Objeto, Clase, UML y Java: un enfoque práctico

Objeto

En el paradigma de objetos, los objetos son los elementos primarios utilizados para construir programas. Dichos objetos son abstracciones de alguna entidad del mundo real (tangible o no), circunscripta al dominio del problema a resolver (espacio del problema).

Al hablar de abstracción se hace referencia al hecho de aislar los aspectos relevantes, en un determinado contexto, descartando los detalles que no son significativos en el dominio del problema.

Para pasar del problema al espacio de la solución (resolver el problema en cuestión) en el paradigma OO solo se cuenta con objetos, que se comportan de una determinada manera, definida por sus métodos, e interactúan entre ellos enviándose mensajes. Este es el concepto básico de la POO.

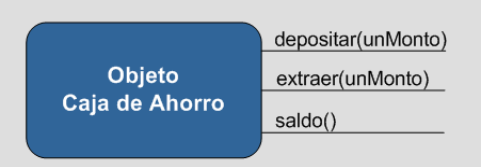
Un objeto es una entidad que queda caracterizada por:

- **su estado interno**: representado por un conjunto de variables de instancia (v.i.). Al crear un objeto, sus atributos toman valores particulares, representando las características intrínsecas de una entidad particular. Estas variables se utilizan para representar aquellas características propias del objeto (como puede ser el saldo de una cuenta bancaria) y a los objetos con los que deberá colaborar a lo largo de su vida (por ejemplo un titular). En este último caso se da una relación de conocimiento, tema que será tratado más ampliamente.
- **su comportamiento**: el cual está dado por los mensajes a los que puede responder. El comportamiento del objeto puede verse como los servicios que presta, su utilidad. Esto determina para qué existe. Por ejemplo, la administración de una cuenta bancaria, representada por los métodos extraer, depositar, consultar saldo, etc.
- **una identidad**: ni un objeto es idéntico a otro, incluso aunque tengan los mismos valores en sus atributos. Por ejemplo, dos titulares tendrán distinta identidad, a pesar de llamarse ambos "Juan".

Otra de las propiedades importantes de los objetos es el encapsulamiento, que asegura que un objeto no puede invadir el espacio privado (estado interno) de otro objeto. Es el propio objeto el único que puede manipular su estado interno, es decir, ver o modificar los valores de sus atributos.

Comportamiento: ¿Qué hace?

El comportamiento indica qué sabe hacer el objeto, es decir cuáles son sus responsabilidades. El comportamiento se especifica a través del conjunto de mensajes que el objeto sabe responder. Un objeto se define en términos de su comportamiento, por lo que al modelar con objetos se debe establecer **qué sabe hacer** cada objeto. El objeto debe verse como una entidad activa, con la capacidad de llevar tareas adelante. Además, se debe tener en cuenta que para llevar a cabo su trabajo, un objeto puede recibir la colaboración de otros objetos.



Al conjunto de mensajes que un objeto puede entender se lo denomina **protocolo**.

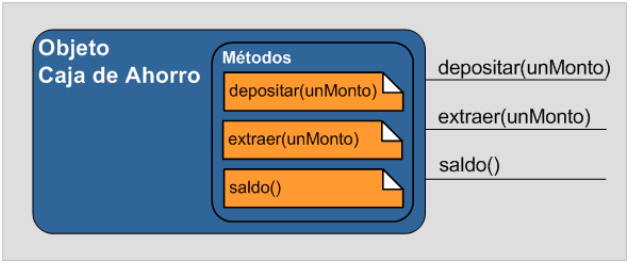
Comportamiento: ¿Cómo lo hace?

La implementación de un objeto indica **cómo hace** el objeto para responder a los mensajes que recibe. Se especifica a través de un conjunto de métodos. Esto significa que en el cuerpo de cada método está el código que realiza la tarea que debe llevar adelante el objeto.

La encapsulación también alcanza a la implementación del objeto. En efecto, el código es privado. Solamente el propio objeto puede accederlo de manera directa, y hacer uso de él.

Implementación: Método vs. Mensaje

Cuando un objeto envía un mensaje a otro, el objeto receptor responde activando el método asociado a ese mensaje. Básicamente un método es la implementación (código) asociada a un mensaje. Una vez definido el protocolo de un objeto (qué es lo que va a hacer, cuáles son sus responsabilidades), es necesario especificar cómo va a llevar a cabo sus responsabilidades y esto se hace por medio de la codificación de los métodos.



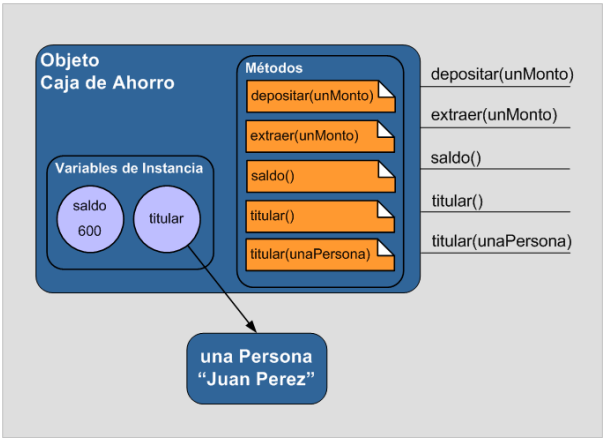
El estado interno

El estado interno está representado por los valores que toma el conjunto de atributos de un objeto, representado por un conjunto de variables de instancia (**v.i.**). Las variables de instancia acompañan al objeto durante todo su ciclo de vida.

Una v.i. puede hacer referencia a una propiedad intrínseca del objeto que representa, o bien **a otro objeto que colabora con el objeto primario** para que éste lleve a cabo sus responsabilidades. A lo largo de la vida del objeto, las propiedades y colaboradores pueden cambiar. Por ejemplo, al depositar $100 en una cuenta bancaria la variable de instancia saldo cambiará su valor. Este estado interno es privado del objeto. Es una de las características fundamentales del paradigma, y resulta crucial para lograr software escalable. Por definición, las v.i. (estructura interna) de un objeto son privadas a éste. El único que puede acceder y manipular sus v.i. es el propio objeto. Si un objeto externo quiere acceder a una v.i. lo tiene que hacerlo por medio del envío de un mensaje. Esto le da la posibilidad al objeto de decidir qué es lo que quiere publicar, o cómo quiere hacerlo.

En la siguiente figura puede observarse un ejemplo concreto: un objeto ***Caja de Ahorro***, cuyo estado interno queda representado por sus v.i.: toma un valor particular en el caso del ***saldo***, e indica una relación de conocimiento con el objeto ***Persona***, que colabora con él a través de la v.i. ***titular***. A su vez puede determinarse el comportamiento (qué hace el objeto), a través del conjunto de métodos (***depositar(), extraer(), saldo(),*** etc), que responde a sendos mensajes.

Atención: No se debe confundir el **atributo** saldo con el **método** saldo().



Mensajes

Para llevar a cabo la tarea de un sistema en POO, los objetos deben colaborar entre sí. Los objetos interactúan enviándose mensajes.

Un mensaje es una petición o solicitud que hace un objeto a otro para que realice una tarea.

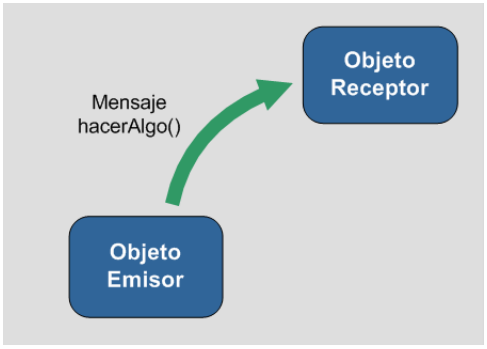
"Llamada a una operación o a un objeto, en la que se incluye el nombre de la operación y una lista de valores de argumentos" [Rumbaugh et al., 1991]

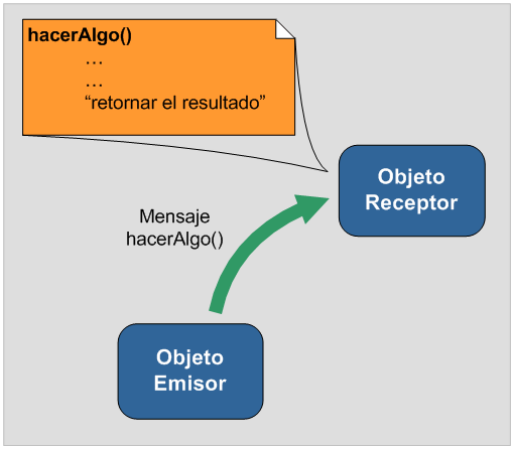
"Operación que un objeto realiza sobre otro" [Booch, 1994]

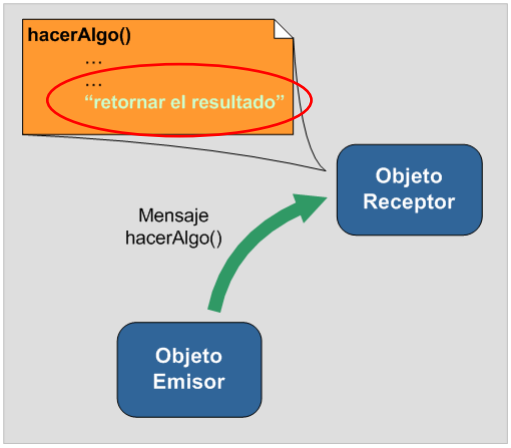
"Una comunicación entre objetos que transmite información con la expectativa de desatar una acción. La recepción de un mensaje es, normalmente, considerado como un evento" [OMG, 2003]

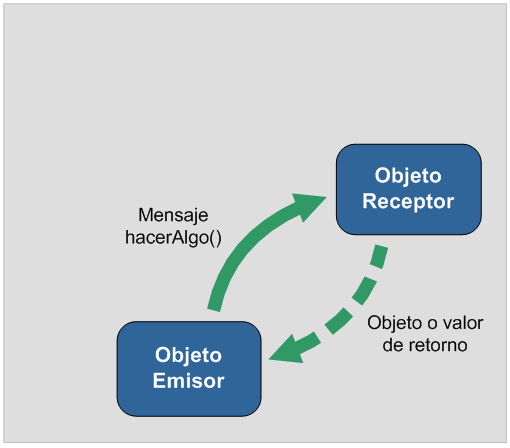
Para poder enviarse mensajes, los objetos deben conocerse. Esto significa que debe haber una "relación" entre ellos. El objeto receptor responde en forma activa, ejecutando el método asociado al mensaje. El método, es decir la implementación (el código), especifica qué hace el objeto receptor al recibir el mensaje.

Cuando un objeto (emisor) le envía un mensaje a otro objeto (receptor), el objeto receptor responde activando el método asociado a este mensaje, siempre que el mensaje recibido forme parte del protocolo del objeto receptor. Caso contrario, pueden darse distintas situaciones, dependiendo del lenguaje: algunos lenguajes pueden evitar esta situación mediante un sistema de tipos fuerte que chequee estas situaciones en tiempo de compilación (java), mientras que otros brindan una mayor libertad en cuanto al envío de mensajes y responden, eventualmente, con un error en tiempo de ejecución (smalltalk).









Formato de mensajes

Típicamente un mensaje tiene un nombre, puede o no tener argumentos, y se invoca sobre un objeto. Los distintos lenguajes de programación orientados a objeto proponen una sintaxis particular para indicar el envío de un mensaje. El formato aceptado habitualmente es el siguiente:

```
objeto_receptor.nombre_de_mensaje(argumentos);
```

Ejemplo:

```
unaCuenta.depositar(100);
```

Los valores pasados como argumento en el mensaje (***parámetros reales***), son reemplazados en los ***parámetros formales*** del método, al momento de ejecutarse.

Métodos

Un método es un conjunto de sentencias que llevan a cabo una acción. Puede tener parámetros de entrada que regularán dicha acción y, puede tener un valor de salida (o valor de retorno). La diferencia con un procedimiento o función usados en programación estructurada es que un método, al estar asociado con un objeto o clase en particular, puede acceder y modificar los datos privados del objeto correspondiente de forma tal que sea consistente con el comportamiento deseado para el mismo. Así, es recomendable entender a un método no como una secuencia de instrucciones sino como la forma en que el objeto es útil (el modo de hacer su trabajo). Por lo tanto, podemos considerar al método la respuesta de un objeto ante el pedido para que realice una tarea.

Cada método tiene:

- una parte que es pública (la **firma** del método), que constituye la interfaz pública, lo que los demás objetos pueden ver. Expone lo que sabe hacer.
- una parte que es privada (el cuerpo del método – el código), que está oculta. El ocultamiento garantiza que no se propaguen los efectos de los cambios en las estructuras de datos encapsuladas dentro del objeto, sobre otras partes del sistema.

En la declaración de un método se establece un nombre (se recomienda que tenga significado en el dominio del problema, dado que propicia la legibilidad), la visibilidad, el tipo de retorno, y los **parámetros formales.** Los parámetros formales actúan como variables locales en el cuerpo del método, y se inicializan al valor que se pasa como argumento en la invocación del método (**parámetros reales**):

```
public void asignarSaldo(double unMonto){
          saldo = unMonto;
}
```

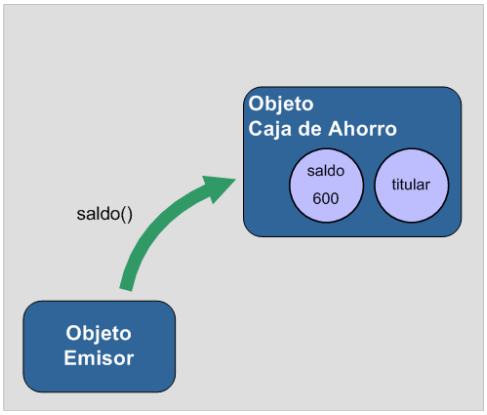
Un método es la contraparte funcional del mensaje. Por cada mensaje que el objeto debe responder, declarado como parte del protocolo del objeto, debe existir un método asociado que exprese la forma de llevar a cabo la tarea solicitada por dicho mensaje, es decir, **cómo se hace**. El *cómo* en un método puede implicar básicamente tres cosas:

1) Modificar el estado del objeto. Esto implica asignar a una v.i. un valor distinto al que tiene. Por ejemplo, en el caso de la cuenta bancaria, se podría tener un método como el siguiente:

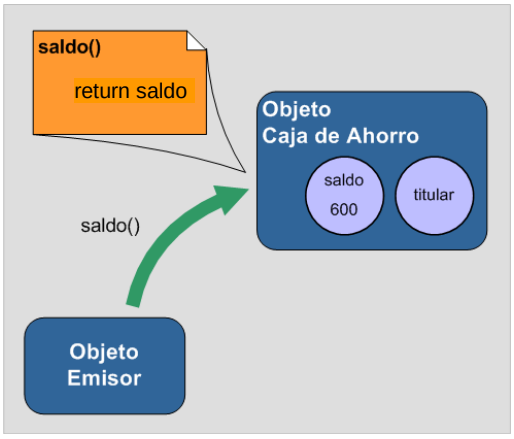
```
public void asignarSaldo(double unMonto){
    saldo = unMonto;
}
```

2) Retornar y terminar. Un método puede retornar un resultado. Por ejemplo, en el caso de la cuenta bancaria es lógico que le podamos pedir el saldo:

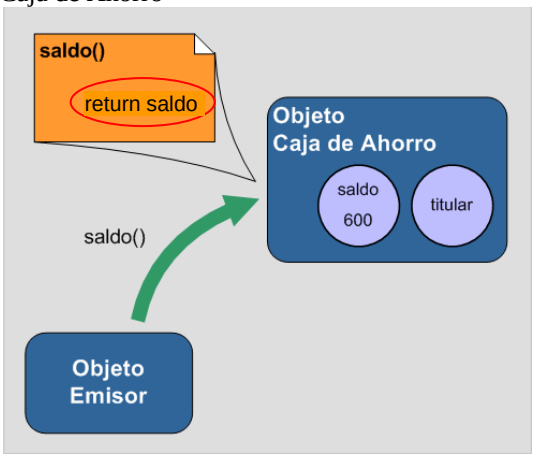
```
public double saldo(){
      return saldo;
}
```



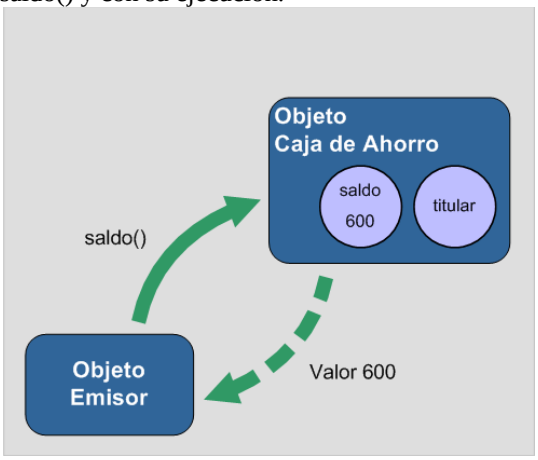
Un objeto emisor le envía el mensaje saldo() a un objeto Caja de Ahorro



El objeto receptor procede con la activación del método saldo() y con su ejecución.



El método simplemente retorna el saldo actual de la caja de ahorro (el número 600)



Ese valor es el resultado que el emisor obtiene en respuesta al envío del mensaje saldo()

3) Colaborar con otros objetos. En la mayoría de los casos, para llevar a cabo una determinada responsabilidad, el objeto necesita la colaboración de otros objetos. Por ejemplo, para alquilar una película es necesario saber si el cliente adeuda alguna película. Por lo tanto, el método *alquilar* en el objeto *videoClub* sería algo de la forma:

```
      public void alquilar(unaPelicula, unCliente){
            if (unCliente.esDeudor()) {
                  System.out.println("No alquilar");
            } else {
                    unaPelicula.alquilarA(unCliente)
                  }
      }
```

El video club básicamente funciona como un coordinador de comportamiento del cliente y de la película.

Especificación de un método

Cada lenguaje de programación propone su propio modo de especificar un método. Particularmente en Java se utiliza el siguiente (básico, incompleto):

```
/** Breve comentario del método
  * @param tipo_dato parametro1, tipo_dato parametro2
  * @return Devuelve una variable de tipo tipo_retorno
  * @throws No dispara ninguna excepción
  */
  tipo_acceso tipo_retorno nombreMensaje(tipo_dato parametro1, tipo_dato parametro2){
                tipo_dato temp1;
                tipo_dato temp2;
                varDeInstancia = parametro1;
                temp1 = parametro2;
                temp2 = temp1.unMensaje(parametro1);
                return temp2;
  }
```

Los métodos se diferencian por: nombre del método y cantidad, tipo y orden de los parámetros. Todo esto constituye la "**firma del método**" (*method signature* en ingles)

Importante: el tipo de valor retornado y el tipo de acceso no forman parte de la firma del método (no son utilizados para distinguir entre métodos)

Ejemplo 1: implementación en java del método saldo() del objeto caja de ahorro.

```
/** Devuelve el saldo actual de la cuenta
  * @param No recibe parámetros
  * @return Devuelve un valor de tipo double
  * @throws No dispara ninguna excepción
  */
  public double saldo(){
           return saldo;
  }
```

Ejemplo 2: implementación en java del método depositar() del objeto caja de ahorro.

```
/** Agrega unMonto al saldo actual de la cuenta
  * @param double unMonto
  * @return No devuelve ningún valor
  * @throws No dispara ninguna excepción
  */
  public void depositar(double unMonto){
           saldo = saldo + unMonto;
  }
```

Observadores y Mutadores (Accessors: getters y setters)

Dado que el estado de los objetos es privado, se debe proveer el mecanismo para que los demás objetos puedan acceder a sus atributos, de modo consistente. Para ello se utiliza lo que se conoce como observadores (getters), que solo muestran sus atributos, sin modificarlos, y los mutadores (setters), que asignan valores o modifican los atributos. Estos no son nombres obligatorios. Solo son utilizados por su significado: get=obtener, set=asignar.

Por ejemplo, si se desea obtener el saldo de una caja de ahorro, en java se escribirá lo siguiente:

```
/** Retorna el saldo actual de la cuenta.
  * @return Un valor de tipo double.
  * @throws No dispara ninguna excepcion.
  */
```

```
  public double getSaldo(){
            return saldo;
  }
```

En el caso de querer asignar un monto particular al saldo, en java se procederá de la siguiente manera:

```
/** Asigna unMonto al saldo actual de la cuenta.
  * @param double unMonto.
  * @return No devuelve ningún valor.
  * @throws No dispara ninguna excepcion.
  */
  private void setSaldo(double unMonto){
            saldo = unMonto;
  }
```

En los lenguajes OO el encapsulamiento garantiza que los usuarios de un objeto sólo pueden modificar su estado e interactuar con él a través de sus métodos. La gran ventaja del encapsulamiento consiste en que si un módulo u objeto cambia internamente sin modificar su interfaz, el cambio no desencadenará ninguna otra modificación en el sistema. Esta característica puede ser llevada a un nivel superior, o **doble encapsulamiento**, lo cual significa que el propio objeto no accede a sus atributos directamente sino a través de sus métodos. Para la implementación de este concepto, por ejemplo en los métodos constructores, se utilizan los métodos set para las asignaciones:

```
    CajaAhorro(int p_codigo, double p_saldo) {
        setCodigo(p_codigo);
        setSaldo(p_saldo);
    }
```

En el caso de un método de actualización, como por ejemplo una extracción de una caja de ahorro, se implementa mediante los *accessors*, como puede observarse en el siguiente ejemplo:

```
  public void extraer(double p_importe){
        setSaldo(getSaldo()- p_importe);
    }
```

En conclusión, la implementación del mecanismo de **doble encapsulamiento** mediante los *accessors* permite acceder a los atributos de modo consistente y evita la modificación en cadena del sistema.

Objeto y Clase

Una clase representa otro nivel de abstracción. Según lo expresado, un objeto es una abstracción de una entidad del modelo del dominio. Dado que generalmente en el dominio del problema aparece un conjunto de objetos que se comportan en forma similar, surge la necesidad de encontrar una forma de definir el comportamiento de todos los objetos en forma general.

Por ejemplo, si el objeto que se está analizando es una caja de ahorro, es evidente que en un banco habrá una gran cantidad de objetos de este tipo, uno por cada caja de ahorro existente en el banco. Se puede afirmar que todos estos objetos tendrán el mismo comportamiento, y si hay algún cambio (por ejemplo, el interés devuelto), el mismo se reflejará en todos los objetos de este tipo.

Esto lleva a pensar que sería útil algún mecanismo que permita agrupar el comportamiento común a un conjunto de objetos, de manera que pueda ser reutilizado sin tener que ser especificado individualmente. Por otra parte, el estado interno de cada caja de ahorro será diferente (por ejemplo, tendrán distinto saldo), pero todas compartirán la misma estructura interna (todas conocen su saldo, titular, etc.). Desde el punto de vista de la reutilización, es así como surge la noción de clase: como una herramienta que permite factorizar y reutilizar estructura y comportamiento en común. La clasificación es un mecanismo de abstracción, concepto que se desarrollará más ampliamente.

Por este motivo, aparece la noción de ***clase*** "Caja de Ahorro", la cual se utiliza para describir el comportamiento de todas y cada una de sus ocurrencias (***instancias***). Cada objeto es un elemento único de la clase en la que se basa. Una clase es como un molde de un ***tipo*** específico de objeto, por lo tanto un objeto es lo que se crea a partir del molde. La ventaja que representa una clase está en que permite describir en un sólo lugar el comportamiento genérico de un conjunto de objetos. Una vez definido este comportamiento, es posible crear objetos que lo reutilicen y se comporten de la manera allí descripta.

Como definición de clase se puede tomar la siguiente: "*Una clase es una descripción abstracta de un conjunto de objetos".*

Una clase especifica qué forma tendrán sus instancias (sus variables de instancia) y cómo responderán a los mensajes que se le envíen. De esta forma, una clase puede ser vista como una descripción de sus instancias y como un repositorio de comportamiento. Cuando se le envía un mensaje a un objeto, lo que hace el intérprete es buscar **en la**

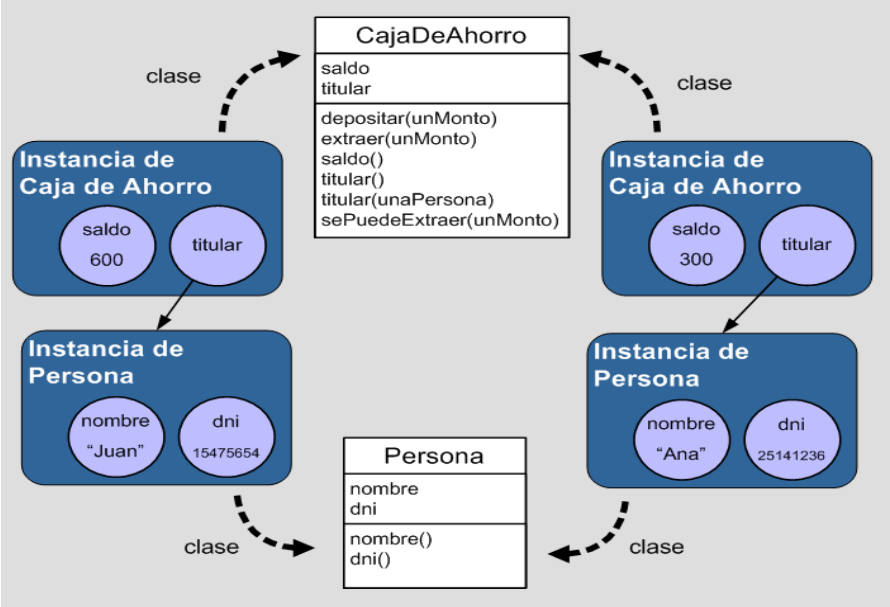
clase del objeto receptor un método con el cual responder al envío del mensaje. De ésta forma, todos los objetos que comparten una misma clase responden de la misma forma al envío de un mensaje.

En el caso de la cuenta bancaria, dado que en un banco habrá miles de cuentas, lo que se hace es modelar la noción de cuenta bancaria en una clase. Esta clase definirá qué variables tendrán sus instancias, así como los mensajes a los que puede responder. Por ejemplo, la clase se puede llamar CuentaBancaria, y especificar que sus instancias (cada caja de ahorro particular) tiene dos variables (saldo y titular) y que puede responder a los mensajes saldo(), titular(), depositar(unMonto) y extraer(unMonto).

Dado que las clases son las encargadas de describir a sus instancias, ante la necesidad de crear una nueva instancia, lo lógico es pedírselo a la clase. Por ese motivo, la clase tiene la responsabilidad de crear sus instancias. Por ejemplo, para crear una nueva cuenta bancaria, se debe pedir a la clase CuentaBancaria que retorne una nueva instancia. Para ello, las clases cuentan con métodos especiales, llamados "constructores" o "inicializadores" (según sea el lenguaje). Cuando se desea crear un objeto (instanciar una clase), se invocan estos métodos, que definen un proceso de inicialización que prepara el objeto para ser usado (le asigna un espacio de memoria).

Es frecuente la confusión entre clase e instancias. Es importante remarcar que una clase es un molde de estructura y comportamiento, así como la encargada de crear a sus instancias. Una vez que se crea una instancia, esta tiene sus **propias** variables (v.i.). Por ejemplo, al crear dos cuentas bancarias, cada una tendrá **su** saldo y **su** titular. Lo único que comparten es que ambas son instancia de la misma clase, lo que implica que al enviarles un mensaje, el método que se activará se buscará en la clase CuentaBancaria, que es común a las dos.

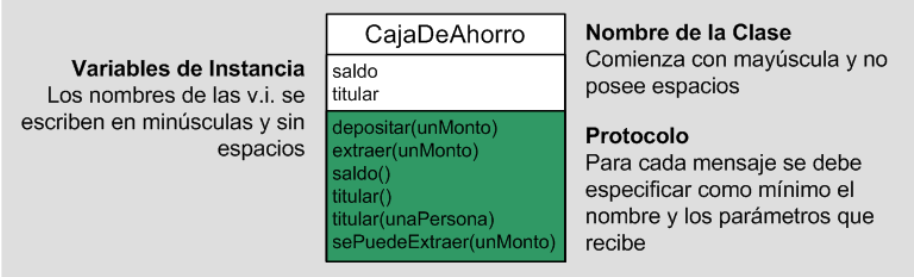
En la figura puede observarse una clase CajaDeAhorro, que define el comportamiento y la estructura común a todas sus instancias. Una CajaDeAhorro sabrá extraer(unMonto) de su saldo, depositar(unMonto) y devolver su saldo actual. Además deberá colaborar con un objeto que cumplirá el rol de titular de la cuenta. Se observan dos instancias de CajaDeAhorro, cada una con su propio estado interno. Las dos instancias de la clase CajaDeAhorro comparten el mismo comportamiento, definido en la propia clase. Esto evita la repetición de los métodos en cada objeto.



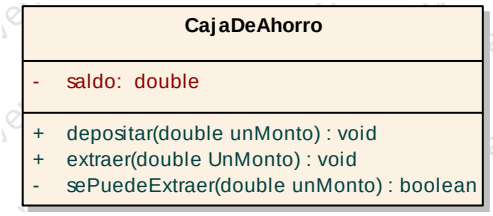
Clase y UML

Las clases se especifican por medio de un nombre, la estructura interna que tendrán sus instancias y los mensajes y métodos asociados que definen su comportamiento. El nombre debe comenzar con mayúscula y no contener espacios. Las v.i. representan el estado o estructura interna de las instancias de la clase. Por último, se detalla el conjunto de mensajes que entenderá una instancia de dicha clase, también llamado ***protocolo***.

La representación gráfica de una clase, utilizando una versión simplificada del lenguaje UML es:



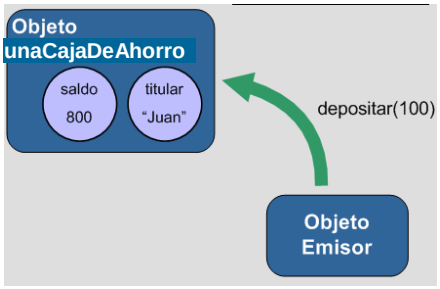
La representación de la clase puede variar según la herramienta utilizada. Sin embargo, generalmente tiene la forma de la figura adjunta, donde puede observarse el tipo de dato, de retorno de los métodos, y un signo que indica el tipo de acceso: público (+), privado (-), protegido (#)



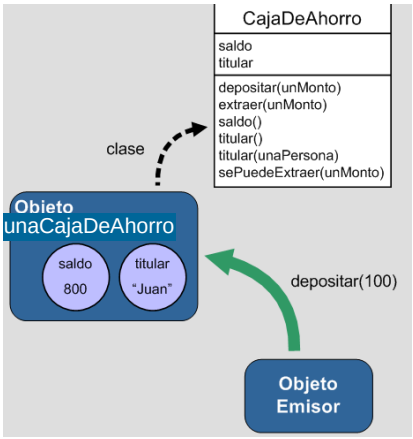
Envío de mensajes considerando el concepto de Clases

Al hablar del concepto de objeto, se observó que, ante el envío de un mensaje, un objeto busca entre sus métodos aquél que corresponde al mensaje en cuestión, para proceder con la activación del mismo.

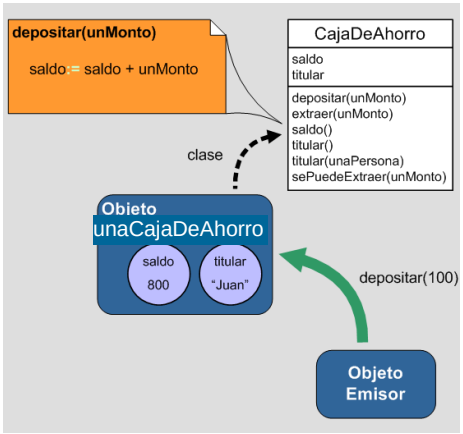
Al aparecer el concepto de clases como repositorio de comportamiento, la búsqueda del método asociado a un mensaje ya no será plena responsabilidad de las instancias, sino que este trabajo será delegado a la clase a partir de la cual fueron creados. En el ejemplo se puede ver cómo el objeto unaCajaDeAhorro delega la búsqueda del método a su clase (CajaDeAhorro) ante la recepción del mensaje *depositar(unMonto)*. La clase buscará entre su colección de métodos aquél que se corresponda con el mensaje *depositar(unMonto)* y se procederá a **ejecutar el método en el contexto del objeto receptor** del mensaje (unaCajaDeAhorro).



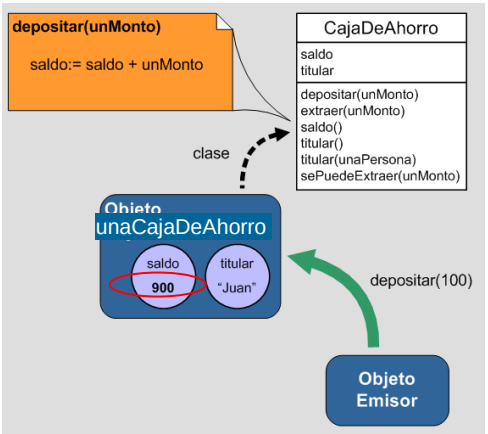
Un objeto emisor envía el mensaje depositar(100) al objeto unaCajaDeAhorro



El objeto unaCajaDeAhorro delega la búsqueda del método a la clase



La clase busca el método que se corresponde con el mensaje recibido y lo ejecuta



El objeto unaCajaDeAhorro cambia de estado

Atributos y Mensajes de Clase

Los mensajes que se le envían a una clase se denominan mensajes de clase y sólo pueden ser enviados a la clase, no a sus instancias.

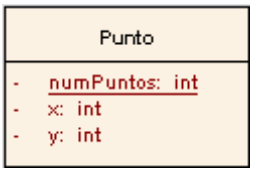
En lenguajes como Java o C++ existe la noción de métodos **static**, que indican que son métodos de clase. Atributos y mensajes estáticos se representan en UML con el nombre subrayado.

Atributos estáticos

Un atributo estático es una variable miembro que no se asocia en particular a un objeto (instancia) de una clase, sino que se asocia a la clase misma; no hay una copia del dato para cada objeto sino una **única** copia que es compartida por todos los objetos de la clase.

Por ejemplo, en java:

```
public class Punto {
    private int x;
    private int y;
    static private int numPuntos;

    Punto ( int a , int b ) {
       x = a ;
       y = b;
       numPuntos = numPuntos + 1;
    }
}
```



En el ejemplo, *numPuntos* es un contador que se incrementa cada vez que se crea una instancia de la clase Punto. Para declararlo, se coloca el modificador **static** delante del tipo.

El acceso a las variables estáticas desde fuera de la clase donde se definen, se puede realizar a través del nombre de la clase (además de hacerlo desde la referencia a un objeto, como es lo habitual). Basado en el ejemplo anterior, para saber la cantidad de puntos creados, puede escribirse:

```
int canti = Punto.numPuntos;
```

NOTA: en este ejemplo simplificado se observa un acceso directo al atributo, que no es correcto ya que el atributo es privado. El siguiente ejemplo incluye un método de acceso.

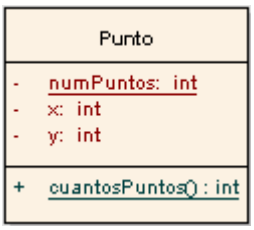
Las variables estáticas de una clase existen, se inicializan y pueden usarse **antes** de que se cree ningún objeto de la clase. Esto no es así con los atributos de un objeto. No pueden usarse antes de crear una instancia.

Métodos estáticos

Para los métodos, el concepto es el mismo que para los atributos: los métodos estáticos se asocian a una clase, y permiten acceder a los atributos estáticos.

Por ejemplo, en java:

```
public class Punto {
    private int x;
    private int y;
    static private int numPuntos;

   Punto(int a, int b) {
        x = a ;
        y = b;
        numPuntos = numPuntos + 1;
    }

    public static int cuantosPuntos() {
        return numPuntos;
    }
}
```



El acceso a los métodos estáticos se hace igual que a los atributos estáticos, es decir, usando el nombre de la clase, (además de hacerlo desde la referencia a un objeto, como es lo habitual):

```
int totalPuntos = Punto.cuantosPuntos();
```

Dado que los métodos estáticos tienen sentido a nivel de clase y no a nivel de objeto (instancia), **los métodos estáticos no pueden acceder a variables de instancia ni a métodos que no sean estáticos**.

Nota: ver ejemplo desarrollado y explicado en **Material de Clase No Presencial**, provisto por la asignatura.

Formas de conocimiento entre objetos

Es fundamental tener presente que, para que un objeto le envíe un mensaje a otro, debe conocerlo, o sea, nombrarlo. Se dice que existe una ligadura (binding) entre un objeto y su nombre.

Ej: Persona titular = unaPersona;

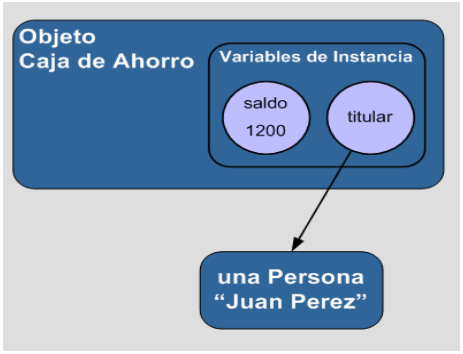
En este ejemplo, **titular** es el nombre del objeto al que hace referencia.

También se dice que hay una "relación" entre los objetos, que puede ser de distintos tipos. A la hora de programar existen diversas formas de conocimiento:

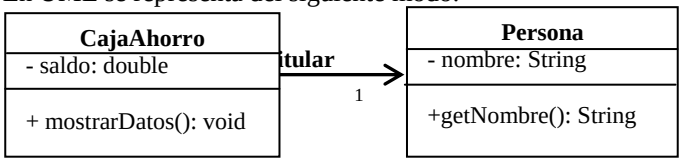
1. Variables de instancia
2. Parámetros
3. Variables temporales
4. Seudo-variables (son una forma especial de conocimiento)

1. Variables de instancia

Definen una relación entre un objeto y sus atributos. Constituyen la estructura de la clase/objeto. Las variables de instancia acompañan al objeto desde su creación hasta que muere.



En UML se representa del siguiente modo:



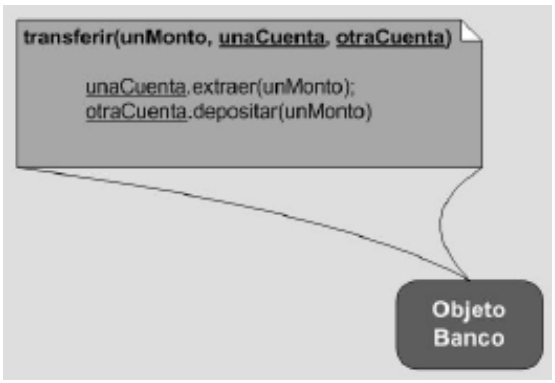
En java se declara del siguiente modo:

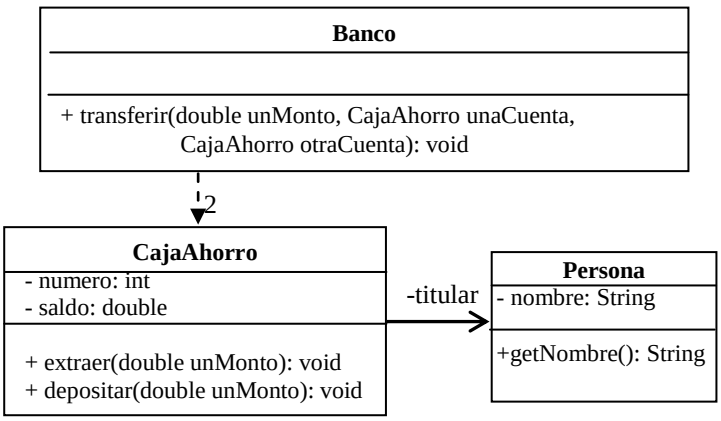
```
public class CajaAhorro{
  private double saldo;
  private Persona titular;

  public void mostrarDatos(){
    System.out.println("Nombre"+ titular.getNombre());
    System.out.println("Saldo:" + this.getSaldo());
  }
}
```

2. Parámetros

Se utilizan para nombrar objetos que el objeto receptor necesita para cumplir un requerimiento. Se refiere a los parámetros de un mensaje. La relación de conocimiento se establece dentro del cuerpo del método, y dura el tiempo que el método se encuentra activo.

Por ejemplo, para hacer una transferencia de una cuenta a otra, el banco debe saber cuáles son las cuentas intervinientes. En este caso las cuentas son parámetros que se envían junto con el monto, en el mensaje transferir para poder cumplir el requerimiento.



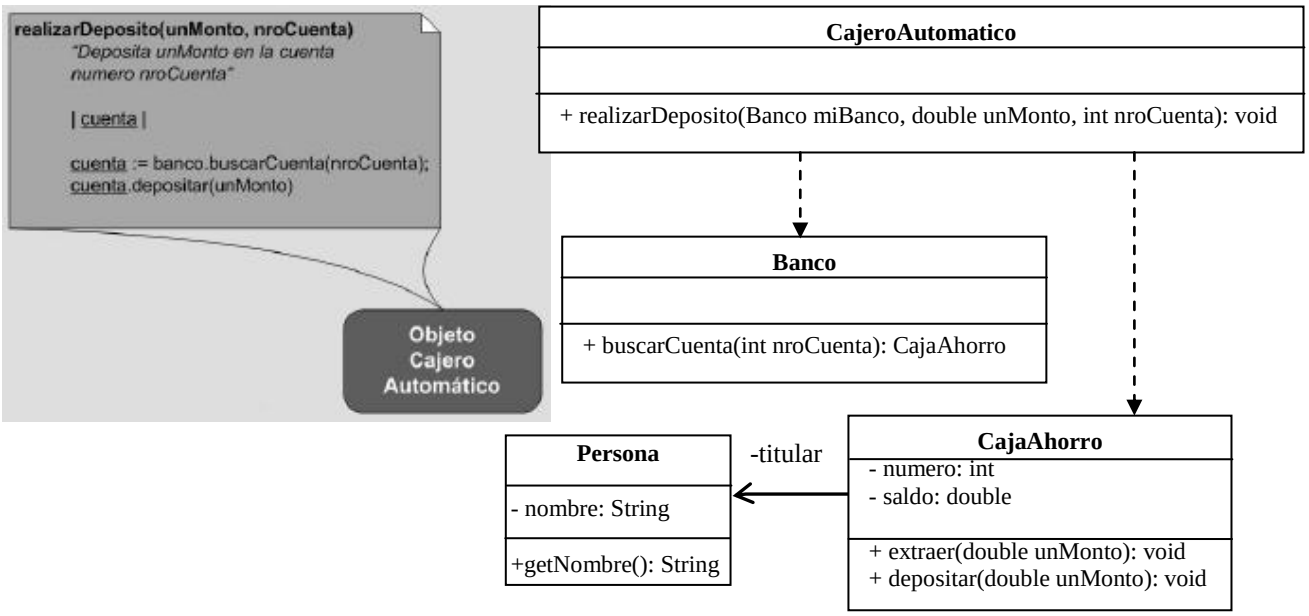



```
public class CajaAhorro{
   private int numero;
   private double saldo;
   private Persona titular;
```

```
    public void extraer(double unMonto){
          saldo = saldo - unMonto;
    }
    public void depositar(double unMonto){
          saldo = saldo + unMonto;
    }
}
public class Banco{
    public transferir(double unMonto, CajaAhorro unaCuenta, CajaAhorro otraCuenta){
          unaCuenta.extraer(unMonto);
          otraCuenta.depositar(unMonto);
    }
}
```

3. Variables temporales

Definen relaciones temporales dentro de un método. Estas variables se utilizan para nombrar objetos temporalmente. La relación con el objeto se crea durante la ejecución del método, y dura el tiempo que éste se encuentra activo. Al finalizar la activación del método, estas variables dejan de existir.



En este ejemplo, el método buscarCuenta() recibe como parámetro sólo un número de cuenta (atributo de tipo entero del objeto caja de ahorro). Con este dato, se crea la variable temporal **cuenta** de tipo CajaAhorro, que se utiliza sólo como una referencia auxiliar. No es necesaria fuera del ámbito de la activación de este método.
En java se declara del siguiente modo:

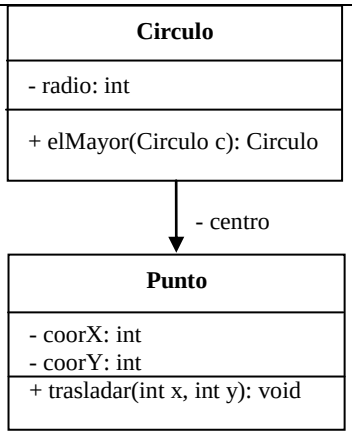
```
public class CajeroAutomatico{
        . . .
   public void realizarDeposito(Banco miBanco; double unMonto, int nroCuenta){
        CajaAhorro cuenta = miBanco.buscarCuenta(nroCuenta);
        cuenta.depositar(unMonto);
   }
}
```

4. Seudo-Varibles

Así como un objeto necesita conocer a otro objeto para enviarle un mensaje, es necesario que tenga una forma de nombrarse a sí mismo para poder enviarse mensajes. Este autoreferenciamiento se hace por medio de una seudo-variable, que toma distintos nombres en los lenguajes de programación. En algunos casos es *self*, y en el caso de Java, C# y C++ es *this*. Se dice que es una seudo-variable ya que funciona similar a una variable de instancia, excepto por dos motivos: no está definida en ninguna clase y no puede ser asignada (no se le puede asignar un valor).

Todos los objetos tienen la referencia implícita **this,** que apunta a sí mismo.

```
public class Circulo {
  private Punto centro;
  private int radio;
    . . .
  public Circulo elMayor(Circulo c) {
        if (this.getRadio() > c.getRadio()){
              return this;
         }
         else{
              return c;
         }
  }
}
```



El método elMayor() devuelve una referencia al objeto círculo que tiene mayor radio, comparando los radios del Circulo **c**, que se recibe como parámetro (c.getRadio()) y del propio círculo (this.getRadio()). En caso de que el propio resulte mayor, el método debe devolver una referencia a sí mismo. Esto se consigue con la expresión ***return this***.

Conocimiento entre objetos y Relaciones en UML

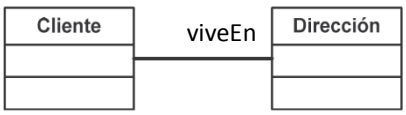
Para enviarse mensajes, los objetos deben estar relacionados, es decir, deben "conocerse". Los mensajes "navegan" por las relaciones existentes entre las clases a las que pertenecen los objetos. Las relaciones entre las clases indican cómo se comunican los objetos de esas clases entre sí.

Existen cuatro tipos de relaciones entre los elementos de un modelo UML: asociación, dependencia, realización y generalización.

1 - Asociación

Es una relación estructural entre entidades (una entidad se construye a partir de otras entidades /una clase tiene en su estructura a otra clase. Al codificar esto se representa como un atributo que es una instancia de otra clase.

La asociación se representa con una línea continua que a veces incluye una etiqueta y otros elementos para indicar la multiplicidad y roles de los objetos involucrados.



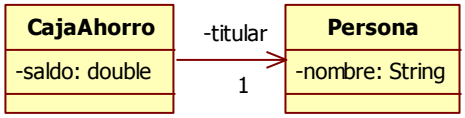
Navegación de las asociaciones

Aunque las asociaciones suelen ser bidireccionales (se pueden recorrer en ambos sentidos), en ocasiones es necesario hacerlas unidireccionales, es decir, restringir la navegación en un solo sentido.

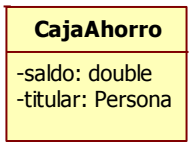
Gráficamente, cuando la asociación es unidireccional, la línea termina en una punta de flecha que indica el sentido de la navegación.

Se observa la cardinalidad, representada por el número 1, que indica que una caja de ahorro tiene un solo titular.

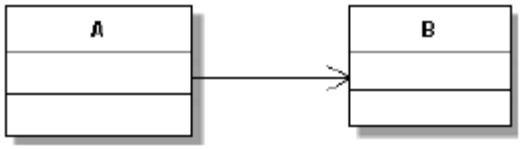
Al estar representada gráficamente la navegación, la interpretación del problema es más fácil de visualizar.



Esta relación también puede representarse como:



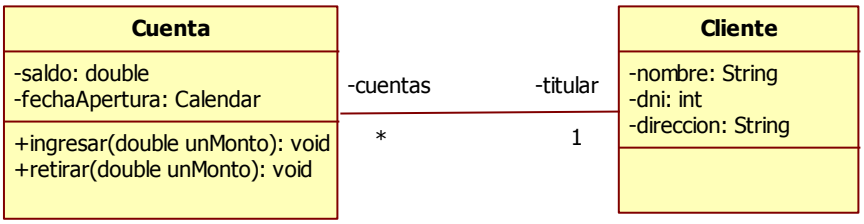
Un diagrama UML genérico será:



Este diagrama además permite observar que:

- La clase A depende de la clase B
- La clase A conoce la existencia de la clase B, pero la clase B no conoce la existencia de la clase A (sentido de la flecha)
- Todo cambio que se haga en la clase B podría afectar a la clase A. Sin embargo, si A no conoce los detalles internos de B (*Principio de Ocultación)*, esto fortalecerá el diseño, y evitará cambios en cascada.

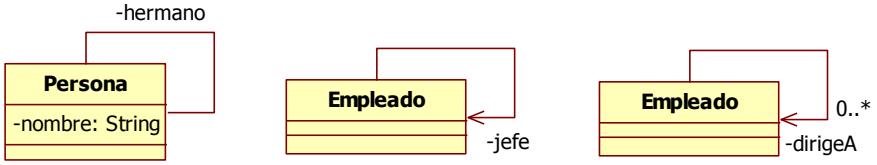
La asociación bidireccional se representa con una línea sin flecha:



La gráfica indica que una cuenta puede tener un solo titular de tipo Cliente, pero que cada cliente puede tener más de una cuenta.

Relaciones involutivas

Son las relaciones en las que interviene el mismo tipo de entidad desempeñando distintos roles. Es decir, la misma clase aparece en los dos extremos de la asociación.



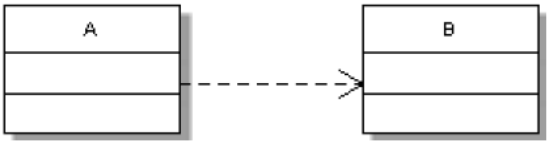
2 – Dependencia

Es una relación de uso entre dos entidades. Una clase depende de la funcionalidad que ofrece otra clase. Puede verse desde el punto de vista "Cliente / Servidor", es decir, existe una entidad que necesita de un "servicio" y otra entidad que lo provee. Una clase es "cliente" del servicio de otra clase.

Se dice que es una relación "débil", ya que no forma parte de la estructura de la clase dependiente.

En UML se representa con una flecha discontinua que va desde la clase "cliente" hasta la clase que ofrece el "servicio/funcionalidad".

Un diagrama genérico sería:



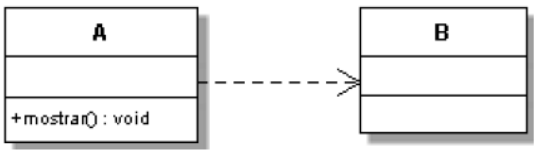
Este diagrama permite observar que:

- La clase A usa la clase B y depende de ella
- La clase A conoce la existencia de la clase B, pero la clase B no conoce la existencia de la clase A
- Todo cambio que se haga en la clase B podría afectar a la clase A. Sin embargo, si A no conoce los detalles internos de B (Principio de Ocultación), esto fortalecerá el diseño, y evitará cambios en cascada.

Cómo se traduce a código

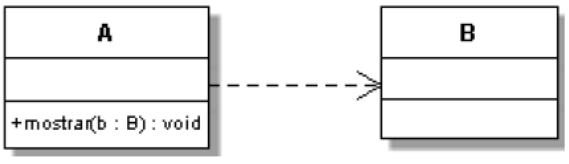
Existen dos situaciones posibles

a) En un método de la clase **A** se instancia un objeto de tipo **B** y posteriormente se lo usa (**variable temporal**).




```
public class A {
   public void mostrar() {
        b = new B();
        System.out.print(b);
         /* etc */
   }
}
```

b) En un método de la clase **A** se recibe por parámetro un objeto de tipo **B** y posteriormente se lo usa.

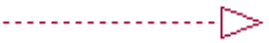



```
public class A {
    public void mostrar(B b) {
        System.out.print(b);
        /* etc */
    }
}
```

3 – Realización

Es una relación de contrato con otra clase: una clase (interfaz) especifica un contrato que otra clase garantiza que cumplirá. Se la utiliza para implementar una interfaz. En lenguajes como java o php se utiliza la palabra reservada "implements".

La parte realizante cumple con una serie de especificaciones propuestas por la clase realizada (interfaces).



Se representa con una línea discontinua con una punta de flecha cerrada y vacía

4 - Generalización

Es una relación en la que el elemento generalizado (padre) puede ser substituido por cualquiera de los elementos hijos, ya que comparten su estructura y comportamiento.

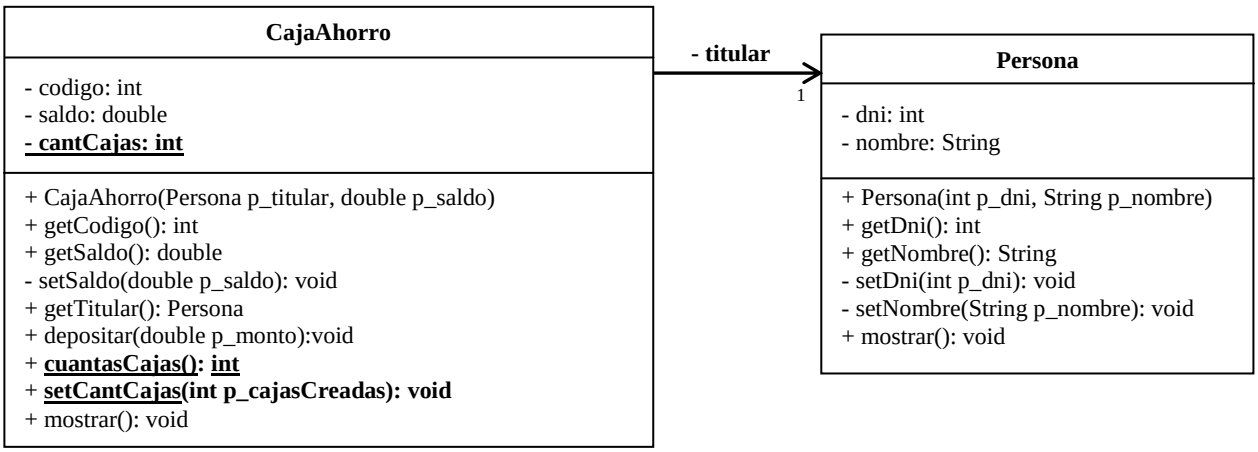
Gráficamente, la generalización se representa con una línea con punta de flecha cerrada y vacía.

Esta relación será tratada en detalle en temas posteriores.

Integrando conceptos: un ejemplo en java

Generar en java las clases representadas en UML, con sus respectivos constructores y métodos de acceso a sus variables privadas.
Generar además una clase que permita crear una caja de ahorro para María Gonzales, cuyo DNI es 25.132.456, con un saldo inicial de $1000, y otra para Juan Dominguez, DNI 17.425.236, sin saldo inicial. Luego mostrar en pantalla el titular con su saldo.

Diagrama de Clases en UML:



1. Generar la clase Persona

```
/** Permite crear y manipular un objeto persona
  * @author Juan Perez
  */
public class Persona{
   private int dni;
   private String nombre;

   Persona(int p_dni, String p_nombre){
       this.setDni(p_dni);
       this.setNombre(p_nombre);
   }
   private void setDni(int p_dni){
            this.dni = p_dni;
   }
   public int getDni(){
            return this.dni;
   }
   private void setNombre(String p_nombre){
            this.nombre = p_nombre;
   }

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

```
   /** Permite visualizar en pantalla las características de un objeto persona
    */
   public void  mostrar(){
        System.out.println("*****   metodo mostrar() de Persona   *****");
        System.out.println("DNI: "    + this.getDni());
        System.out.println("Nombre: " + this.getNombre());
   }
} // fin clase Persona
```

2. Generar la clase CajaAhorro

```
/** Permite crear y manipular una cuenta de caja de ahorro.
  * @author Juan Perez
  * @version 1
  */

public class CajaAhorro{
    private int codigo;
    private double saldo;
    private Persona titular;
    private static int cantCajas;

/** Constructor sin saldo inicial.
  * @throws No dispara ninguna excepcion.
  */
    public CajaAhorro(Persona p_titular) {
        this.setTitular(p_titular);
        this.setSaldo(0);
        this.setCantCajas(this.cantCajas + 1);
        this.setCodigo(this.cantCajas); // el ultimo numero de caja creada pasa a ser el codigo
    }

/** Constructor con saldo inicial.
  */
    CajaAhorro(Persona p_titular, double p_saldo) {
        this.setTitular(p_titular);
        this.setSaldo(p_saldo);
        this.setCantCajas(this.cantCajas + 1);
        this.setCodigo(this.cantCajas);
    }

/** Retorna el codigo de la cuenta.
  * @return Un valor de tipo int.
  */
   public int getCodigo(){
     return this.codigo;
   }

/** Asigna p_codigo al codigo de la cuenta.
  * @param int p_codigo.
  * @return No devuelve ningún valor.
  */
  private void setCodigo(int p_codigo){
     this.codigo = p_codigo;
  }

/** Retorna el titular de la cuenta.
  * @return Un objeto de tipo Persona.
  */
  public Persona getTitular(){
     return this.titular;
  }

/** Asigna p_titular al titular de la cuenta.
  * @param Persona p_titular.
  */
  private void setTitular(Persona p_titular){
     this.titular = p_titular;
  }
```

```
/** Retorna el saldo actual de la cuenta.
  * @return Un valor de tipo double.
  */
  public double getSaldo(){
     return this.saldo;
  }

/** Asigna p_saldo al saldo actual de la cuenta.
  * @param double p_saldo.
  * @return No devuelve ningún valor.
  */
  private void setSaldo(double p_saldo){
     this.saldo = p_saldo;
  }

/** Agrega p_monto al saldo actual de la cuenta.
  * @param double p_monto.
  */
  public void depositar(double p_monto){
         this.setSaldo(this.getSaldo() + p_monto);
  }


/** Devuelve el número de cajas de ahorro creadas.
  * @return devuelve un entero.
  */
    public static int cuantasCajas () {
        return cantCajas;   // this hace referencia a un objeto por lo tanto no
    }                       // puede ser usada en un método static

/** Asigna un valor a la variable de clase cantCajas.
  * @return no retorna nada.
  */
public static void setCantCajas(int p_cajasCreadas) {
        cantCajas = p_cajasCreadas;  // this hace referencia a un objeto. No
}                                    // puede ser usada en un método static
public void  mostrar(){
     System.out.println("\n" + "****  metodo mostrar() de CajaAhorro  ****");
     this.getTitular().mostrar();
     System.out.println("Codigo de cuenta: " + this.getCodigo());
     System.out.println("Saldo: "            + this.getSaldo());
   }
} // fin clase CajaAhorro
```

3. Generar una clase ejecutable de prueba

```
public class Banco{
   public static void main(String arg []){
      Persona cliente_1 = new Persona(25132456, "Maria Gonzales");
       Persona cliente_2 = new Persona(17425236, "Juan Dominguez");

      CajaAhorro.setCantCajas(0); //inicializa el contador de cajas, antes de instanciar
      CajaAhorro caja_1 = new CajaAhorro(cliente_1, 1000);
       CajaAhorro caja_2 = new CajaAhorro(cliente_2);
      System.out.println("Se crearon: " + CajaAhorro.cuantasCajas() + " cajas");
       caja_1.depositar(200);
       caja_2.setSaldo(500);
       caja_2.depositar(300);
      cliente_1.mostrar();
      caja_1.mostrar();
      caja_2.mostrar();
   }
} // fin clase Banco
```

4. Resultado de la ejecución

```
Se crearon: 2 cajas

---------  metodo mostrar() de Persona  ----------
DNI: 25132456
Nombre: Maria Gonzales

*********  metodo mostrar() de CajaAhorro  ********
---------  metodo mostrar() de Persona  ----------
DNI: 25132456
Nombre: Maria Gonzales
Numero de cuenta: 1
Saldo: 1200.0

*********  metodo mostrar() de CajaAhorro  ********
---------  metodo mostrar() de Persona  ----------
DNI: 17425236
Nombre: Juan Dominguez
Numero de cuenta: 2
Saldo: 800.0
```

5. Generación documentación

Class CajaAhorro

`java.lang.Object`

`CajaAhorro`

`public class` **`CajaAhorro`** `extends java.lang.Object`

Permite crear y manipular una cuenta de caja de ahorro.

Version:

1

Author:

Juan Perez

Constructor Summary

`CajaAhorro``(Persona p_titular)` Constructor sin saldo inicial.
`CajaAhorro``(Persona p_titular, double p_saldo)` Constructor con saldo inicial.

Method Summary

`static int`	**`cuantasCajas`**`()` Devuelve el número de cajas de ahorro creadas.
`void`	**`depositar`**`(double p_monto)` Agrega p_monto al saldo actual de la cuenta.
`int`	**`getCodigo`**`()` Retorna el codigo de la cuenta.
`double`	**`getSaldo`**`()` Retorna el saldo actual de la cuenta.
`Persona`	**`getTitular`**`()`

	Retorna el titular de la cuenta.
void	**mostrar**()
static void	**setCantCajas**(int p_cajasCreadas) Asigna un valor a la variable de clase cantCajas.
void	**setCodigo**(int p_codigo) Asigna p_codigo al codigo de la cuenta.
void	**setSaldo**(double p_saldo) Asigna p_saldo al saldo actual de la cuenta.
void	**setTitular**(Persona p_titular) Asigna p_titular al titular de la cuenta.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

CajaAhorro

```
public CajaAhorro(Persona p_titular)
```

Constructor sin saldo inicial.

Throws:

No - dispara ninguna excepcion.

CajaAhorro

```
CajaAhorro(Persona p_titular,
           double p_saldo)
```

Constructor con saldo inicial.

Method Detail

cuantasCajas

```
public static int cuantasCajas()
```

Devuelve el número de cajas de ahorro creadas.

Returns:

devuelve un entero.

---

depositar

```
public void depositar(double p_monto)
```

Agrega p_monto al saldo actual de la cuenta.

Parameters:

double - p_monto.

---

getCodigo

```
public int getCodigo()
```

Retorna el codigo de la cuenta.

Returns:
Un valor de tipo int.

---

getSaldo

```
public double getSaldo()
```

Retorna el saldo actual de la cuenta.
Returns:
Un valor de tipo double.

---

getTitular

```
public Persona getTitular()
```

Retorna el titular de la cuenta.
Returns:
Un objeto de tipo Persona.

---

mostrar

```
public void mostrar()
```

---

setCantCajas

```
public static void setCantCajas(int p_cajasCreadas)
```

Asigna un valor a la variable de clase cantCajas.

---

setCodigo

```
public void setCodigo(int p_codigo)
```

Asigna p_codigo al codigo de la cuenta.
Parameters:
`int` - p_codigo.

---

setSaldo

```
public void setSaldo(double p_saldo)
```

Asigna p_saldo al saldo actual de la cuenta.
Parameters:
`double` - p_saldo.

---

setTitular

```
public void setTitular(Persona p_titular)
```

Asigna p_titular al titular de la cuenta.
Parameters:
`Persona` - p_titular.