DISEÑO ORIENTADO A OBJETOS en UML

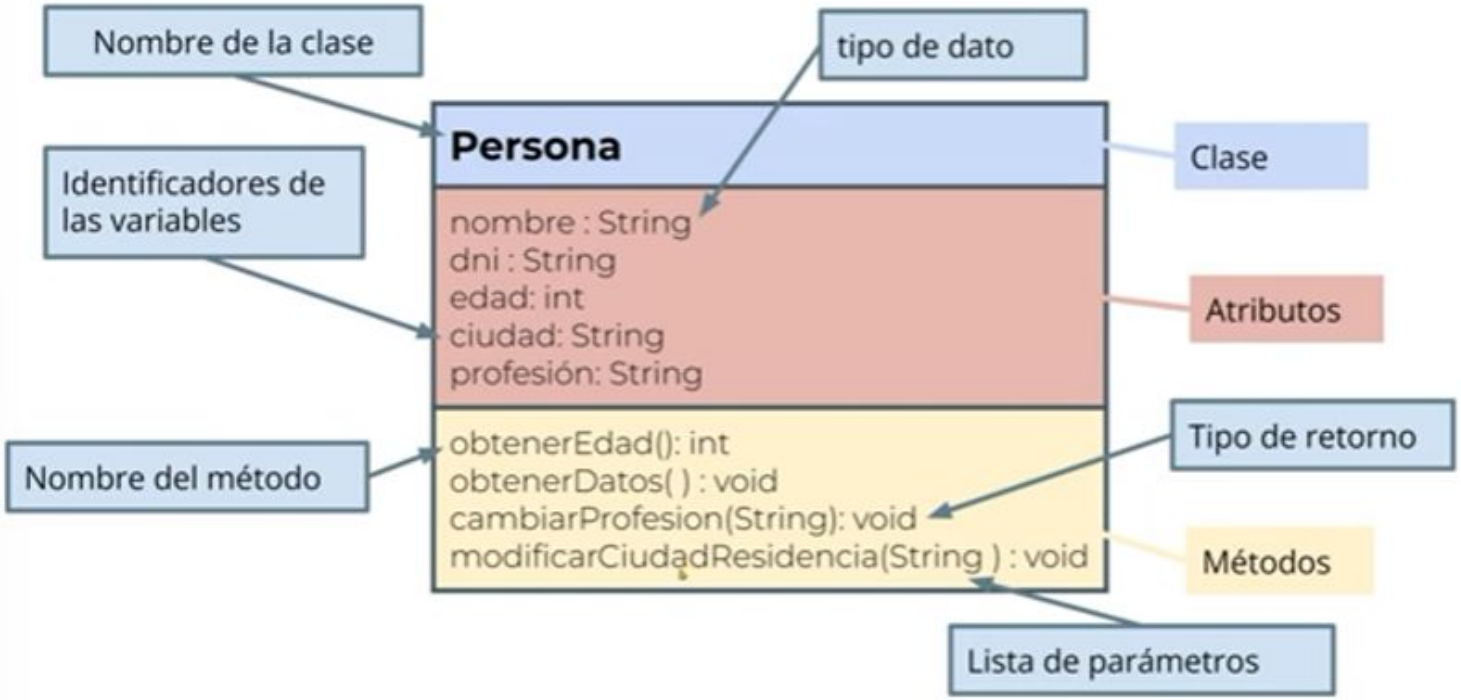
Entornos de desarrollo
1ºDAW

INTRODUCCIÓN

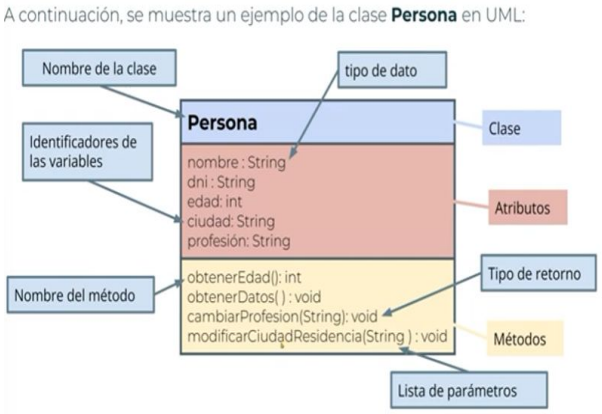
- El **UML** (Unified Modelling Language o Lenguaje de Modelado Unificado) se utiliza para establecer cómo se estructura la resolución de un problema mediante la orientación a objetos. Además, se utiliza para saber de qué manera interactúan los diferentes componentes para lograr una tarea concreta.
- 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

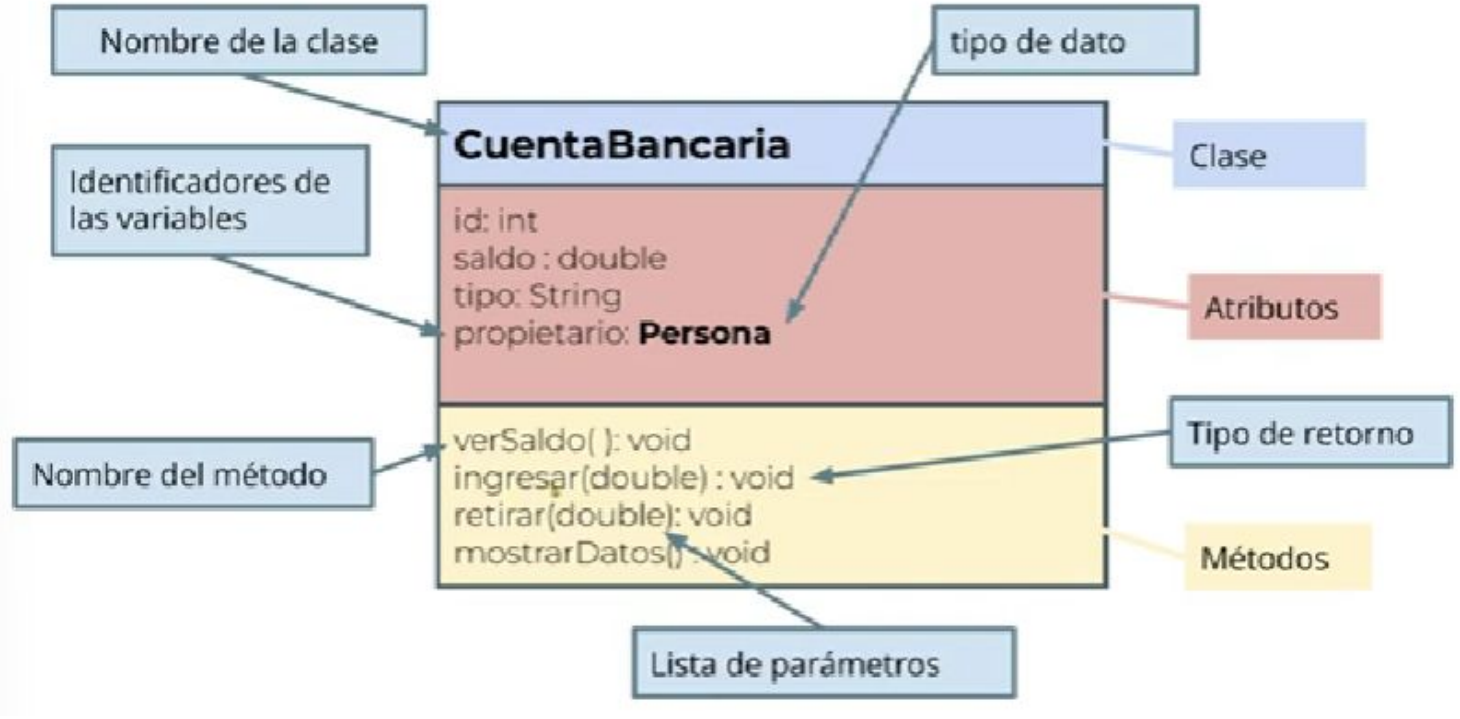


Clase persona representada en UML.
Estructura: rectángulo con 3 cajitas:

- nombre,
- atributos (con tipo de dato que guarda) y
- métodos (con tipo de parámetros y tipo de retorno o void si no devuelve nada).

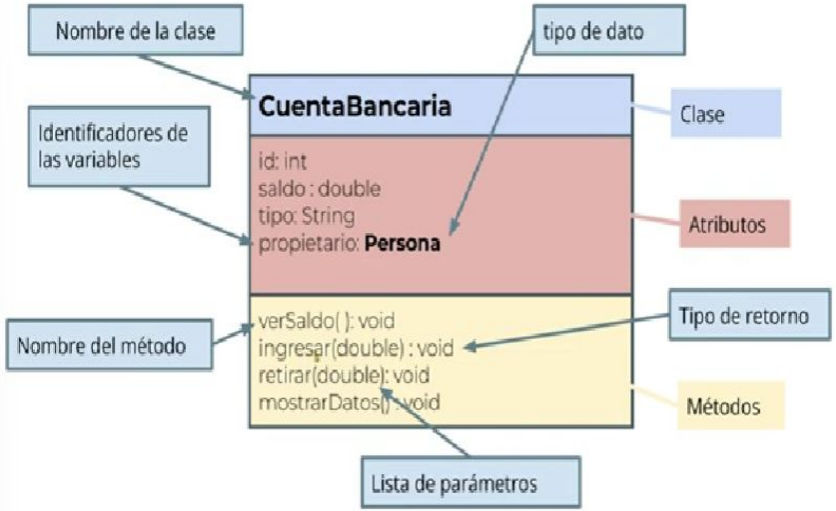
Ejemplo de clase en UML

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



Ejemplo de clase en UML

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



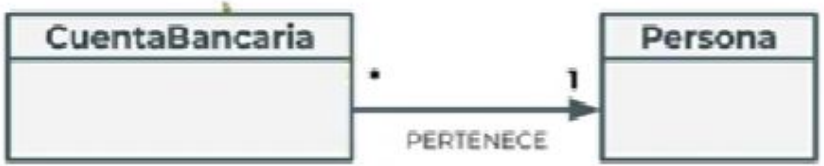
Clase CuentaBancaria.

Atributo cuyo tipo de dato es una Persona (clase creada). Por tanto, propietario es una instancia de la clase **Persona**.

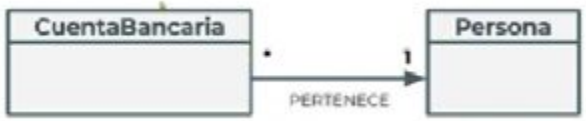
Ahora veremos cómo se **relacionan** estas clases.
En este ejemplo, cada cuentabancaria tiene sólo una persona propietaria, pero una persona podrá tener varias cuentas bancarias: **cardinalidad**.

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



Relación con cardinalidad
Flecha con 1 de cuentabancaria a persona: dentro de la cuentabancaria habrá 1 única persona.
* Una persona podrá tener de 0 a n cuentas bancarias

Cada **asociación** llevará el nombre y flecha con navegabilidad (origen y destino).
Si no hay flecha, será **bidireccional** y ambas podrán tener una instancia de la 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.

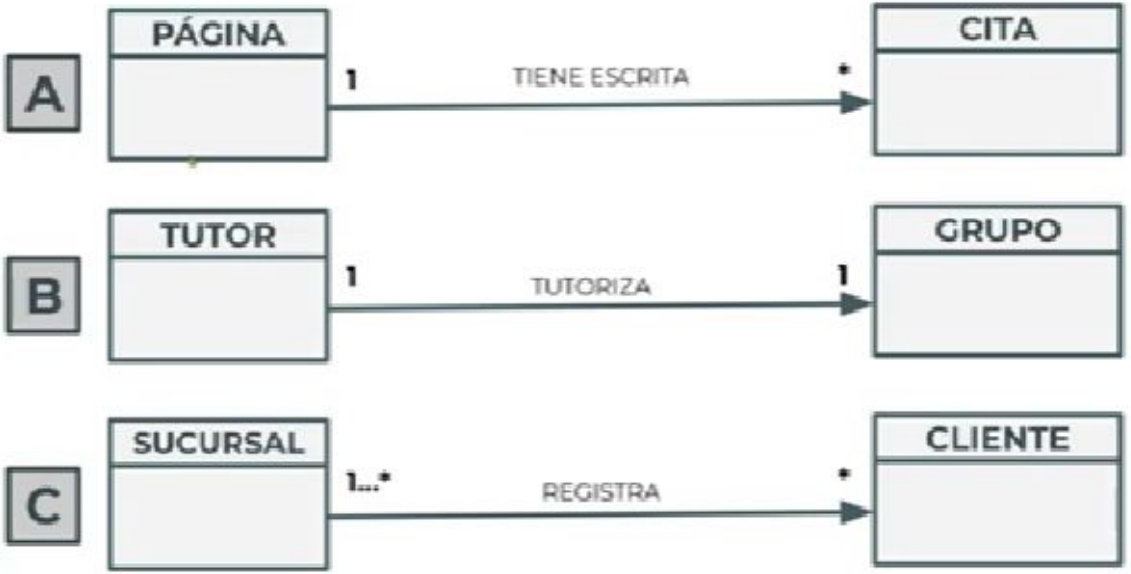
Relaciones entre clases

- **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.

TIPOS DE ASOCIACIÓN

- AGREGACIÓN
- COMPOSICIÓN
- REFLEXIVA

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.

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.

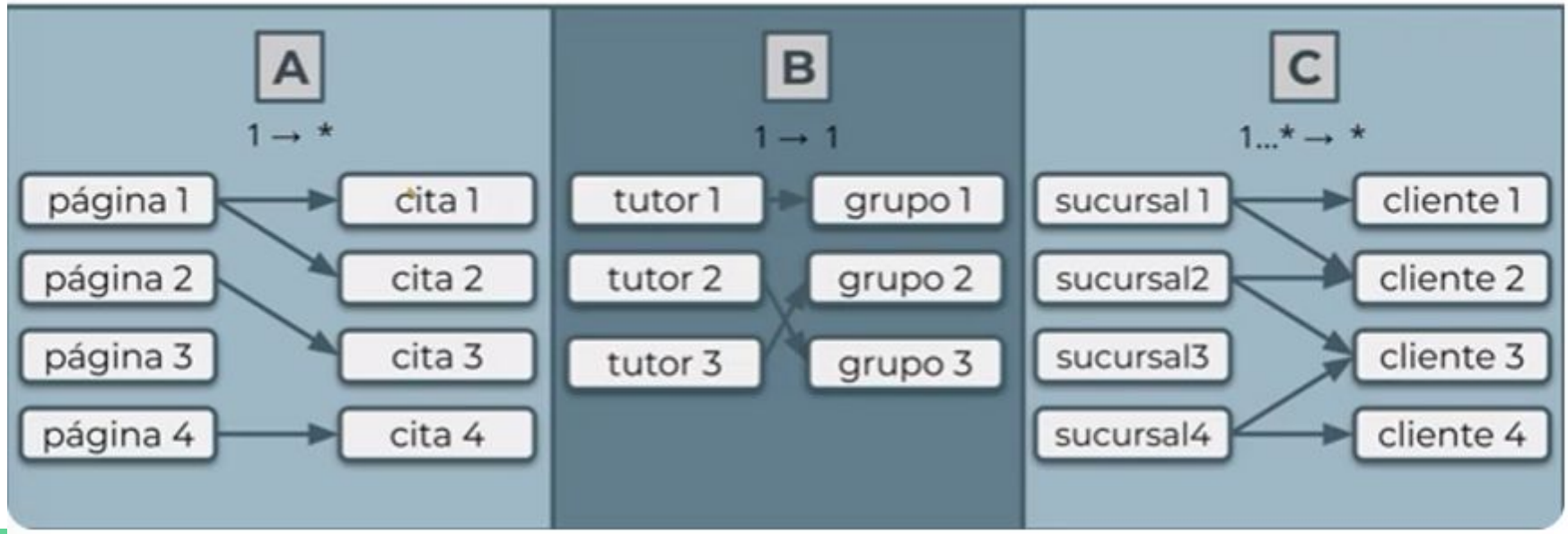
Un cliente puede recomendar a un número indeterminado de clientes, o a nadie.
Un cliente puede haber sido recomendado o no.

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.

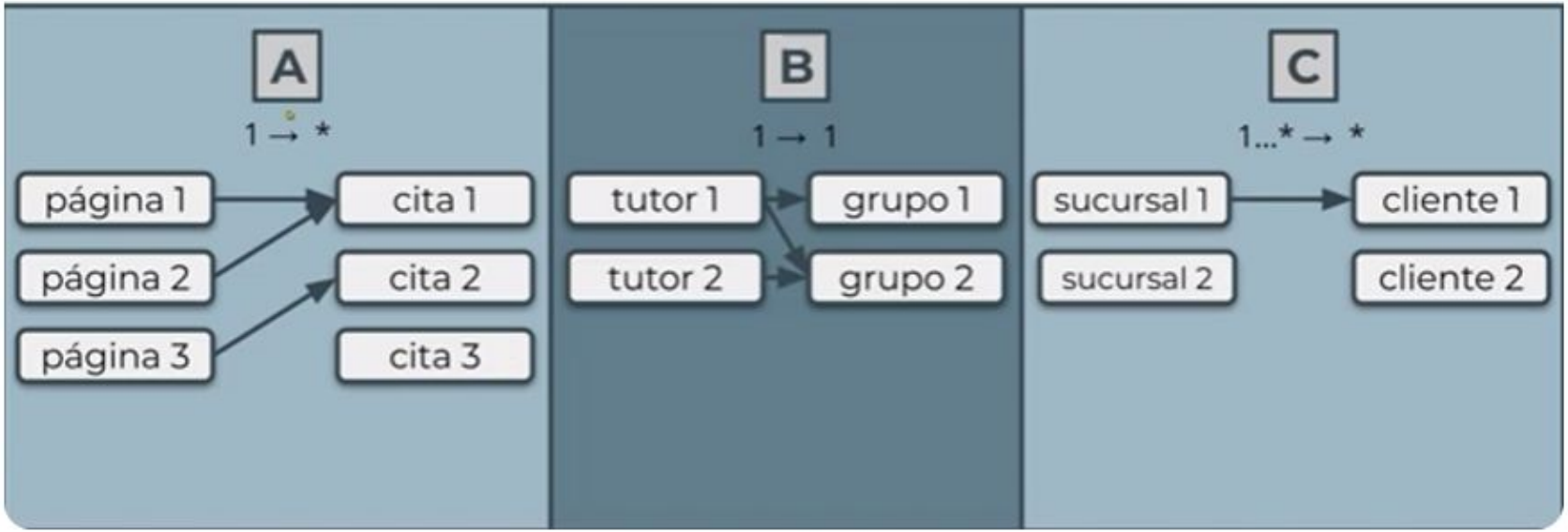
MAPA DE OBJETOS

- 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

- 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
}
```

clase

atributos

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
        "Circulo 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());
    }
}
```

Ejemplo de cómo una clase está creando objetos (instancias de tipo círculo) de otra clase. El main utiliza la clase anterior. Crea un círculo usando el constructor de la clase:

Circulo c1 = new Circulo(); Tipo de objeto que quiero crear al que llamaré c1, y lo creará usando el constructor. Con radio 50, etc.

c1.setRadio(2.9); Cambia el radio del círculo.

Y al usar toString concatena varios valores del círculo.

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   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;
    }
}
```

VISUAL STUDIO CODE

- Copiar código de Círculo en VSC: circulo.java y ejemplocirculo.java
- Cambiar el método toString a mostrarDatos y que imprima por pantalla directamente en vez de devolver la cadena (void). Probamos que al llamarlo obtenemos el mismo resultado
- Cambiar setRadio e impedir que se pueda cambiar el radio a uno negativo. (if radio < 0...) En este caso, el radio valdrá 0. También podríamos cambiar el radio sólo en el caso en el que sea >=0.
- Crear otro círculo c2. Cambiar el radio de c1 a 35. Mediante setRadio y getRadio, asignar a c2 el radio de c1.
- Crear dos constructores (al menos un parámetro distinto): Circulo () y Circulo (double r). Este último de asingara al radio el valor del parámetro r.

VISUAL STUDIO CODE

- Diseño orientado a objetos en UML ☕ DAM - DAW
- JAVA: Mapas de objetos ☕ DAM - DAW
- JAVA: Introducción a la POO ☕ DAM - DAW
- JAVA: Ejemplo clase Círculo ☕ DAM - DAW