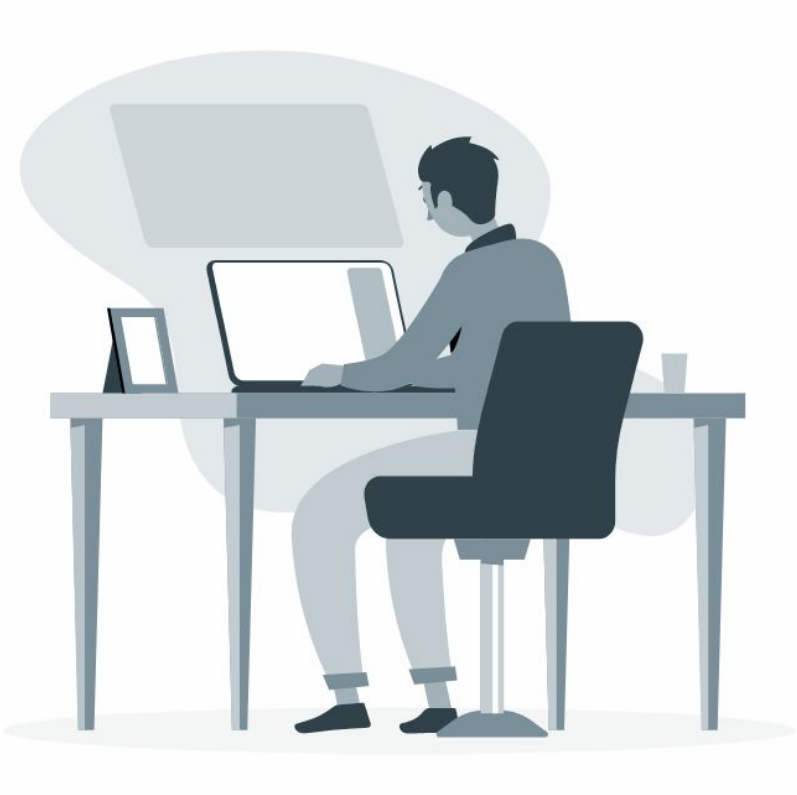
GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Edu Torregrosa Llácer

(aulaenlanube.com)

Introducción a la POO en JAVA

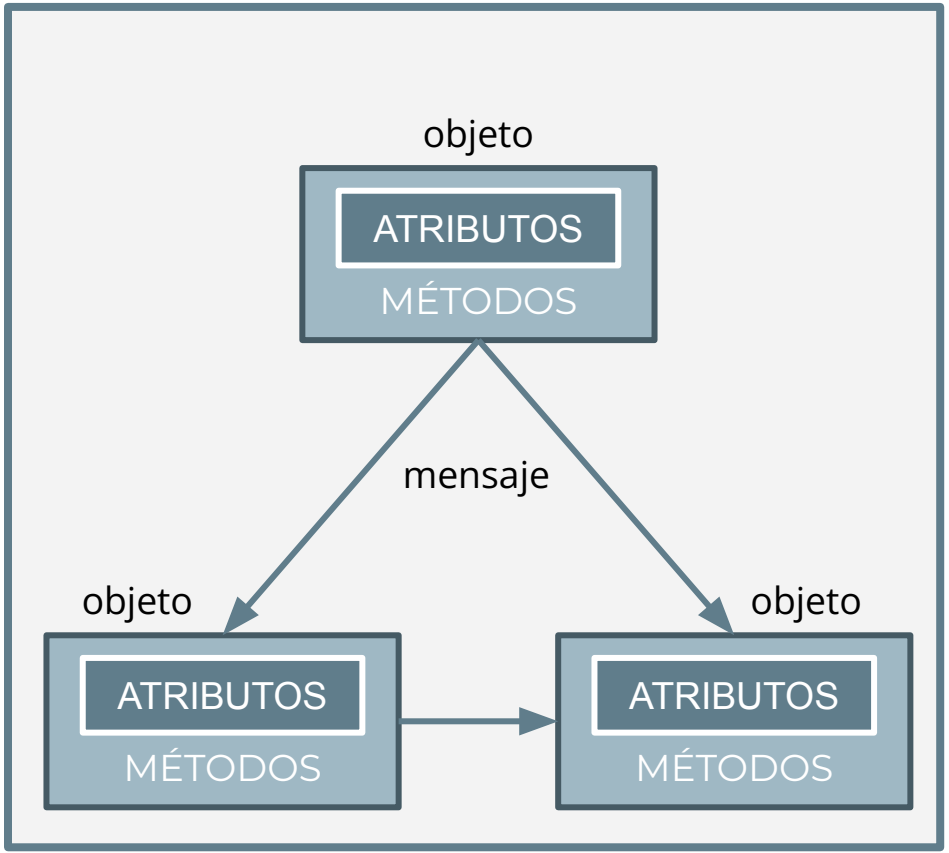
1. Elementos de la POO
2. Características de la POO
3. Diseño OO - UML
4. Ejemplos de diseño OO
5. POO en Java
    - 5.1. Clases, instancias y objetos
    - 5.2. Constructores, métodos y this
    - 5.3. Acceso: public, private y protected
    - 5.4. Elementos estáticos (static)
    - 5.5. Clases inmutables y referencias de objetos
    - 5.6. 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, (propiedades), y unas formas de operar sobre ellos, los **métodos** o **procedimientos**(definen el comportamiento del objeto).
- Durante la ejecución, los objetos reciben y envían **mensajes** a otros objetos para realizar las acciones requeridas.



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, su dni es 20202020A trabaja de profesor, y tiene 45 años de edad; mientras que Marta es una mujer de 25 años, su dni es 30303030B, vive en Madrid y es periodista(este es su estado).
  - De ambas personas podemos extraer información, y modificar sus estado (éste es su comportamiento).
- La **POO** es una manera de diseñar y desarrollar software que trata de **imitar** la **realidad** tomando algunos conceptos esenciales de ella.

Introducción

- Si nos pidieran que hiciéramos un programa orientado a objetos que simulara lo anterior haríamos:
  - La **clase** Persona que tendría las **variables** nombre, dni, 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 POO

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

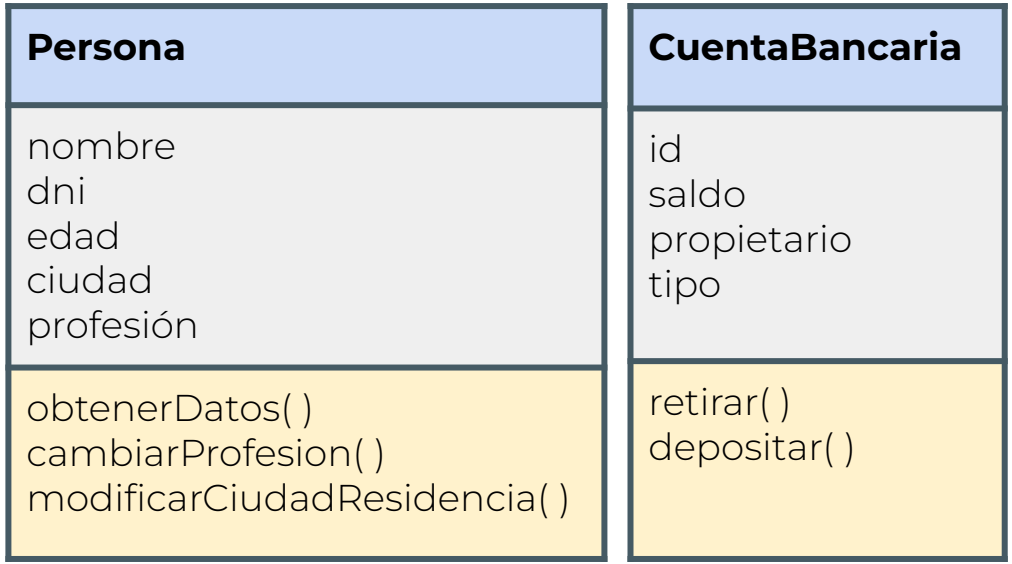
Elementos básicos de la POO

- **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 POO

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 "Persona" y la clase “Cuenta bancaria” se puede representar gráficamente como sigue:



Elementos básicos de la POO

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 POO

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 "retirar(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.

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

Elementos básicos de la POO

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

Características de la POO

Características de la POO

- 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.
  - Controlar cada uno de los objetos de forma individual.
  - Desarrollar un código más breve y conciso.
  - Reutilizar el código escrito.

Características de la POO

Abstracción:

- Viene de abstracto, algo abstracto es algo a la que hacemos referencia de forma conceptual, sin definirlo de una forma específica. Por ejemplo: para abstraer el objeto Persona, nos debemos hacer preguntas de tipo... ¿Qué datos queremos almacenar de la persona? ¿Qué comportamiento de la Persona queremos simular?
- El 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 POO

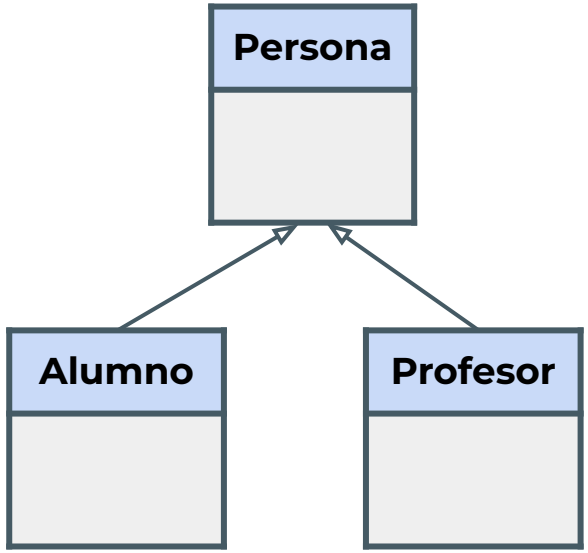
Herencia:

Es la característica por la cual los objetos para su creación se basan en otra clase de base, heredando todos sus atributos y sus métodos.

Podemos crear una **clase Alumno** que hereda todos los atributos y métodos de la clase Persona, pero dicha clase(Alumno), 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 un mismo **método** puede **comportarse diferentemente** sobre distintas clases.
- Por ejemplo, el método "obtenerDatos()" puede comportarse diferentemente sobre la clase Alumno y la clase Profesor.

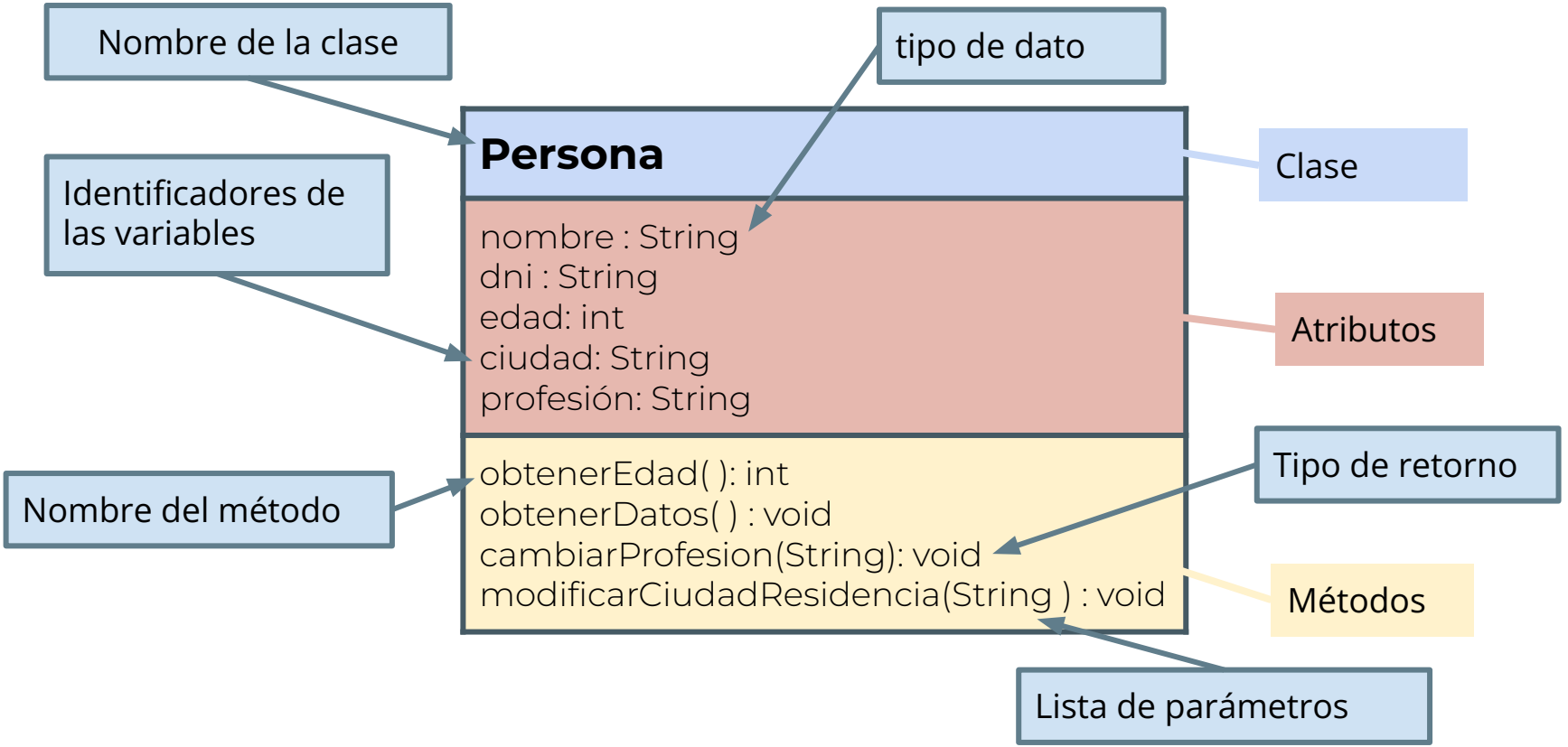
Diseño orientado a objetos - UML

Introducción a UML

- El **UML** (Unified Modelling Language o Lenguaje de Modelado Unificado) se utiliza para establecer cómo se estructura la resolución de un problema mediante la orientación a objetos. Además, se utiliza para saber de qué manera interactúan los diferentes componentes para lograr una tarea concreta.
- El UML es un lenguaje estándar que permite especificar con notación gráfica software orientado a objetos.
- 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.

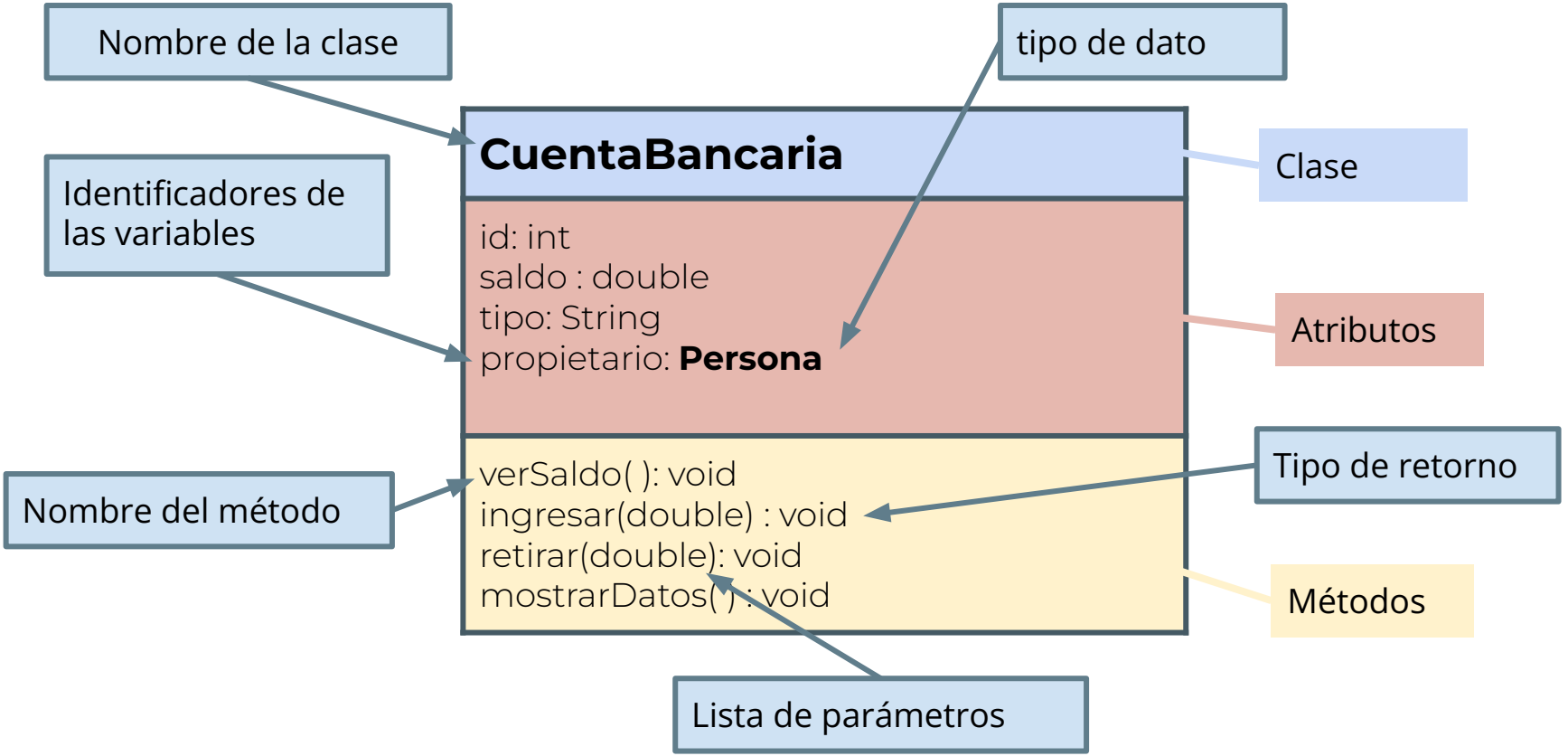
Ejemplo de clase en UML

A continuación, se muestra un ejemplo de la clase **Persona** en UML:



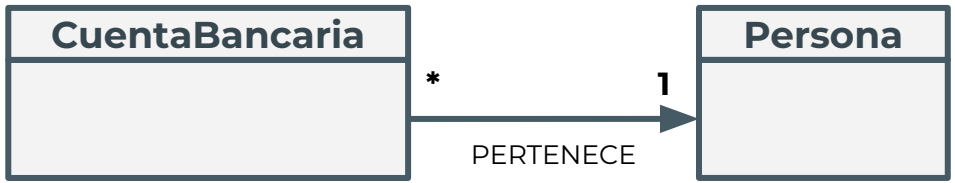
Ejemplo de clase en UML

A continuación, se muestra un ejemplo de la clase **CuentaBancaria** en UML:



Relaciones entre clases

- Una vez identificadas las clases de los objetos que componen el problema a resolver, el siguiente paso es establecer qué relaciones hay entre dichas clases. 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.
  - → Si no se especifica navegabilidad(sin flecha), se tratará de una **asociación bidireccional**, ambas clases podrán llamar a métodos de la otra.

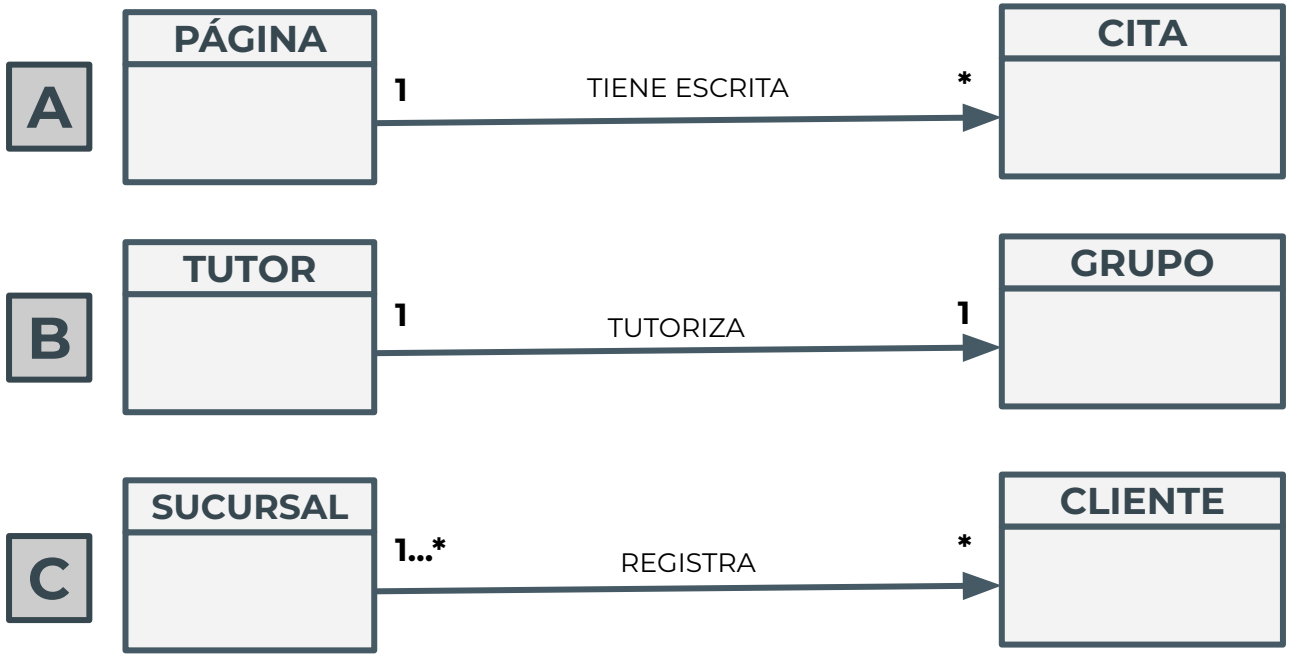
Relaciones entre clases

- **En cada extremo se especifica la cardinalidad**.
- **La cardinalidad debe especificar 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 **Tutor** siempre tiene un objeto de la clase **Grupo** enlazado. Por tanto, un tutor siempre tutoriza a un grupo.
- No se puede dar el caso de que un tutor no tutorice a ningún grupo. La inversa también es cierta, todo grupo es tutorizado por algún tutor.
- El tutor puede llamar operaciones sobre el grupo, pero no al revés. Esto tiene sentido, ya que es el tutor el que controla al grupo.

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.

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.

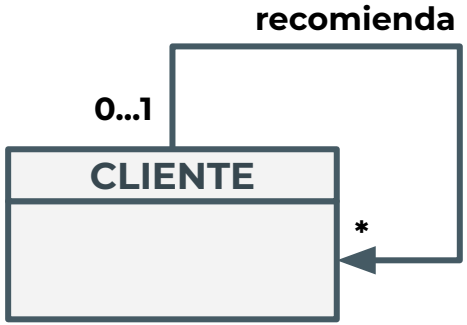
- Una página contiene citas escritas en su interior y se puede considerar que lo escrito en una página es parte de la misma.
- Expresa **contiene o puede contener.** En el ejemplo anterior una página puede contener citas, pero puede existir una página que en principio no tenga citas.

Composición

- Expresa el concepto de **es parte de**, ya que 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 es parte de, será 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.

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.



Implementar asociaciones entre clases

Implementación de asociaciones en JAVA

- Las asociaciones entre clases se captan mediante atributos 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 la cardinalidad es 2 habrán 2 atributos de la clase destino en clase origen.
        - Si cardinalidad 1, habrá un único atributo de la clase destino en la clase origen.
        - Las cardinalidades indeterminadas (caso *), se suele usar cualquier colección que se pueda modificar de forma dinámica, por ejemplo un ArrayList.
- Decidir qué métodos incluir en cada clase puede ser difícil durante el proceso de diseño
- Un principio clave es lograr cohesión, es decir, tener clases coherentes y con un propósito claro en el problema que se está resolviendo. Es importante que las clases trabajen juntas en armonía y que cada una tenga un objetivo específico.
- En una clase, solo se deben incluir los atributos y operaciones esenciales para cumplir su determinada función.

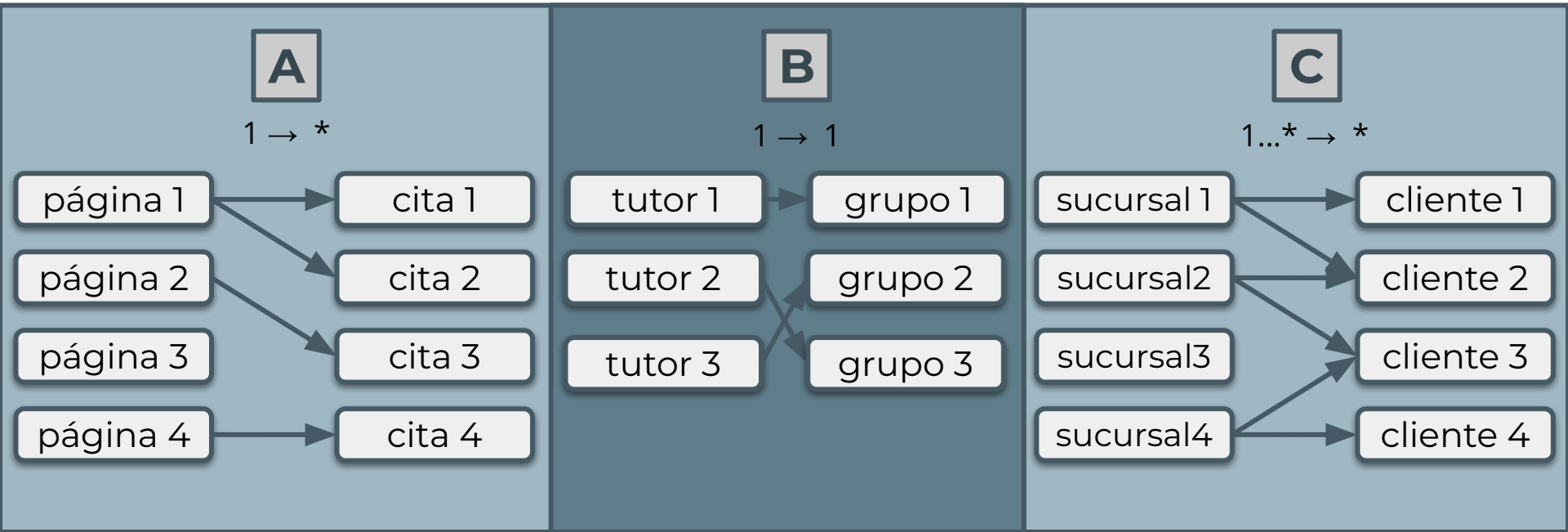
Mapa de objetos

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 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.

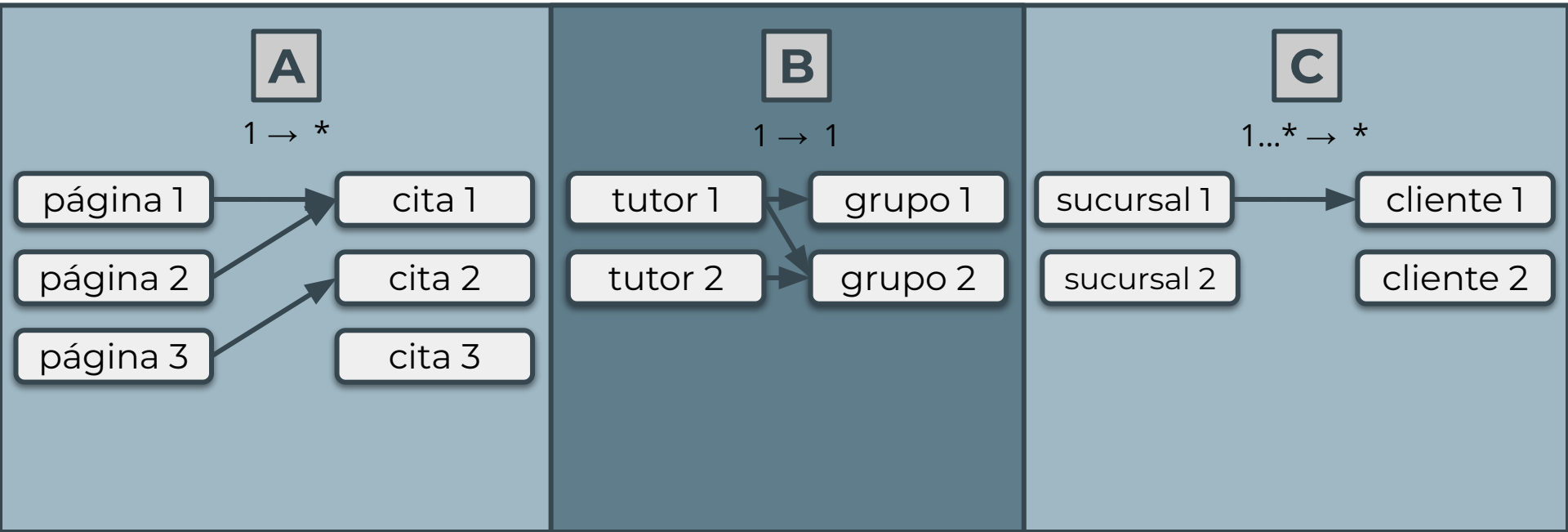
Mapa de objetos

- 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 consideran incorrectos según las cardinalidades especificadas en las asociaciones.



Programación orientada a objetos en JAVA

Programación orientada a objetos en JAVA

- JAVA es un lenguaje **orientado a objetos** y programar en JAVA consiste en construir clases y utilizar esas clases para crear objetos de forma que representen correctamente el problema que queremos resolver.
- En función de la estructura de la clase y del uso tenemos dos tipos básicos de clases:
    - **Clase – Tipo de datos**: definen el conjunto de atributos y de posibles valores que tomarán los objetos. Además se incluirán las operaciones(métodos) que se podrán realizar con dichos atributos.
    - **Clase – Programa**: son los que inician la ejecución del código, la clase que contiene el main

Estructura de una clase en JAVA

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

```
[ámbito] class NombreDeLaClase {
        // Definición de atributos
        [ámbito] tipo nombreVar1;
        [ámbito] tipo nombreVar2;
        ……
        // Definición de métodos
        // Constructores
        …
        // Otros métodos
}
```

Ejemplo de clase en JAVA

```
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 creación de instancia en JAVA

```
public class PrimerPrograma {

    public static void main(String[] args) {

        // Crear un círculo
        Circulo c1 = new Circulo();
        c1.setRadio(2.9);
        System.out.println("Los datos del circulo: " + c1.toString());
    }
}
```

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.
  - Es el caso de los métodos **getRadio( )** o **area( )** de la clase Circulo.

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

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

Modificadores de acceso: atributos y métodos

Modificadores de acceso para atributos y métodos:

- **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).

Modificadores de acceso en 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**.

Atributos de una clase

Los atributos o variables de instancia representan información de cada objeto de la clase y se declaran de un tipo de dato concreto. Además se puede añadir el ámbito de cada atributo, 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(instancias) 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;
  }
}
```

Getters y Setters

Los **métodos** utilizados para consultar o modificar atributos de la propia instancia se denominan Getters y Setters:

- **Getters**: permiten obtener el valor de un atributo.
- **Setters**: permiten modificar el valor de un atributo.

No es obligatorio que todos los atributos de una clase dispongan de setters y getters, dependerá de la propia aplicación. Una buena práctica es únicamente poner los mínimos getters y setters, con ello proporcionaremos una buena encapsulación de los datos.

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

public void setRadio(double nuevoRadio) {
   radio = nuevoRadio;
}
```

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

public void setRadio(double nuevoRadio) {
     if(nuevoRadio >= 0)
          radio = nuevoRadio;
     else radio = 0;
}
```

Métodos de una clase

- Los clasificamos según su función:
  - **Constructores**: permiten crear el objeto.
  - **Modificadores(setters)**: permiten modificar el estado (valores de los atributos).
  - **Consultores(getters)**: permiten conocer, sin cambiar, el estado del objeto.
  - **De clase**: permiten realizar cualquier operación adicional sobre la instancia.

```
public class Circulo {
          ...
          //Constructor vacío
          public Circulo() {      radio = 50; color = "negro“; }
          //Constructor con 4 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; }
          //Modificador: setter
          public void setRadio(double nuevoRadio) { radio = nuevoRadio; }
          //De clase
          public double area() { return 3.14 * radio * radio;
      }
}
```

Ejemplo POO en JAVA
Clase Televisor

POO en JAVA

- Vamos a crear una clase Televisor y añadiremos un atributo llamado canal que va a ser de tipo int.

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

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

```
public class Televisor {
    private int canal;
    // constructor por defecto sin parámetros
    public Televisor() {  }
    public void subirCanal() {  canal++;      }
    public void bajarCanal() {  canal--;      }
    public int getCanal()    {  return canal; }
}
```

POO en JAVA

- Añadimos a la clase Aplicación el siguiente código:

```
Televisor tv;
```

  - Aquí estamos declarando una variable de referencia **tv**. De momento su valor es **null** ya que todavía no apunta a ninguna Instancia.

```
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.

POO en JAVA

- Para invocar los métodos de un Objeto(instancia), tenemos que tener 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 otra clase llamada **AplicaciónTv**.

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

POO en JAVA

- Los constructores se declaran de la siguiente forma:

```
public 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

POO en JAVA

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

```
public class Televisor {

    int canal; //no indicamos modificador de acceso

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

POO en JAVA

Ahora vamos a crear dos instancias de Televisor teniendo en cuenta que el canal no tiene modificador de acceso. Uno de ellos estará referenciado por una variable de referencia llamada **tv1** y el otro por otra variable de referencia llamada **tv2**, en clase **AplicacionTv**.

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

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**.

POO en JAVA

- De nuevo vamos a indicar que el **canal** es **private**.
    - Al indicar que el ámbito es **private** estamos diciendo que sólo desde dentro de la Clase Televisor se puede acceder al atributo canal, para modificar el atributo desde el exterior lo debemos hacer a través de métodos(setters).
    - Cuando el usuario baja el canal, se podría dar el caso que llegara al canal 1 y si sigue bajando el canal llegaría al canal 0, -1, etc. Para evitar esto vamos a invocar al método setCanal(...) pasándole como argumento el resultado de la operación de bajar el canal.
        - Si por ejemplo el canal más alto no puede superar el número 99, simplemente lo podemos solucionar modificando el método setCanal(...)
- **public void setCanal(int valorCanal) { ... }**
    - Al ser **public**, este método podrá ser llamado tanto desde cualquier lugar de su clase, ya sea en el constructor, como desde cualquier otro método. Incluso desde una clase externa.

POO en JAVA

Clase Televisor con posibles canales entre 1 y 99:

```
public class Televisor {
    private int canal;
    public Televisor() { canal = 1; }
    public Televisor(int valorCanal) {
        setCanal(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 < 1 || valorCanal >= 100) canal = 1;
         else   canal = valorCanal;
    }
}
```

Ejercicios iniciación OO en JAVA
Clase Televisor

POO en JAVA

- Amplía la Clase Televisor con un atributo adicional para el volumen. Cuando creemos una instancia de Televisor, ésta ya tendrá el volumen por defecto en el nivel 5, y el canal por defecto en el 1.
- Por lo que respecta a la implementación de los métodos que modifican el canal y el volumen, no tendremos en cuenta los valores negativos ni tampoco el valor máximo(100). Tendremos que implementar estos métodos para que no hagan nada en esos casos concretos
- Al modificar el volumen y el canal se debe mostrar mensajes indicando tanto el nivel de volumen, como el canal. Por ejemplo: "**Volumen: valorVolumen**", "**Canal: valorCanal**".
- Para comprobar el funcionamiento, desde la Clase AplicacionTv tenemos que:
    - Crear instancias de Televisor.
    - Cambia los canales y el volumen para comprobar que todo funciona de forma correcta, y cada Televisor tiene su propio estado.

Ejercicios

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

- Cuando se crea el objeto por defecto se inicializa a 7.
- El método subirColor( ) será el que incremente la variable color.
- El método bajarColor( ) será el que decrementa la variable color.

Ejercicio 2: Modificar 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 dé el 0 y en el caso de que estés en el canal 0 y bajes el canal nos dé 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 en clases JAVA:
Ejemplo clase Punto

Métodos en clases

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

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

  {/*Cuerpo método*/}

- Donde:
  - **acceso**: public, private, protected, sin modificador.
  - **static**: se puede acceder a los métodos sin necesidad de instanciar un objeto de la clase.
  - **final**: constante, evita que un método sea sobreescrito.
  - **tipoRetorno**: se pueden devolver tipos primitivos u objetos, ya sean objetos predefinidos u objetos de clases creadas por el propio usuario. Si el método no devuelve nada, el tipo de retorno será void.
  - **nombreMetodo**: identificador del método.
  - **args**: lista de parámetros separados por comas.

Métodos en clases: constructor

- **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.
  - Tiene el mismo nombre que la clase.
- Sobrecarga de constructores:
  - Una clase puede tener varios constructores, que se diferencia por el tipo y su número de sus argumentos, pueden tener la misma cantidad de argumentos(parámetros), pero como mínimo uno de ellos tiene que ser de un tipo distinto.
- Constructores por defecto, constructor sin argumentos que inicializa un objeto:
  - Tipos primitivos a su valor por defecto.
  - Strings y referencias a objetos a null.

Métodos en 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() {
        return Math.sqrt((x*x) + (y*y));
    }
}
```

ATRIBUTOS

CONSTRUCTORES

MÉTODO

Métodos en clases

- Los métodos se invocan siempre a partir de una instancia de una clase, se usará el operador punto (.), salvo los métodos declarados como **static**.

```
nombreInstancia.metodo(args);
```

- **Sobrecarga de métodos**: 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
- **Modificadores de acceso**:
  - Los modificadores de acceso permiten al diseñador de una clase determinar quién accede a los métodos de una clase. Los modificadores de acceso preceden a la declaración del método.

Métodos en clases

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

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

Variable local

Una buena práctica es no declarar variables que no sean necesarias.

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

La palabra reservada this en JAVA

This en JAVA

- La palabra reservada **this** hace referencia a la instancia actual de la clase.
- Está definida implícitamente en el cuerpo de todos los métodos.
- Dentro de un método o de un constructor, **this hace referencia al objeto actual(instancia actual)**.

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

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

This en JAVA

- 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;
}
```

Métodos y atributos estáticos (static)

Métodos estáticos

- Los elementos (atributos y métodos) definidos como **static** son independientes de los objetos(instancias) 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 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**.

Métodos estáticos

```
public class Punto {
     …
     //atributo estático de la clase
     private static int cantidadPuntos;
     //cada instancia incrementa este atributo
     public Punto ( ) {
          cantidadPuntos++;
     }
     //método estático que retorna un atributo estático
     public static int getPuntos( )
          return cantidadPuntos;
     }
     …
}
```

```
public class App {
     public static void main (String[] args) {
          //se crean instancias
          Punto p1 = new Punto ( );
          Punto p2 = new Punto ( );
          Punto p3 = new Punto ( );
          //accedemos al método estático para ver el
          //número de instancias de tipo Punto creadas
          System.out.println(Punto.getPuntos( ));
     }
}
```

Atributos y métodos static

- **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.nombreAtributo**
- **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.nombreMetodo(args)**

Atributos y métodos static

- 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(si es static) y su valor no puede cambiarse.
- Suele combinarse con el modificador **static**.
- El identificador de una variable final por convención debe escribirse en mayúsculas.

Atributos y métodos static

```
public class Constantes{
    //constantes públicas
    public static final float PI = 3.141592f;
    public static final float E = 2.728281f;

    public static final void mostrarConstantes() {
        System.out.println("PI = " + PI);
        System.out.println("E = " + E);
    }
}

public class App {
    public static void main (String[] args) {
        Constantes.mostrarConstantes();
        System.out.println("PI = " + Constantes.PI);
        System.out.println("E = " + Constantes.E);
    }
}
```

Paso por valor y paso por referencia: Objetos en JAVA

Paso por valor y paso por referencia : Objetos

- En JAVA, el paso de parámetros a los métodos se realiza por valor. Esto implica que cuando se pasa una variable a un método, se pasa una copia de su valor en lugar de la variable original. Por lo tanto, cualquier cambio realizado en el parámetro dentro del método **NO** afecta al valor original de la variable fuera del método.
- Sin embargo, cuando se pasa un objeto a un método, se está pasando la referencia del objeto, no una copia del objeto en sí. Por lo tanto, si pasamos un objeto como parámetro, cualquier cambio realizado en el objeto dentro del método, **SI** afecta al objeto original fuera del método. En resumen:
    - En JAVA los **tipos primitivos** siempre se pasan por **valor** a los métodos.
    - En JAVA los **objetos** siempre se pasan por **referencia** a los métodos.
- Por otra parte, existen casos especiales, como por ejemplo las Strings, ya que éstas son objetos, pero son inmutables. Esto implica que no se puede modificar una String sin crear una nueva instancia, lo veremos en detalle en un par de diapositivas.

Paso por valor y paso por referencia: Objetos

```
public static void main(String[] args) {

        int num = 1;
        String nombre = "String inmutable";
        Punto p1 = new Punto(1,1);

        modificarPunto(p1);
        modificarEntero(num);
        modificarString(nombre);

        System.out.println("Punto.x = " + p1.getX());    //  Punto.x = 999
        System.out.println("num = " + num);              //  num = 1
        System.out.println("nombre = " + nombre);        //  nombre = String inmutable
}

public static void modificarPunto(Punto p)    {  p.setX(999);                }
public static void modificarEntero(int n)     {  n = 999;                    }
public static void modificarString(String s)  {  s = "String modificada";   }
```

Clases inmutables y referencias de objetos

Clases inmutables en JAVA

- En JAVA, una clase inmutable es aquella cuyo estado interno no puede ser modificado después de su creación. Esto significa que una vez que se ha creado una instancia de un objeto de una clase inmutable, no se pueden modificar las variables de instancia de ese objeto.
- Una clase inmutable tiene numerosos beneficios, se puede utilizar en un ambiente multihilo sin necesidad de sincronización adicional, ya que cada hilo tiene su propia copia de la clase y no hay posibilidad de conflicto de acceso. Además, las clases inmutables son fáciles de entender y testear. Son más seguras y menos propensas a errores, ya que su comportamiento y estado no cambia después de su creación.
- Un ejemplo típico de clase inmutable en JAVA es la clase **String**. Una vez se ha creado una String, no se pueden modificar ninguno de sus caracteres sin crear una nueva instancia. Cualquier operación que se realice en un objeto String, como concatenar, cambiar mayúsculas o minúsculas, etc, genera un nuevo objeto String con los cambios realizados.

```
String s = "Hola";
s = s + " Mundo!"; // Crea un nuevo objeto con el contenido "Hola Mundo!"
```

Strings en JAVA

- En JAVA existen dos formas de crear un objeto de la clase String: mediante una asignación literal, y mediante el uso del constructor new String().

```
String cadena1 = "Hola Mundo";
String cadena2 = new String("Hola Mundo");
```

- La principal diferencia entre estas dos formas es que, cuando se crea una String mediante una asignación literal, la JVM busca en su pool de Strings si existe una String igual y, si existe, retorna su dirección en memoria. Si no existe, la JVM crea una nueva String en el pool de Strings y retorna su dirección en memoria.

- En cambio, cuando se crea una String mediante el constructor new String(), se crea un nuevo objeto String en el heap de memoria, independientemente de si existe o no una String igual en el pool de Strings. Por lo tanto, es recomendable utilizar asignaciones literales para crear Strings porque son más eficientes, ya que evitan la creación de objetos innecesarios en el heap. Sin embargo, si necesitas crear una copia de una String o crear una String a partir de otro tipo de datos, debes utilizar el constructor **new String()**.

Strings en JAVA

- En Java, existen dos formas de crear un objeto de la clase String: mediante una asignación literal y mediante el uso del constructor new String().

```
String s1 = new String("Hola Mundo");
String s2 = new String("Hola Mundo");

if(s1==s2) System.out.println("Iguales");
else System.out.println("Distintas");           //Distintas

if(s1.equals(s2)) System.out.println("Iguales"); //Iguales
else System.out.println("Distintas");
```

Aunque s1 y s2 apunten al mismo espacio de memoria, la inmutabilidad de las String hará que las modificaciones en s1 no afecten a s2 y viceversa.

```
String s1 = "Hola Mundo";
String s2 = "Hola Mundo";

if(s1==s2) System.out.println("Iguales");        //Iguales
else System.out.println("Distintas");

if(s1.equals(s2)) System.out.println("Iguales"); //Iguales
else System.out.println("Distintas");
```

Clases inmutables en JAVA

Para hacer que una clase sea inmutable en Java, hay varias pautas a seguir:

1. **Declarar todas las variables de instancia como finales**: esto garantiza que una vez que se ha asignado un valor a una variable, no se puede cambiar.
2. **No proporcionar ningún método que permita modificar las variables de instancia(setters)** esto garantiza que el estado interno de la clase no puede ser modificado después de su creación.
3. **Declarar la clase como final**, esto permite que no se pueda extender una clase que permita modificar las variables de instancia.
4. **No proporcionar ningún método que devuelva una referencia(puntero) a una variable de instancia**: esto garantiza que una clase externa no puede modificar las variables de instancia de la clase inmutable.
5. **Si la clase tiene variables de instancia que son objetos, se deben crear copias de estos objetos en lugar de almacenar referencias.** Por lo tanto, cualquier getter de un objeto, deberá devolver una copia del objeto de la instancia, así se evita la modificación desde fuera de la clase, ya que realmente se trata de un puntero.

Clases inmutables en JAVA

```java
public final class ObjetoInmutable {
    private final int n;
    private final String s;
    private final Punto p;

    public ObjetoInmutable(int n, String s, Punto p) {
        this.n = n;
        this.s = new String(s);
        this.p = new Punto(p.getX(), p.getY());
    }

    public int getN() {
        return n;
    }
    public String getS() {
         return new String(s);
    }
    public Punto getPunto() {
        return new Punto(p.getX(), p.getY());
    }
}
```

```java
public final class Punto {

    private final int x;
    private final int y;

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

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

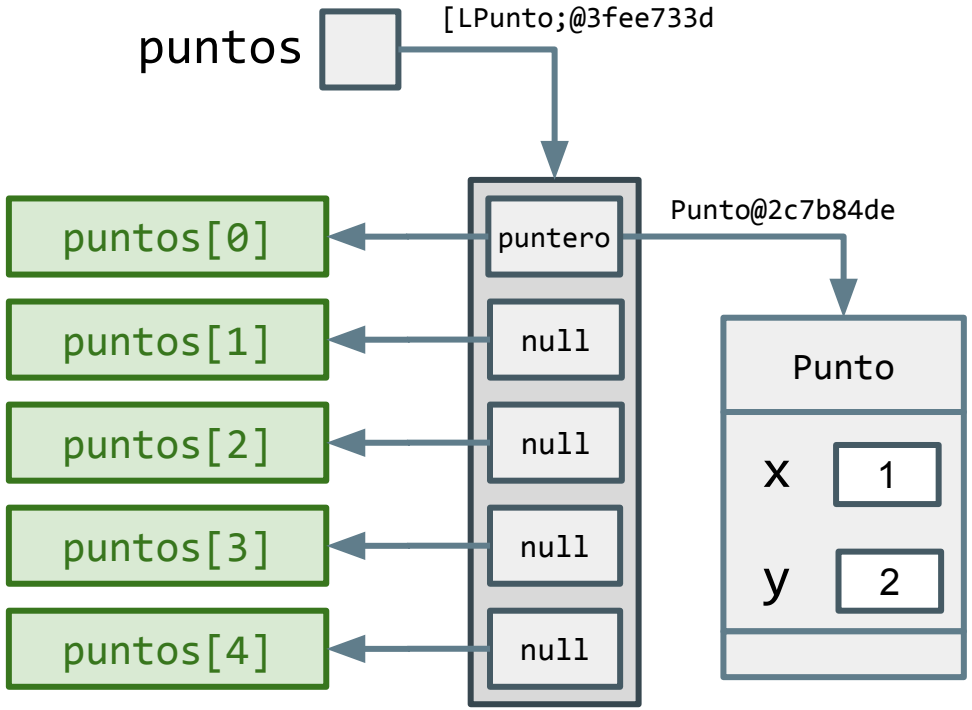
Arrays y ArrayLists de objetos

Arrays de objetos

El uso de Arrays no tiene por qué restringirse a elementos de tipo primitivo. Por ejemplo, pueden crearse referencias a arrays de la clase Punto:

```
Punto[] puntos = new Punto[5];
puntos[0] = new Punto(1,2);
```

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 Punto y se almacena su dirección de memoria en el primer elemento del array de punteros.



Arrays de objetos

```
public class ArrayPuntos {
    public static void main (String[] args) {
        //creamos 5 puntos aleatorios
        Random r = new Random();
        Punto[] puntos = new Punto[5];
        for (int i = 0; i < puntos.length; i++) {
            puntos[i] = new Punto(r.nextInt(100), r.nextInt(100));
            puntos[i].mostrarDatos();
        }
        // obtenemos la distancia del punto más lejano del centro(0,0)
        double max = 0;
        for (int i = 0; i < puntos.length; i++) {
            if (max < puntos[i].calcularDistanciaCentro())
                max = puntos[i].calcularDistanciaCentro();
        }
        System.out.println("La distancia máxima del centro es " + max);
    }
}
```

ArrayList de objetos

Un **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 fijado al inicio. La declaración de un ArrayList se hace según el siguiente formato:

ArrayList<nombreClase> nombreDeLista = new ArrayList<>();

- Entre **< >** indicamos la clase o tipos básicos de los objetos que se almacenarán.

```
ArrayList<Integer> listaNums = new ArrayList<>();
ArrayList<Boolean> listaBooleans = new ArrayList<>();
ArrayList<Double> listaDoubles = new ArrayList<>();
ArrayList<String> listaPalabras = new ArrayList<>();
ArrayList<Punto> listaPuntos = 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;
```

ArrayList: insertar elementos

El método **add** de la clase ArrayList posibilita añadir elementos.Se colocan después del último elemento que hubiera en el ArrayList

```
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
```

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.

```
...
//El orden hasta ahora es: España, Francia, Portugal
listaPaises.add(1, "Italia");
//El orden ahora es: España, Italia, Francia, Portugal
```

ArrayList: consultar un elemento

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

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

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

Podemos utilizar el ArrayList con cualquier tipo de datos, incluso con objetos que hayamos creado:

```
ArrayList<Punto> puntos = new ArrayList<>();
puntos.add(new Punto());
puntos.add(new Punto(1));
puntos.add(new Punto(1,3));

puntos.get(2).mostrarDatos(); //sobre la instancia con
coordenadas(1,3), se invocará al método mostrarDatos
```

ArrayList: eliminar elemento

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.

```
ArrayList<String> listaPaises = new ArrayList();
listaPaises.add("España");
listaPaises.add("Francia");
listaPaises.add("Portugal");
//listaPaises → España, Francia, Portugal
listaPaises.add(1, "Italia");
//listaPaises → España, Italia, Francia,
Portugal
listaPaises.remove(2);
//listaPaises → España, Italia, Portugal
listaPaises.remove("Portugal");
//listaPaises → España, Italia
```

```
ArrayList<Punto> puntos;
puntos = new ArrayList();
Punto p1 = new Punto(1,2);
Punto p2 = new Punto(1);
Punto p3 = new Punto(1,3);
Punto p4 = new Punto(3,3);
puntos.add(p1); //p1
puntos.add(p2); //p1,p2
puntos.add(p3); //p1,p2,p3
puntos.add(1, p4); //p1,p4,p2,p3
puntos.remove(2);  //p1,p4,p3
puntos.remove(p1); //p4,p3
```

ArrayList: modificar y buscar un elemento

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

```
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, si no lo encuentra devuelve -1.

```
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 del ArrayList se utiliza 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 típica un ArrayList.

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

```
System.out.println(listaPaises.toString());
```

Otros métodos de interés: Clase ArrayList

- **void clear()**: Borra todo el contenido de la lista.
- **Object clone()**: Retorna una copia de la lista.
- **boolean contains(Object o)**: 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.
- **int lastIndexOf(Object o)**: Devuelve el índice de la última ocurrencia del elemento. Si el elemento no se encuentra devuelve -1.
- **void sort(Comparator<Object>)**: Permite la ordenación de los elementos de las lista en función del Comparator< > pasado como parámetro, lo veremos más adelante.

Clases inmutables en JAVA : Array y ArrayList

Clases inmutables en JAVA con Array

```
public final class ObjetoInmutable2 {

    private final Punto[] arrayPuntos;

    public ObjetoInmutable2 (Punto[] arrayPuntos) {
        this.arrayPuntos = arrayPuntos.clone();
    }

    public Punto[] getArrayPuntos() {
        return arrayPuntos.clone();
    }
}
```

```
public final class Punto {

    private final int x;
    private final int y;

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

    public int getX() { return x;  }
    public int getY() { return y;  }
}
```

Clases inmutables en JAVA con Array

```
public final class ObjetoInmutable2 {

    private final Punto[] arrayPuntos;

    public ObjetoInmutable2(Punto[] arrayPuntos) {
        this.arrayPuntos = obtenerCopia(arrayPuntos);
    }

    public Punto[] getArrayPuntos() {
        return obtenerCopia(arrayPuntos);
    }

    private Punto[] obtenerCopia(Punto[] array) {

        Punto[] copia = new Punto[array.length];
        for (int i = 0; i < copia.length; i++) {
            copia[i] = new Punto(array[i].getX(), array[i].getY());
        }
        return copia;
    }
}
```

```
public class Punto {
    private int x;
    private int y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x;  }
    public int getY() { return y;  }
    public void setX(int x) {
         this.x = x;
    }
    public void setY(int y) {
         this.y = y;
     }
}
```

Clases inmutables en JAVA con ArrayList

```
public final class ObjetoInmutable3 {

    private final ArrayList<Punto> listaPuntos;

    public ObjetoInmutable3 (ArrayList<Punto> listaPuntos) {
        this.listaPuntos = new ArrayList<>(listaPuntos);
    }

    public ArrayList<Punto> getListaPuntos() {
        return new ArrayList<>(listaPuntos);
    }
}
```

```
public final class Punto {

    private final int x;
    private final int y;

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

    public int getX() { return x;  }
    public int getY() { return y;  }
}
```

Clases inmutables en JAVA con ArrayList

```
public final class ObjetoInmutable3 {

    private final ArrayList<Punto> listaPuntos;

    public ObjetoInmutable3 (ArrayList<Punto> listaPuntos)
        this.listaPuntos = obtenerCopia(listaPuntos);
    }

    public ArrayList<Punto> getListaPuntos() {
        return obtenerCopia(listaPuntos);
    }

    private ArrayList<Punto> obtenerCopia(ArrayList<Punto> lista) {

        ArrayList<Punto> copia = new ArrayList<>();
        for (int i = 0; i < lista.size(); i++) {
             copia.add(new Punto(lista.get(i).getX(), lista.get(i).getY()));
        }
        return copia;
    }
```

```
public class Punto {
    private int x;
    private int y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {…}
    public int getY() {…}
    public void setX(int x) {…}
    public void setY(int y) {…}
}
```

Ejercicios de refuerzo OO

Ejercicios de refuerzo OO

Una empresa informática necesita llevar un registro de todos sus empleados que se encuentran en la oficina central, para ello, se debe crear una clase que incluya lo siguiente:

Atributos: nombre completo, permanencia(número de años en la empresa) y salario.

Métodos:

1. Constructor con y sin parámetros de entrada.
2. Método que permita mostrar la clasificación según la la permanencia de acuerdo al siguiente algoritmo:
   a. Si la permanencia es menor o igual a 3, "***Principiante***".
   b. Si la permanencia es mayor que 3 y menor que 18, "***Intermedio***".
   c. Si la permanencia es mayor o igual a 18, "***Senior***".
3. Método que muestre los datos del empleado por pantalla, incluida la permanencia(se debe utilizar salto de línea para separar los atributos).
4. Un método que permita aumentar el salario en un porcentaje que sería pasado como parámetro al método.

Implementa también un main para probar lo implementado anteriormente.

Ejercicios de refuerzo OO

Crea la clase **GraficoLineas2D**, para simular un gráfico formado por puntos en un plano de 2 dimensiones, se deben tener en cuenta las siguientes consideraciones:

- El gráfico estará formado por un ArrayList de Puntos, en principio estará vacío.
- Las coordenadas de todos los puntos deben ser positivas.
- En la lista no puede existir ningún punto en la misma posición que otro.
- El orden de los puntos es muy importante, a medida que se avance en la lista, **las coordenadas en el eje de las x deberán ser siempre igual o mayor al siguiente punto de la lista**. A continuación se muestran ejemplos de Gráficos válidos y no válidos.

Se deberán crear métodos que permitan tanto añadir como eliminar puntos del gráfico. Siempre se añadirán y eliminarán por el final.Se debe informar tanto de la posible inserción, como del posible error, si el punto no cumple con las restricciones del problema.

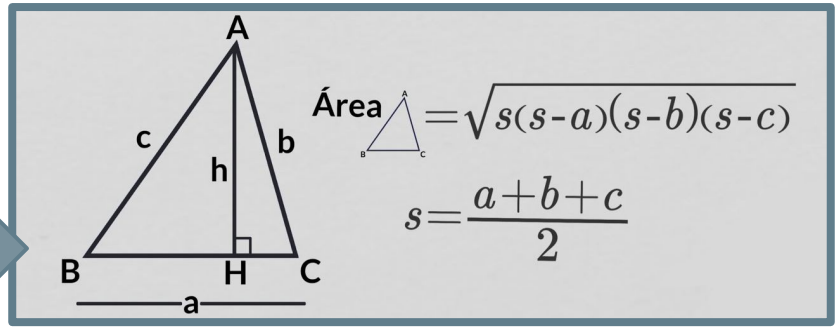
Añadir un método que permita hacer un set del ArrayList de puntos, para ello se deberá comprobar previamente si dicha lista cumple con los requisitos del problema, de lo contrario, no se modificará la lista y se avisará por pantalla. Utilizar dicho método para crear un constructor adicional que reciba el ArrayList de Puntos.

Ejercicios de refuerzo 00

Modifica la clase **Punto** desarrollada a lo largo de la unidad para incluir los siguientes métodos:

1. Un método que compruebe si dos puntos están en distinta posición.
2. Ampliar el método anterior, para que reciba una cantidad indeterminada de puntos a comparar. Es decir, si le pasamos 4 puntos, devolverá un TRUE si los 4 puntos se encuentran en posiciones distintas.
3. Un método que obtenga la distancia entre dos puntos A y B.
4. Reutilizar los métodos anteriores para crear un método que obtenga a partir de 3 puntos, el área del triángulo que formarán dichos puntos. Se deben tener en cuenta las siguientes consideraciones:

    a. Los tres puntos deben tener una posición única.
    b. Además, se deberá indicar el tipo de triángulo que forman los 3 puntos según sus lados (equilátero, escaleno, isósceles).
    c. El área del triángulo se puede obtener con la fórmula de Herón.



Ejercicios de refuerzo OO

Construir una clase **Racional** que permita representar y manipular números racionales. Un número racional permite representar la relación a/b entre dos números enteros. La clase tendrá dos atributos: numerador y denominador.

Para utilizar dicha clase, implementa un menú en un main que contenga las siguientes opciones:

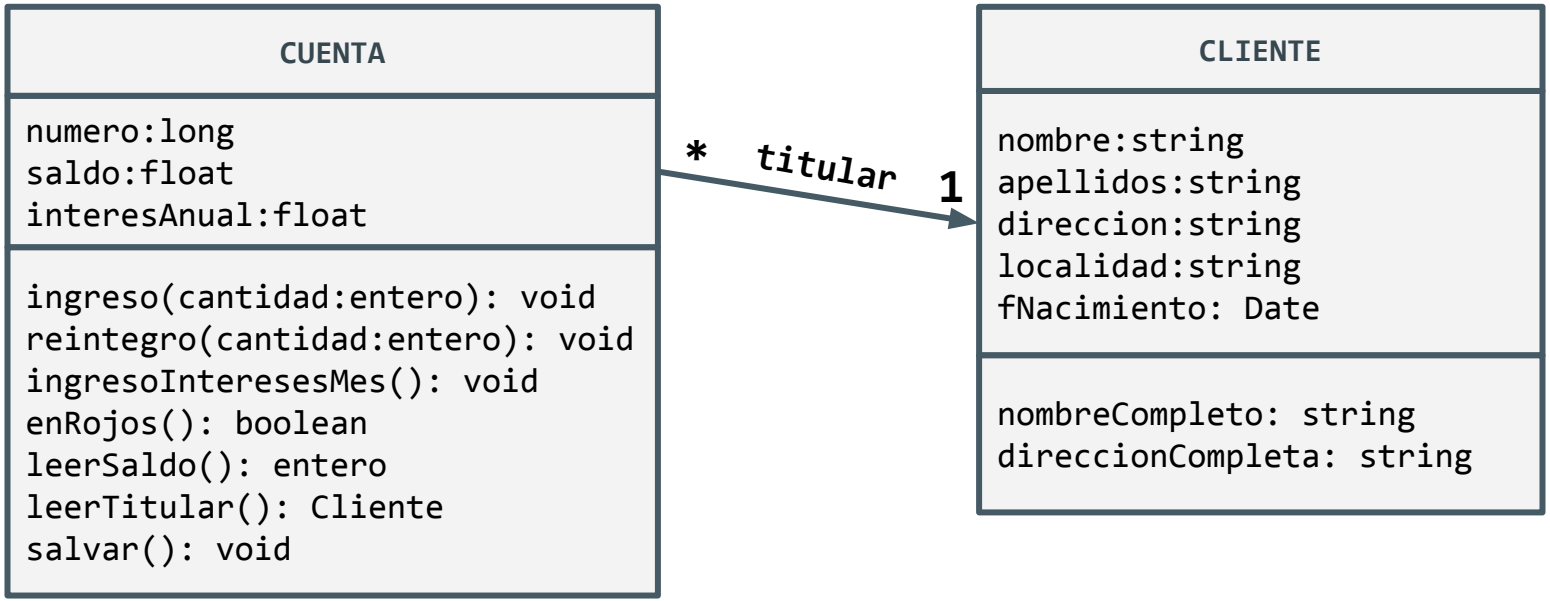
1. Introduce número A
2. Introduce número B
3. Suma de A y B
4. Resta de A y B
5. Multiplicación de A y B
6. División de A y B
7. Son iguales A y B
8. Salir

Se deben implementar los métodos que se consideren necesarios para el correcto funcionamiento de la aplicación.

Ejemplo 1 con Diseño orientado a objetos: 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:

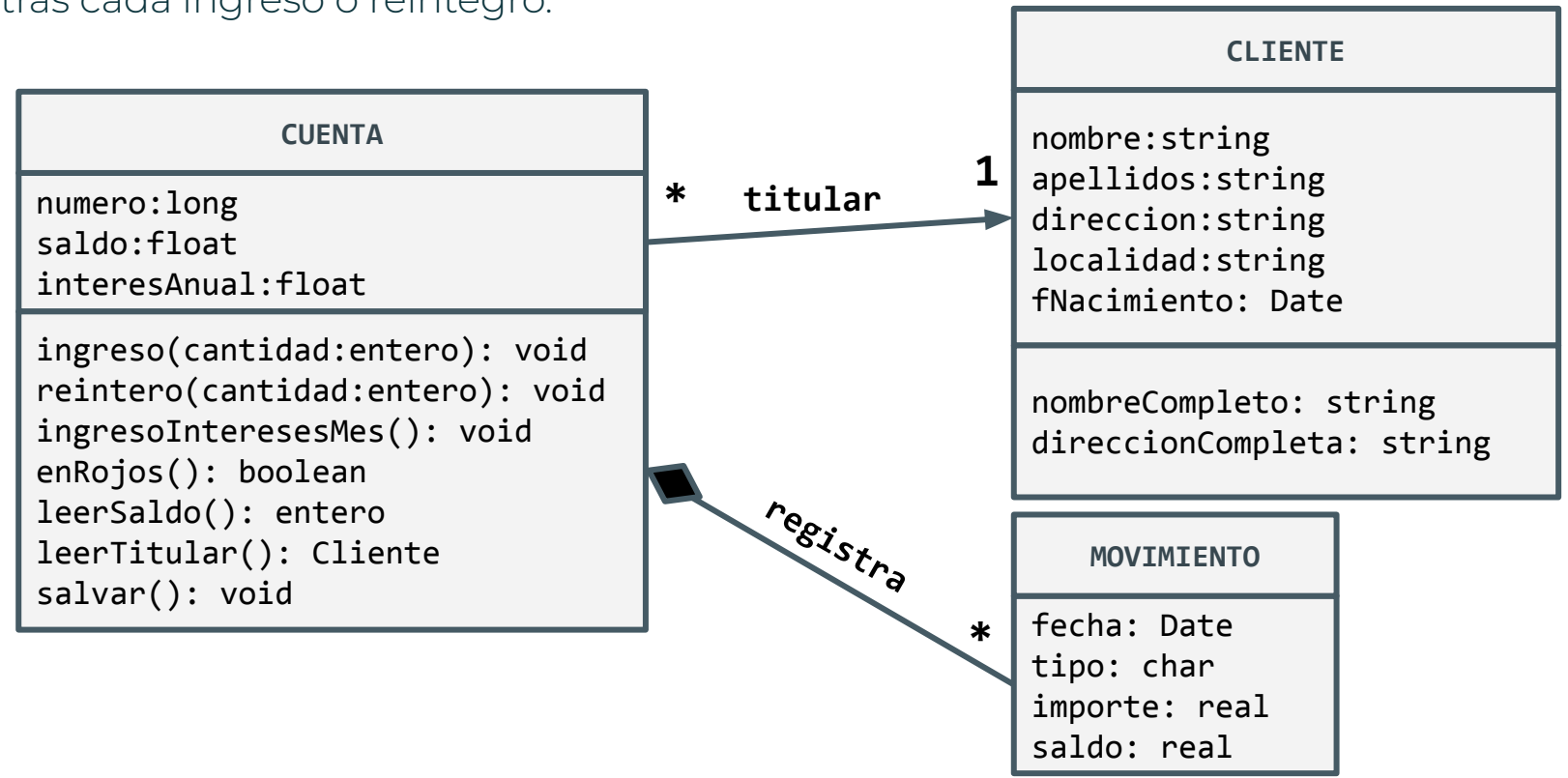
```
public class Cuenta {
private long numero;
private Cliente titular;
private float saldo;
private float interesAnual;

// 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 = alipo; 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 asignan 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

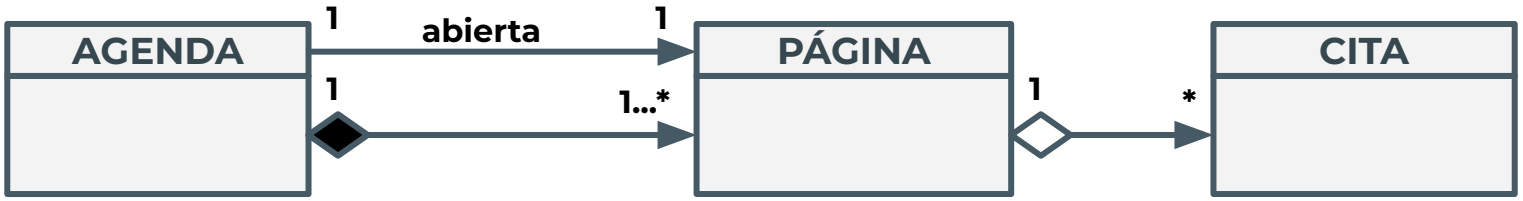
- Si el usuario selecciona la opción 2, se solicitará que introduzca nombre y apellidos del cliente y se buscará en el Array para ver si existe alguna Cuenta de ese Cliente.
    - Si existe, se pondrá esa cuenta como activa y se mostrará el submenú.
    - Si no existe, se avisará al usuario de que no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

Ejemplo 2 con Diseño orientado a objetos: Agenda

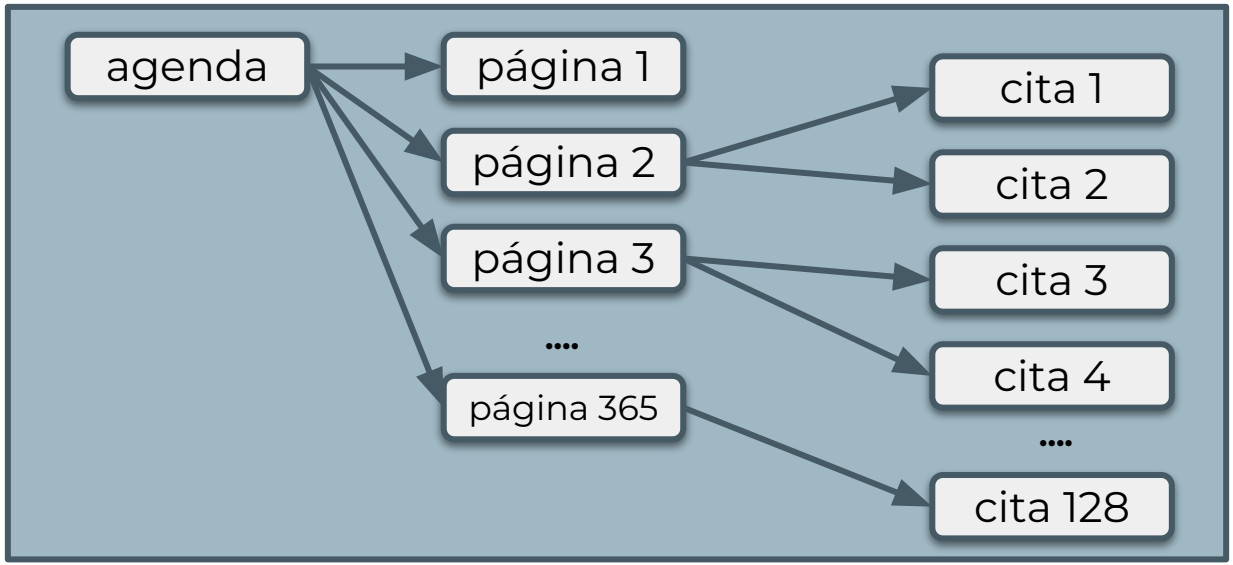
Ejemplo: Agenda

- 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.
- 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
anyo:entero
avanzarPagina() : void retrocederPagina() : void leerPagina(): Pagina

PÁGINA
dia:entero mes:entero
agregarCita(c: Cita) : void borrarCita(c: Cita) : void buscarCita(título:string): Cita verCitas(): List<Cita>

CITA
hora:entero minutos:entero titulo:string texto:string
modificarTexto(texto:string): void

Ejemplo: Agenda

- **El protocolo para escribir una cita**, es el siguiente:
  - A partir del objeto agenda: se pasa de página hasta llegar a la que corresponde a la fecha escogida, usando las operaciones avanzarPagina y retrocederPagina.
  - Cada vez que se pasa de página, se puede ver cuál es la página actual con la operación llegirPàgina.
  - Una vez se obtiene el objeto de la página actual, se puede inspeccionar el contenido mediante las operaciones accesorios de la clase Página
  - Para cada página, se pueden visualizar todas las citas existentes con la operación listarCitas.
  - Si esta es la página en la que se quiere añadir una cita, deberá instanciar un objeto nuevaCita: Cita, inicializando todos sus atributos al valor que corresponda.
  - Finalmente, hay que escribir la cita llamando sobre el objeto de la página actual la operación agregarCita (nuevaCita). Evidentemente, esta operación tiene que controlar que no haya solapamientos de hora entre las citas escritas en la página.

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