CLASE 2

Propiedades de los objetos

Encapsulamiento. Ocultamiento de la Información

Modos de acceso

Métodos como control de acceso

Acceso a los atributos privados.

Cuál es el problema del atributo público?

Control de acceso por clase

Métodos getXXX() y setXXX()

Sobrecargando métodos
Miembros “static” (atributos, métodos)

Constantes

Constantes objetos (no primitivos)

Constantes – cambiar el estado del objeto referido

Colector de basura y “finalize”

Finalize

La relación de asociación: agregación y composición

Diferencias entre agregación y composición

Introducción a los Requerimientos.
Introducción a las Inspecciones, recorridos de prueba, checklist.

Propiedades de los objetos

Abstracción

Significa **extraer las propiedades esenciales de un objeto** que lo distinguen de los demás tipos de objetos y proporciona fronteras conceptuales definidas respecto al punto de vista del observador.

Encapsulamiento

Es la propiedad que permite ocultar al mundo exterior la representación interna del objeto. Esto quiere decir que el objeto puede ser utilizado, pero los datos esenciales del mismo no son conocidos fuera de él.

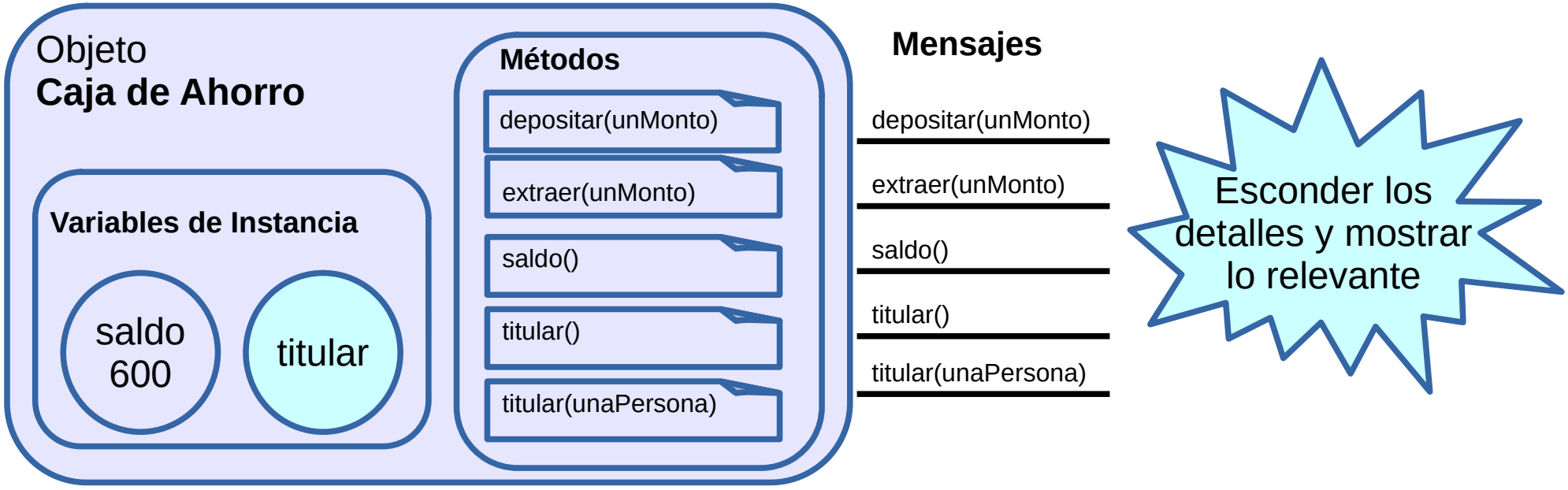
Herencia

Es la propiedad que permite a los objetos **construirse a partir de otros objetos**. La clase base contiene todas las características comunes. Las sub-clases contienen las características de la clase base más las características particulares de la sub-clase. Si la sub-clase hereda características de una clase base, se trata de herencia simple. Si hereda de dos o mas clases base, herencia multiple.

Polimorfismo

Literalmente significa "**cualidad de tener mas de una forma**". En poo, se refiere al hecho que una misma operación puede tener diferente comportamiento en diferentes objetos. En otras palabras, diferentes objetos reaccionan al mismo mensaje de modo diferente.

Encapsulamiento. Ocultamiento de la Información



Distingue el “cómo hacer” del "qué hacer"

La especificación es visible al usuario, mientras que la implementación se le oculta.

Modos de acceso

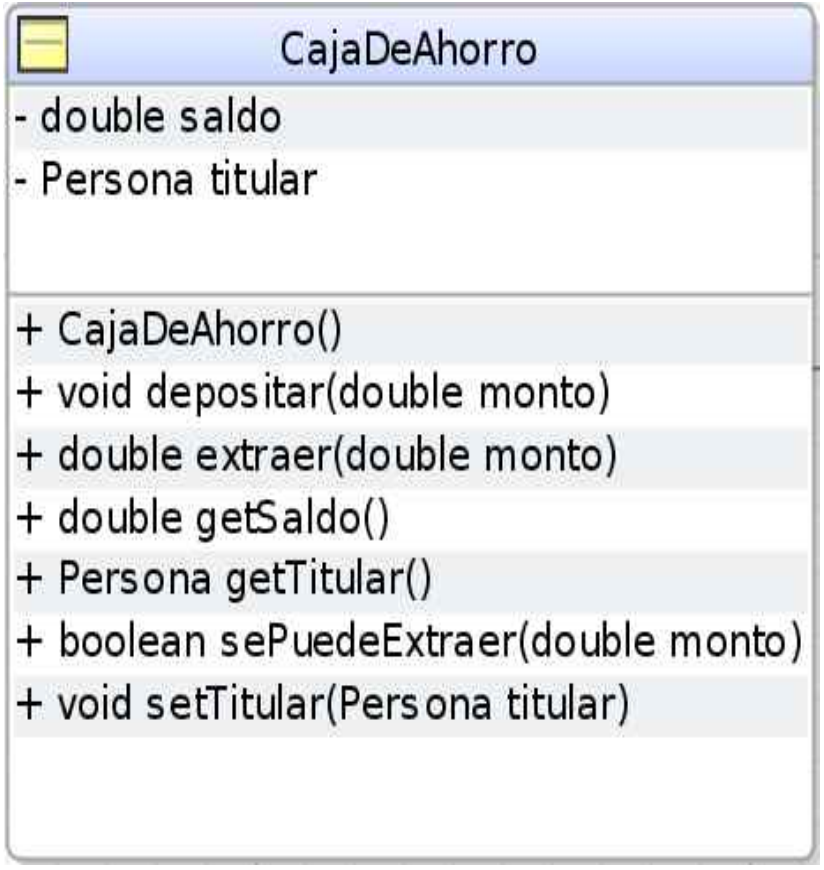
Es la forma de representar el **encapsulamiento**

Público (+): Atributos o Métodos que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella.

Privado (-): Atributos o Métodos que solo son accesibles dentro de la implementación de la clase.

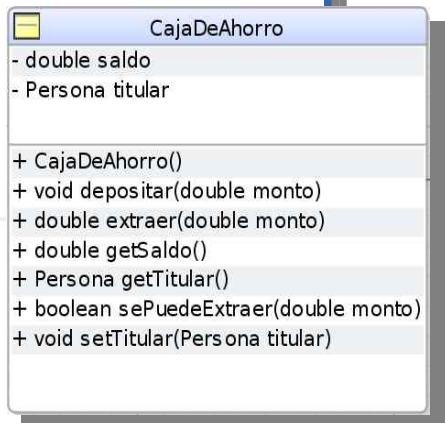
Protegido (#): Atributos o Métodos que son accesibles para la propia clase y sus clases hijas (subclases). Luego se ampliará la definición.

Los atributos y los métodos que son públicos constituyen lo que el mundo exterior conoce de la misma.
Normalmente lo usual es que se oculten los atributos de la clase y solo sean visibles los métodos, *incluyendo entonces algunos métodos de consulta para ver los valores de los atributos*.



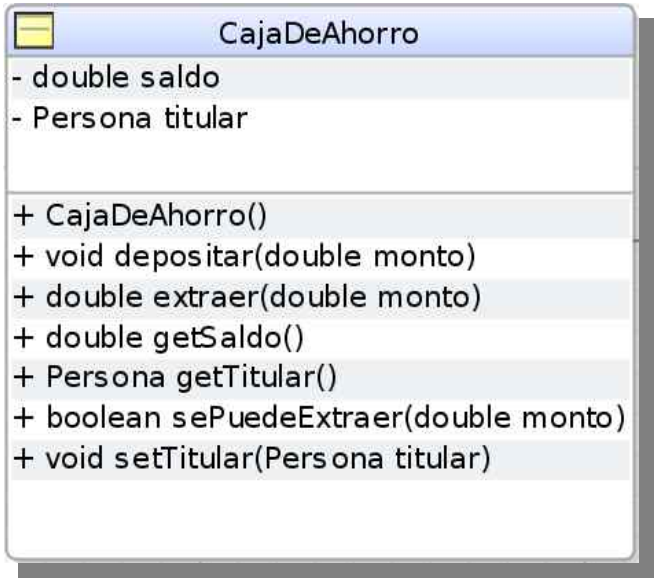
Modos de acceso

```
public class CajaDeAhorro
{
    private double saldo;
    private Persona titular;

    public CajaDeAhorro()
    {
        super();
    }

    public void depositar (double monto)
    {
        this.saldo = this.saldo + monto;
    }

    public double extraer (double monto)
    {
        this.saldo = this.saldo - monto;
        return saldo;
    }

    public double getSaldo()
    {
        return this.saldo;
    }

    public Persona getTitular()
    {
        return this.titular;
    }

    public void setTitular(Persona titular)
    {
        this.titular = titular;
    }

    public boolean sePuedeExtraer(double monto)
    {
        return (this.getSaldo() > monto);
    }
}
```



Métodos como control de acceso

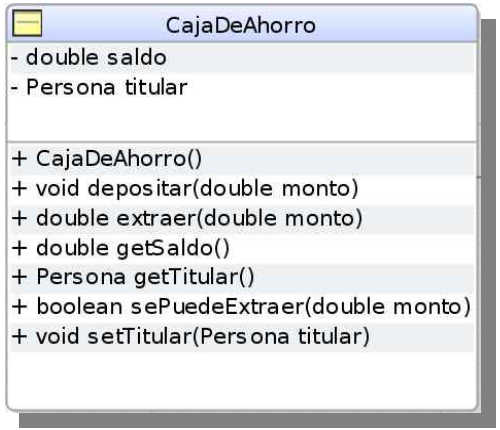
```
System.out.println("mi saldo es" +
unaCajaDeAhorro.saldo)
```



- ✔ *Cómo accedo a los atributos privados ?*
- ✔ *Por qué no hacerlos directamente públicos?*
- ✔ *get's y set's*

Acceso a los atributos privados

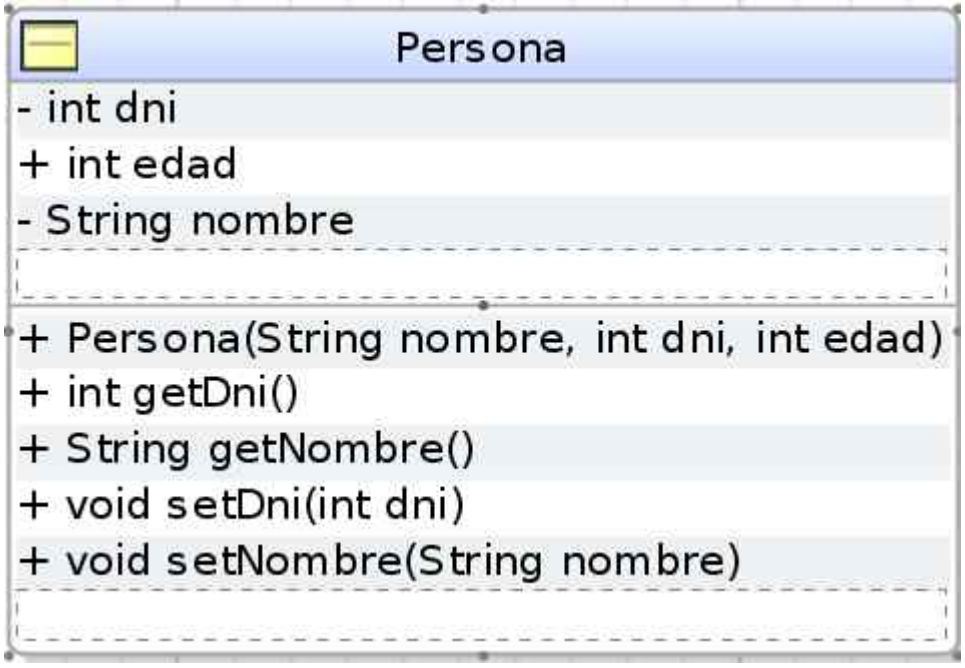
```
public class CajaDeAhorro
{
    private double saldo;
    private Persona titular;

    public double getSaldo()
    {
        return this.saldo;
    }

    public Persona getTitular()
    {
        return this.titular;
    }

    public void setTitular(Persona titular)
    {
        this.titular = titular;
    }
}
```



Se utilizan los métodos públicos como mecanismo para acceder a los atributos privados.

por qué hacerlo así ??...

Cuál es el problema del atributo público ?



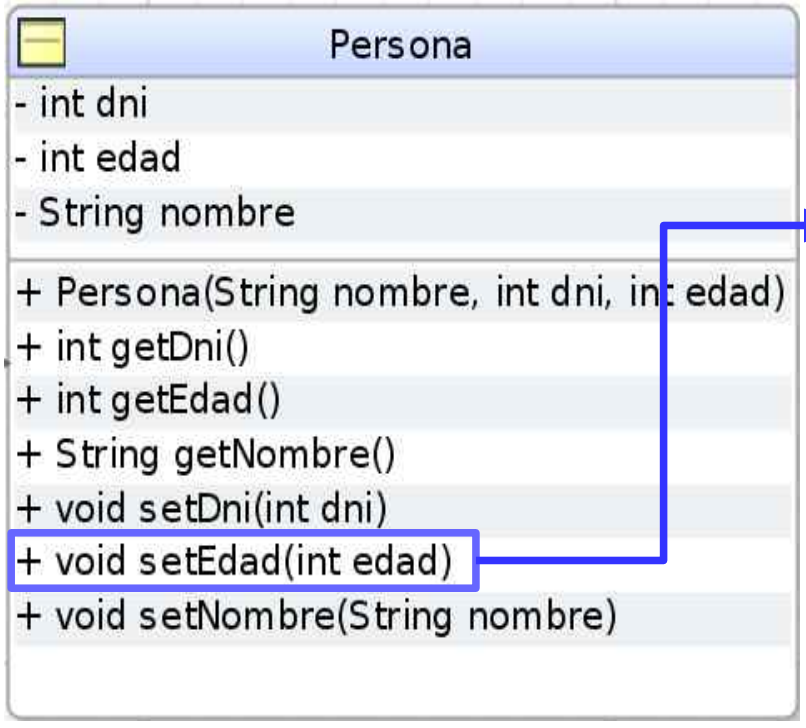
Qué problema surge si permitimos que el atributo edad sea público ?

```
unaPersona.edad = -7;
```

Cómo evito el error ?

La responsabilidad de validar el ingreso de datos corresponde a la propia clase, de otra forma, esa validación se vería trasladada a todo el código en donde se ingrese la edad de la persona.

Métodos como control de acceso




```java
public void setEdad(int edad)
{
    if(edad > 0)
        this.edad = edad;
    else
    {
        //establecer algún criterio de solución o
        //preparar el método para que lance
        //una excepción (se verá más adelante)
    }
}
```

De esta forma, la validación queda bajo la responsabilidad de la propia clase

El control de acceso se realiza por clase y no por objeto...

Ver libro de Gosling, capítulo 2

Métodos getXXX() y setXXX()

Convenio para acceder a los atributos de visibilidad privada

```java
public class Persona
{
    private String nombre;
    private int dni;
    private int edad;

    public Persona(String nombre, int dni, int edad)
    {
        super();
        this.nombre = nombre;
        this.dni = dni;
        this.edad = edad;
    }

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

    public String getNombre()
    {
        return nombre;
    }

    public void setDni(int dni)
    {
        this.dni = dni;
    }

    public int getDni()
    {
        return dni;
    }

    public void setEdad(int edad)
    {
        if(edad > 0)
            this.edad = edad;
        else
        {
            //establecer algún criterio de solución o
            //preparar el método para que lance
            //una excepción (se verá más adelante)
        }
    }

    public int getEdad()
    {
        return edad;
    }
}
```

Sobrecargando métodos

Signatura o firma: consiste del nombre y parámetros (número y tipos). Es toda la cabecera excepto el retorno.

Sobrecarga

Dos o más métodos pueden tener el mismo nombre si tienen diferente número o tipo de parámetros, y por lo tanto diferente signatura. El nombre del método tiene más de un significado.

Cuando se invoca el método, el compilador compara el número y tipo de parámetros para encontrar el método que mejor se ajustan entre las signaturas disponibles.
No se puede diferenciar un método de otro solamente si se diferencian en el tipo de retorno. O sea, no se puede sobrecargar solamente modificando el retorno.

Los constructores también pueden sobrecargarse

```java
public class Calculadora
{

    public int sumar(int x, int y)
    {
        System.out.println("Método1:");
        return x + y;
    }

    public float sumar(float x, float y)
    {
        System.out.println("Método2:");
        return x + y;
    }

    public float sumar(int x, float y)
    {
        System.out.println("Método3:");
        return x + y;
    }

}
```

Sobrecarga de métodos

```java
public static void main(String[] args)
{
    Calculadora c = new Calculadora();
    int tot1 = c.sumar(2, 4);
    System.out.println(tot1);
    float tot2 = c.sumar(2.3f, 4.2f);
    System.out.println(tot2);
    float tot3 = c.sumar(2, 4.2f);
    System.out.println(tot3);
}
```

```java
public class Box
{
    int x, y, ancho, alto;

    public Box(int x, int y, int ancho, int alto)
    {
        this.x = x;
        this.y = y;
        this.ancho = ancho;
        this.alto = alto;
    }

    public Box(int x, int y)
    {
        this(x, y, 10, 10);
    }

    public Box()
    {
        this(1, 1);
    }

    public void setAncho(int ancho)
    {
        this.ancho = ancho;
    }
}
```

Sobrecarga de Constructores

```java
public static void main(String[] args)
{
    Box a = new Box(5, 3, 100, 200);
    Box b = new Box(10, 5);
    Box c = new Box();
}
```

Miembros “static” (atributos, métodos)

Atributos estáticos

Son atributos compartidos por todos los objetos de una clase.

Los atributos específicos de clase se obtienen declarándolos como **`static`**.

No se necesita crear un objeto para poder acceder a estos atributos. Se puede acceder invocando el nombre de la clase seguida del operador de selección (.) y luego el nombre del campo.

Métodos estáticos

Un método estático se invoca en nombre de una clase completa, no sobre una instancia.
Puede acceder sólo a campos estáticos u otros métodos estáticos de la clase.
Dentro de un método estático no hay referencia a **`this`** (no existe el objeto).

Ejemplo

Supongamos que estamos desarrollando un sistema para un comercio, en el que tenemos clientes, productos y realizamos la facturación correspondiente a la compra de productos que hace un cliente.

Ejemplo: necesitamos que cada cliente tenga un número de cliente único

```java
public class Cliente
{
  /**
   * Atributo de clase, compartido por todas las instancias
   */
  public static int siguienteNro = 0;
  private int nroCliente;
  private String nombre;
  private String apellido;

  /**
   * Cada vez que se crea un cliente nuevo,
   * se incrementa en 1 el atributo de clase.
   * Luego se asigna ese valor al atributo nroCliente.
   */
  public Cliente()
  {
    siguienteNro++;
    nroCliente = siguienteNro++;
  }
  public static int getSiguienteNro()
  {
    return siguienteNro;
  }
  @Override
  public String toString()
  {
    return ("Cliente nro: " + nroCliente + " Apellido: " + apellido + "Nombre: " + nombre);
  }
```

```java
  //getters y setters
  public void setNroCliente(int nroCliente)
  {
     this.nroCliente = nroCliente;
  }
  public int getNroCliente()
  {
     return nroCliente;
  }
  public void setNombre(String nombre)
  {
     this.nombre = nombre;
  }
  public String getNombre()
  {
     return nombre;
  }
  public void setApellido(String apellido)
  {
     this.apellido = apellido;
  }
  public String getApellido()
  {
     return apellido;
  }
```

```java
public class Cliente
{
  /**
   * Atributo de clase, compartido por todas las instancias
   */
  public static int siguienteNro = 0;
  private int nroCliente;
  private String nombre;
  private String apellido;

  /**
   * Cada vez que se crea un cliente nuevo,
   * se incrementa en 1 el atributo de clase.
   * Luego se asigna ese valor al atributo nroCliente.
   */
  public Cliente()
  {
    siguienteNro++;
    nroCliente = siguienteNro++;
  }
  public static int getSiguienteNro()
  {
    return siguienteNro;
  }
  @Override
  public String toString()
  {
    return ("Cliente nro: " + nroCliente + " Apellido: " + apellido + "Nombre: " + nombre);
  }
```

Ejemplo: necesitamos que cada cliente tenga un número de cliente único

Declaramos una variable de clase llamada **siguienteNro** que es accesible por todos y que además es compartida por todas las instancias.

Observemos como la declaramos: **public static int siguienteNro;**

siguienteNro es compartida por todas las instancias de Cliente

Cada vez que creamos un cliente nuevo, incrementamos en 1 la variable compartida.

También podemos escribir **Cliente.siguienteNro++;**

Observemos, que para acceder a una variable de clase usamos el operador **"."**, al igual que lo hacemos con las variables de instancia, con la diferencia que no lo hacemos sobre una instancia, sino sobre el **nombre de la clase**.

Qué sucedería si escribiese this.siguienteNro++ ?

```
public static void main(String[] args)
{
  System.out.println("Valor de siguienteNro:"+Cliente.getSiguienteNro());
  Cliente c1 = new Cliente();
  System.out.println("Valor de siguienteNro:"+Cliente.getSiguienteNro());
  c1.setApellido("GOMEZ");
  c1.setNombre("Federico");
  System.out.println(c1);
  System.out.println("Valor de siguienteNro en c1:"+c1.getSiguienteNro());
  Cliente c2 = new Cliente();
  c2.setApellido("Pereira");
  c2.setNombre("Martin");
  System.out.println(c2);
  System.out.println("Valor de siguienteNro en c1:"+c1.getSiguienteNro());
  System.out.println("Valor de siguienteNro en c2:"+c2.getSiguienteNro());
  System.out.println("Valor de siguienteNro:"+Cliente.getSiguienteNro());

}
```

```
Valor de siguienteNro:0
Valor de siguienteNro:2
Cliente nro: 1 Apellido: GOMEZNombre: Federico
Valor de siguienteNro en c1:2
Cliente nro: 3 Apellido: PereiraNombre: Martin
Valor de siguienteNro en c1:4
Valor de siguienteNro en c2:4
Valor de siguienteNro:4
Process exited with exit code 0.
```

¿Puedo acceder directamente a **siguienteNro** o estoy obligado a hacerlo a través de **getSiguienteNro()**? ¿Por qué?

Como **siguienteNro** es una variable pública, es lo mismo hacer **Cliente.siguienteNro** que **Cliente.getSiguienteNro()**

Observemos que cada instancia de **Cliente** tiene sus propios valores para las variables de instancia:

c1 tiene a **Gomez** como **apellido** mientras que **c2** tiene a **Pereira** como apellido. Lo mismo sucede con el nombre y el número de cliente de cada uno de ellos.

Ahora, observemos el valor de la variable **siguienteNro** cuando la accedemos a través de las instancias y a través del nombre de la clase:

El valor es el mismo: 2. Estamos consultado a una **variable compartida, NO es una variable individual de cada instancia.**

¿Es lo mismo hacer **c2.getSiguienteNro()** o **c1.getSiguienteNro()** qué **Cliente.siguienteNro**? ¿Por qué?

SI!!! porque tanto c1 como c2 en el método **getSiguienteNro()** acceden a la variable de clase **compartida siguienteNro**.

```java
package modelo;

public class Cliente {

    private static int siguienteNro = 0;
    private int nroCte;
    private String nombre;
    private String apellido;

    public Cliente(){
        siguienteNro++;
        nroCte=siguienteNro;
    }
    // metodos de clase
    public static int getSiguitenNro(){
    return siguienteNro;
    }
    // metodos de instancia
    public String toString(){
        return ("Cliente nro: "+nroCte+" Apellido: "+
                apellido+" Nombre: " +nombre);
    }
```

El método **getSiguienteNro()** ¿devuelve un valor particular de una variable para cada objeto Cliente?

NO!!!

Siempre devuelve lo mismo, independientemente del objeto Cliente que recibe el mensaje, ya que retorna el valor de la variable de clase **siguienteNro** que es el mismo para todos los objetos **Cliente**.

getSiguienteNro() es un método de clase pues accede a una variable de clase.

Lo declaramos: **public static int getSiguienteNro()**

¿Para ejecutar **getSiguienteNro()** necesito crear un objeto **Cliente**?

NO necesito tener creado un objeto **Cliente**, puedo acceder a la variable **siguienteNro** usando el nombre de la clase:

Cliente.getSiguienteNro();

Ahora en **TestCliente** invocamos al método al método de clase **getSiguienteNro()**:

```
package modelo;

public class TestCliente {

    public static void main(String[] args) {
        System.out.println("Valor de siguienteNro: "
                +Cliente.getSiguitenNro());
        Cliente c1 = new Cliente();
        System.out.println("Valor de siguienteNro: "
                +Cliente.getSiguitenNro());
        c1.setApellido("Gomez ");
        c1.setNombre("Federico");
        System.out.println(c1);
        Cliente c2 = new Cliente();
        c2.setApellido("Pereira ");
        c2.setNombre("Martin");
        System.out.println(c2);
        System.out.println("Valor de siguienteNro: "
                +Cliente.getSiguitenNro());
    }

}
```

Imprime lo siguiente:

```
Valor de siguienteNro: 0
Valor de siguienteNro: 1
Cliente nro: 1 Apellido: Gomez  Nombre: Federico
Cliente nro: 2 Apellido: Pereira  Nombre: Martin
Valor de siguienteNro: 2
```

Podemos acceder al valor de la variable de clase **siguienteNro** invocando al método **getSiguienteNro()** sin haber creado una instancia de Cliente.

- En los métodos de clase NO está disponible el objeto **this**, pues no tienen ningún objeto asociado, y tampoco están disponibles las variables de instancia.
- Un **método de clase** solo tiene acceso a sus variables locales, parámetros y variables de clase. NO puede acceder a variables ni métodos de instancia.
- Un **método de instancia** si puede acceder a métodos y variables de clase.

Constantes

Las constantes en JAVA se definen usando la palabra clave **final**. Una vez que un atributo declarado **final** es inicializado, no es posible cambiar su valor.
Cuando estamos definiendo atributos final tenemos dos formas de inicializarlas:

En el constructor

```
package modelo;

public class Producto {
    private static int siguienteCodigo=0;
    private int codigo;
    private String marca="Sin marca";
    private String nombre="Sin nombre";
    private double precio=0.0;
    private final int iva;

    public Producto(String marca,String nombre,double precio){
        this.marca=marca;
        this.precio=precio;
        this.nombre=nombre;
        this.iva=21;
        siguienteCodigo++;
        this.codigo=siguienteCodigo;
    }
```

En la declaración

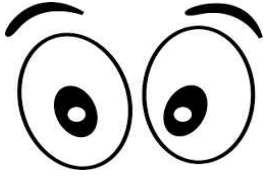
```
package modelo;

public class Producto {
    private static int siguienteCodigo=0;
    private int codigo;
    private String marca="Sin marca";
    private String nombre="Sin nombre";
    private double precio=0.0;
    private final int iva = 21;

    public Producto(String marca,String nombre,double precio){
        this.marca=marca;
        this.precio=precio;
        this.nombre=nombre;
        siguienteCodigo++;
        this.codigo=siguienteCodigo;
    }
```

Constantes

Qué sucede si modificamos el atributo **final** iva?

```
package modelo;

public class Producto {
    private static int siguienteCodigo=0;
    private int codigo;
    private String marca="Sin marca";
    private String nombre="Sin nombre";
    private double precio=0.0;
    private final int iva = 21;

    public Producto(String marca,String nombre,double precio){
    public int getIva() {
        return iva;
    }
    public void setIva(int iva) {
        this.iva=iva;
    }
```

Si queremos modificar el valor del atributo **final iva**, se produce un error de compilación, ya que las constantes una vez inicializadas NO pueden cambiar su valor.



¿Qué sucede si el atributo final en lugar de ser de tipo primitivo es un objeto?

Constantes objetos (no primitivos)

No puedo cambiar el contenido de la variable de tipo final !!
Se producirá un error en compilación si lo intentamos.

```
package modelo;

public class Producto {
    private static int siguienteCodigo=0;
    private int codigo;
    private String marca="Sin marca";
    private String nombre="Sin nombre";
    public  final Empresa productor;
    private double precio=0.0;
    private final int iva = 21;

    public Producto(String marca,String nombre,double precio

    public void setProductor(Empresa productor) {
        this.productor = productor;
    }
```

Observemos que si intentamos **cambiar** el valor del atributo **productor** se produce un **error de compilación**, ya que una vez inicializado NO puede ser modificada la referencia al objeto.

Constantes – cambiar el estado del objeto referido

¿Puedo modificar los datos del objeto productor?
¿puedo cambiarle el valor del atributo nombre?

```
package modelo;

public class TestProducto {

    public static void main(String[] args) {
        Empresa e1 = new Empresa();
        e1.setNombre("Rexona");
        e1.setDireccion("50 y 115");
        Empresa e2 = new Empresa();
        e1.setNombre("Lux");
        e1.setDireccion("1 y 47");
        Producto p1 = new Producto("Lux","jabon tocador",2.2,e1);
        System.out.println(p1);
        p1.productor=e2;
        p1.productor.nombre="Limol";
        System.out.println(p1);
        }

}
```

```
Lux jabon tocador 2.2 Empresa: Lux 1 y 47
Lux jabon tocador 2.2 Empresa: Limol 1 y 47
```

Al intentar cambiar el valor de la referencia al objeto **productor** se produce un error en compilación

Podemos modificar los valores de los datos del objeto productor.

Si!!!. Los valores de los atributos del objeto productor pueden cambiarse.

Colector de basura y “finalize”

Garbage Collection – Destrucción de objetos

- El manejo manual de memoria es una tarea compleja y propensa a errores.
- Cuando no hay ninguna referencia hacia un objeto, este se marca para ser recogido por el recolector
- Los lenguajes OO usualmente incluyen un mecanismo llamado Garbage Collection.
  - Detecta los objetos que ya nadie conoce y libera la memoria que ocupaban.
  - Es automático.
  - Es seguro.
  - Es transparente (corre en background).
- Algunas desventajas
  - No puede utilizarse en sistemas de tiempo real.
  - Los primeros garbage collectors tenían un overhead importante.

Finalize

El método finalize se llama cuando va a ser liberada la memoria (que ocupa el objeto) por el recolector de basura (garbage collector).

Normalmente, no es necesario redefinir este método en las clases, solamente en contados casos especiales. Por ejemplo, si una clase mantiene un recurso que no es de Java como un descriptor de archivo o un tipo de letra del sistema de ventanas, o cuando se han abierto varios archivos durante la vida de un objeto, y se desea que los archivos estén cerrados cuando dicho objeto desaparece, entonces sería acertado el utilizar la finalización para asegurar que los recursos se liberan. Es similar a los destructores de C++.

Finalize

```
class CualquierClase
{  ...
   protected void finalize() trows Throwable
   {
      super.finalize();
      //código que libera recursos externos
   }
}
```

La primera sentencia que contenga la redefinición de finalize ha de ser una llamada a la función del mismo nombre de la clase base, y a continuación le añadimos cierta funcionalidad, habitualmente, la liberación de recursos, cerrar un archivo, etc.

Entonces: **`void objeto.finalize()`** es llamado por el recolector de basura antes de eliminar el objeto. Esta implementación no hace nada, debe ser el programador el que sobreescriba este método en caso de que quiera realizar algo en especial antes de eliminar el objeto de la memoria.

Se profundizará más adelante

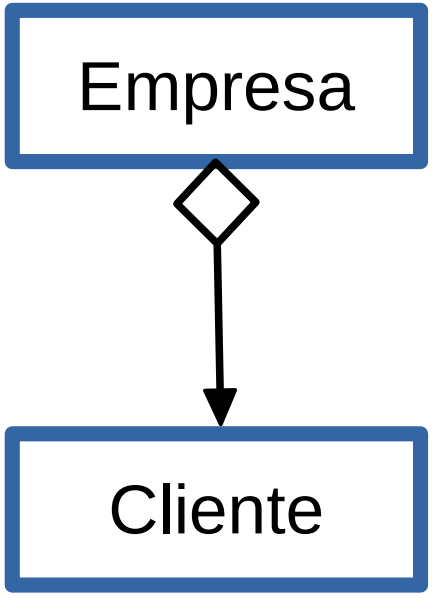
La relación de asociación: agregación y composición

Relación de Agregación

La agregación es un tipo de asociación que indica que una clase es parte de otra clase (composición débil).

Los componentes pueden ser compartidos por varios compuestos (de la misma asociación de agregación o de varias asociaciones de agregación distintas). La destrucción del compuesto no conlleva la destrucción de los componentes. Habitualmente se da con mayor frecuencia que la composición.

La agregación se representa en UML mediante un diamante de color blanco colocado en el extremo en el que está la clase que representa el “todo”.



- Tenemos una clase Empresa.
- Tenemos una clase Cliente.
- Una empresa agrupa a varios clientes.

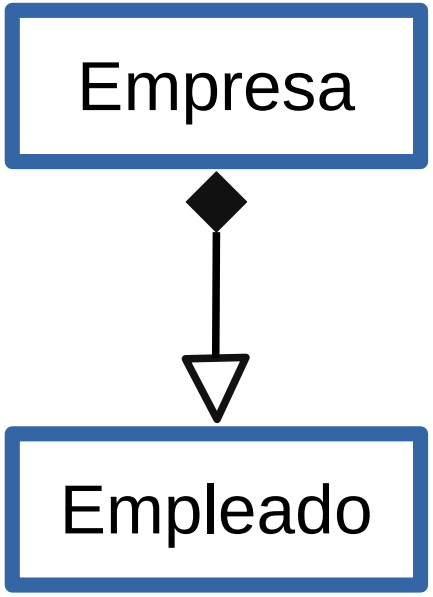
La relación de asociación: agregación y composición

Relación de Composición

Composición es una forma fuerte de asociación donde la vida de la clase contenida debe coincidir con la vida de la clase contenedor. Los componentes constituyen una parte del objeto compuesto.

De esta forma, los componentes no pueden ser compartidos por varios objetos compuestos. La supresión del objeto compuesto conlleva la supresión de los componentes.

El símbolo de composición es un diamante de color negro colocado en el extremo en el que está la clase que representa el "todo" (Compuesto).

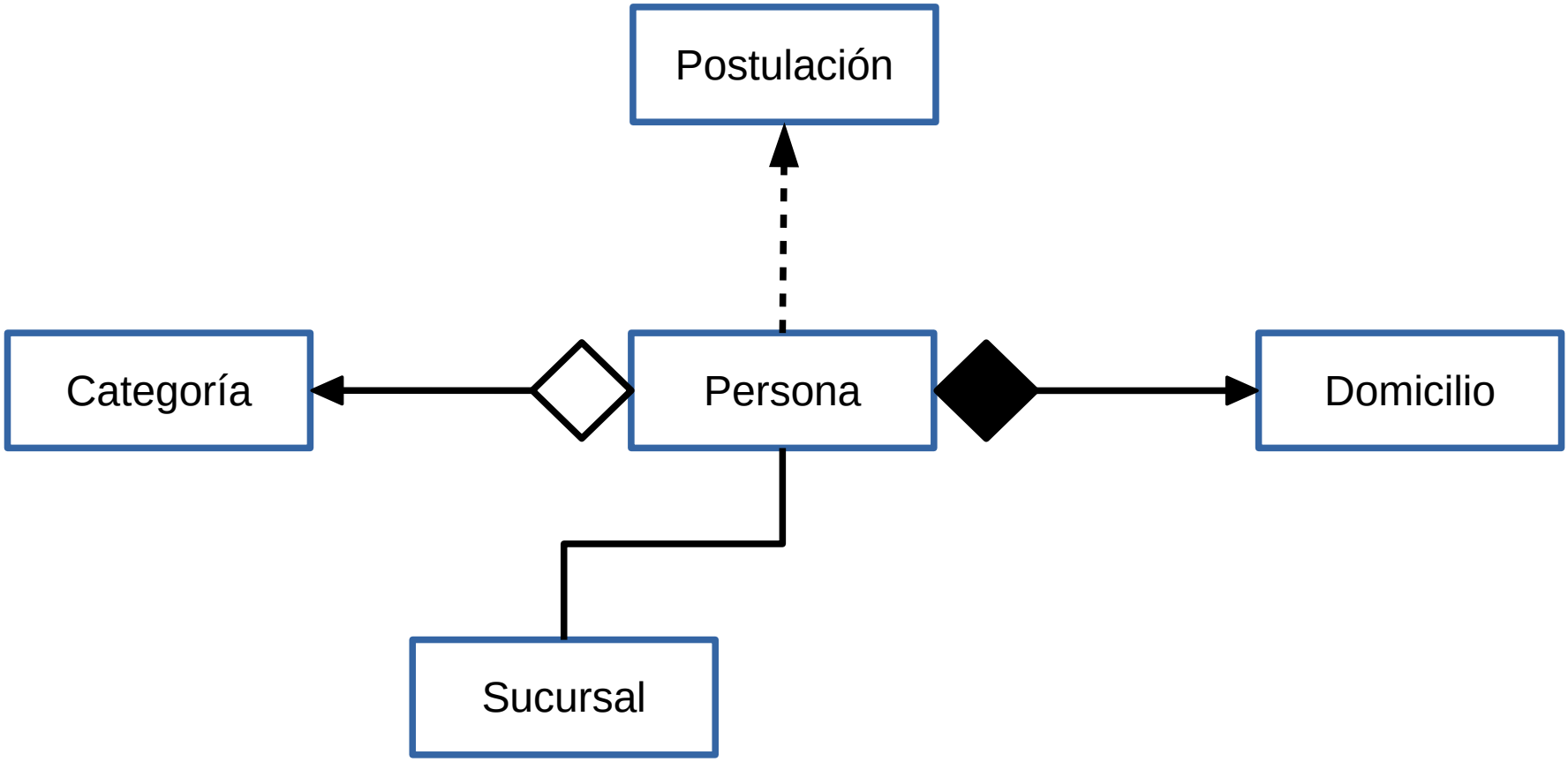


- Tenemos una clase Empresa.
- Un objeto Empresa está a su vez compuesto por uno o varios objetos del tipo empleado.
- El tiempo de vida de los objetos Empleado depende del tiempo de vida de Empresa, ya que si no existe una Empresa no pueden existir sus empleados.

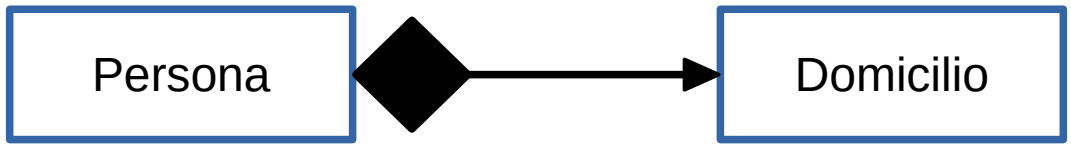
Diferencias entre agregación y composición

	AGREGACIÓN	COMPOSICIÓN
Varias asociaciones comparten los componentes	Sí	No
Destrucción de los componentes al destruir el compuesto	No	Sí
Cardinalidad a nivel de compuesto	Cualquiera	0..1 ó 1
Representación	Rombo transparente	Rombo negro

Asociación, Agregación,Composición y Dependencia



Relación de Composición

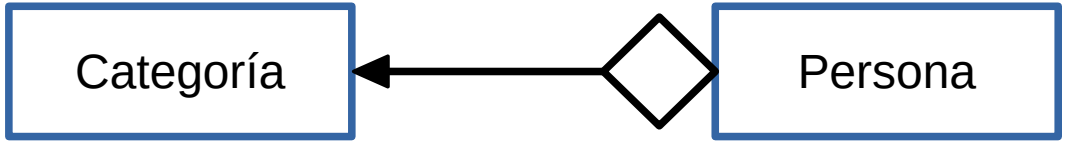


La clase Persona tiene una relación de composición con la clase Domicilio.

Conceptualmente esto significa que los domicilios son una parte inseparable de la persona, por lo que si no existiera una persona entonces el domicilio de la misma debería desaparecer.

Es una relación de composición fuerte.

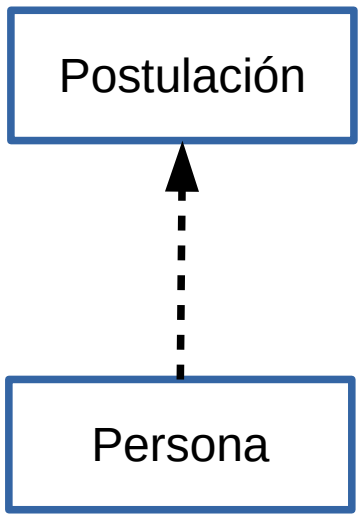
Relación de Agregación



La clase Persona tiene una relación de Agregación con Categoría.

Conceptualmente esto significa que las categorías existen independientemente de la persona que la tenga asignada. En el modelo conceptual esto se corresponde con un enunciado como "una persona puede tener una categoría pero una categoría puede estar presente en muchas personas".

Relación de Dependencia

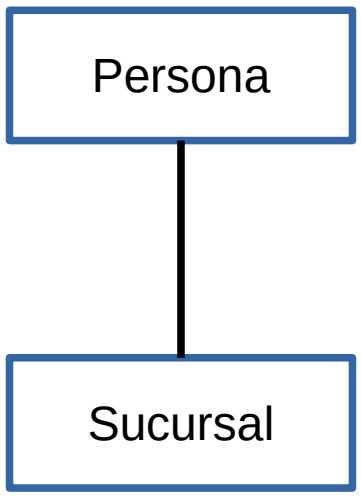


La clase Persona tiene una relación de Dependencia con Postulación.

Conceptualmente esto significa que la Postulación es un objeto que la Persona utiliza para algún fin, dentro de alguna operación que ella realice (por ejemplo Postularse a un cargo). Pero una Persona no tiene en su interior una Postulación, sino que solo lo utiliza para realizar ciertas operaciones.

Por ejemplo, un método de Persona que cree y retorne un objeto de tipo Postulación.

Relación de Asociación



La clase Persona tiene una relación de Asociación con Sucursal.

Conceptualmente la asociación en un diagrama de clases implica transitividad y bidirección de clases. Por ejemplo una persona tiene como atributo interno a una Sucursal, pero (y aquí está la diferencia) una Sucursal también tiene un atributo de tipo Persona; la cardinalidad de la asociación indicará si Sucursal tiene una o muchas instancias de Persona, con lo cual en realidad el atributo de Sucursal podría ser una Lista de Personas.

Otra característica fundamental es que la vida de las instancias de ambas clases no dependen una de la otra.

Bloques de Inicialización

Se utilizan para realizar inicializaciones más complejas. Es un bloque de sentencias que aparece dentro de la declaración de la clase y fuera de la declaración de cualquier miembro o constructor, y que inicializa los campos del objeto.

Se utilizan en inicializaciones no triviales, cuando no se necesitan parámetros de construcción y hay una razón para no proporcionar un constructor sin argumentos.

Bloques de Inicialización

```
class Cuerpo
{
    public long numID;
    public String nombre = "<sin nombre>";
    public Cuerpo orbita = null;
    private static long sigID = 0;

    {
        numID = sigID++;
    }
}
```

Bloques de Inicialización Estático

Los campos estáticos de una clase pueden tener inicializadores. Pero además se puede realizar una inicialización estática más compleja utilizando un *bloque de inicialización estática.*

Son similares a los anteriores, pero sólo pueden referirse a miembros estáticos de la clase.

Por ejemplo, creación de un array estático.

Bloques de Inicialización Estático

```
class Primos
{
    static int[] primosConocidos = new int[4];

    static
    {
        primosConocidos[0] = 2;
        for (int i = 1; i < primosConocidos.length; i++)
        {
            primosConocidos[i] = sigPrimo();
        }
    }
    // declaración de sigPrimo
```