Capítulo 11. Programación orientada a objetos

A. J. Pérez

Programación orientada a objetos

Modelos, clases y objetos

" ... Un modelo es una simplificación que representa algún aspecto o idea interesante de la realidad. En un modelo se interpreta la realidad, centrando la atención en los aspectos relacionados con la solución del problema; ignorando los detalles innecesarios. "

El objetivo de un programa es resolver un problema o aplicar una idea de solución. Para llegar a una solución, primero se concibe un modelo de la realidad, que no necesita reflejar todos los detalles de la misma; se centra sólo en los aspectos que son relevantes para la solución que se pretende concretar. Después, basada en el modelo, se desarrolla una solución (datos y algoritmos) que se expresa por medio de un lenguaje de programación.

Actualmente los lenguajes de programación más comunes son los orientados a objetos. La programación orientada a objetos (POO) está cerca a la forma humana de pensar, haciendo posible la descripción de los patrones del mundo real. Una razón de esto es que la POO proporciona un medio para describir los conceptos que se incluyen en los modelos. Este medio de representación, son las clases.

El elemento básico de la programación orientada a objetos en Java es la *clase*. Una clase define la forma y el comportamiento de un objeto. Cualquier concepto que se quiera representar (modelar) y utilizar en un programa Java estará organizado como una clase de objeto.

Programación orientada a objetos (POO)

La programación orientada a objetos es la sucesora de la programación estructurada en la que se concebían los programas como un conjunto de piezas de código (subprogramas) reutilizables que utilizan parámetros de entrada y producen unos resultados. Los programas estructurados son un conjunto de procedimientos y funciones que se llaman entre sí.

El problema principal que sufre la programación estructurada es que la reutilización del código es limitada y difícil de conseguir porque los procedimientos y funciones son difícilmente generalizables. Tampoco es fácil implementar estructuras de datos que sean suficientemente polivalentes y genéricas.

El enfoque orientado a objetos se basa en el paradigma de que un programa es un conjunto datos que describen las entidades (objetos y eventos) de la vida real. Por ejemplo, un programa de contabilidad trabaja con facturas, mercancías, almacenes, inventarios, ventas, etc. Los objetos describen las características (propiedades) y el comportamiento (métodos) de esas entidades del mundo real.

Las principales ventajas -y objetivos- de la POO están relacionadas con un desarrollo más rápido de soluciones software de cierta complejidad; además de hacer más fácil su mantenimiento. La

programación orientada a objetos permite reutilizar fácilmente el código basándose en reglas simples y universales (principios).

Principios fundamentales de la programación orientada a objetos

Un lenguaje de programación orientado a objetos, no sólo permite trabajar con clases y objetos, sino que debe permitir la aplicación y utilización de los principios y conceptos de la programación orientada a objetos: modularización, *encapsulación, abstracción, herencia, y polimorfismo*.

- **Modularización**
  Proceso de descomposición de un sistema en un conjunto de módulos (piezas) poco acoplados (independientes) y cohesivos (con función específica).
- **Encapsulación**
  Permite ocultar los detalles y la manera de resolver las características de las clases con el fin de presentar una interfaz simple y clara que permite cualquier cambio de la implementación interna sin afectar al comportamiento externo.
- **Abstracción**
  Permite representar (modelar) los objetos desde la perspectiva que interesa para una necesidad específica, e ignorar todos los demás detalles.
- **Herencia**
  Permite organizar jerarquías de clases para mejorar la legibilidad del código y hace posible la reutilización de la funcionalidad.
- **Polimorfismo**
  Permite la generalización y abstracción de comportamientos similares o iguales en diferentes objetos.

Modularización

La descomposición de un sistema en un conjunto de módulos implica un acoplamiento entre ellos. El acoplamiento es la medida de fuerza de la dependencia establecida por la conexión entre un módulo y otro. El acoplamiento fuerte complica un sistema porque los módulos son más difíciles de comprender, cambiar o corregir.

La cohesión mide el grado de coincidencia para un mismo fin entre los elementos de un mismo módulo. [Booch, 96]

Encapsulación

La encapsulación es uno de los principios fundamentales de la programación orientada a objetos. Consiste en aplicar un control que oculta la información para su protección. Un objeto debe proporcionar al usuario los medios de control necesarios; no más. Por ejemplo un usuario de un ordenador sólo ve y accede a la pantalla, el teclado y el ratón, y todo lo demás está oculto. No se necesita acceder al interior del equipo para una utilización normal según, reduciendo el riesgo de estropear algo. En las las clases, la decisión de lo que está oculto y lo que es visible, es criterio del programador. En POO se puede configurar como privado (oculto) cualquier atributo o método que se considere no debe ser accesible por otros objetos de otras clases.

Ejemplo: uso de un método privado

En la implementación de lista enlazada que hay en el capítulo de *Estructuras dinámica lineales*; la operación `add()` que permite añadir un nuevo elemento al final de la lista, utiliza un método auxiliar privado, `obtenerNodo()` para obtener el nodo que hay en una determinada posición de índice.

```
/**
* Representa la implementación básica de una lista enlazada
*/
public class ListaEnlazada {

	//...

	//métodos

	 /**
	  * Añade un elemento al final de la lista
	  * @param elem - el elemento a añadir
	  */
	 public void add(Object elem) {

		 if (numElementos == 0) {
			// si la lista está vacía enlaza el nuevo nodo en la cabeza
			cabeza = new Nodo(elem);
		  }
		 else {
			// obtiene el nodo del último elemento y enlaza el nuevo
			nodo
			obtenerNodo(numElementos-1).siguiente = new Nodo(elem);
		}
		numElementos++;              // actualiza el numero de elementos
	 }

	/**
	 * Obtiene el nodo correspondiente al índice
	 * @param indice - posición del nodo a obtener
	 * @return el nodo que ocupa la posición indicada por el índice
	 */
	private Nodo obtenerNodo(int indice) {

		// lanza excepción si el índice no es válido
		if (indice >= numElementos || indice < 0) {
			throw new IndexOutOfBoundsException("Índice incorrecto: " +
			indice);
```

```
            }
            // recorre la lista hasta llegar a la posición buscada
            Nodo actual = cabeza;
            for (int i = 0; i < indice; i++)
                  actual = actual.siguiente;
            return actual;
      }
} //class
```

La clase **`ListaEnlazada`** no desvela información acerca de cómo funciona internamente y le permite cambiar más tarde la implementación de cómo obtener un elemento de la lista sin que otras clases noten la diferencia.

Otro ejemplo de encapsulación es la clase **`ArrayList`** de la API estándar de Java. Si se ve el código fuente de esta clase, se puede comprobar que dispone de decenas de campos y métodos que se definen como privados y son accesibles sólo desde dentro de la propia clase.

```
package java.util;

public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, Serializable {
    private static final long serialVersionUID = 8683452581122892189L;
    private transient Object [] elementData ;
    private int size;

    private void fastRemove ( int index) {... }
    private void rangeCheck ( int index) {... }
    private void writeObject (ObjectOutputStream s) {... }
    private void readObject (ObjectInputStream s) {... }
}
```

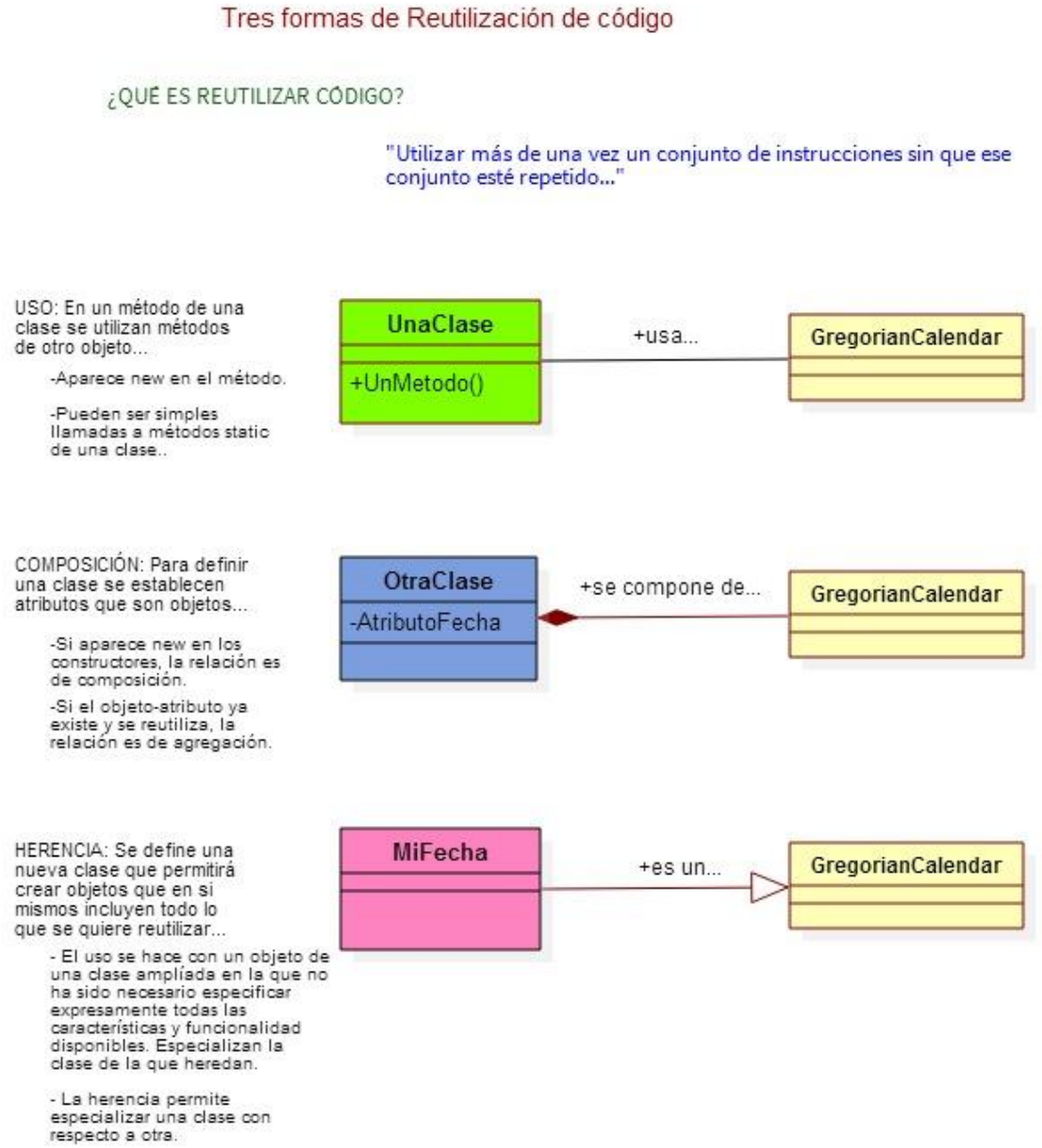
Además de los métodos públicos evidentes, la clase **`ArrayList`** ha ocultado cosas; se trata de estructuras internas necesarias para su funcionamiento. La preservación de los elementos que forman parte de la lista y cuántos son (`elementData` y `size`) y algunos métodos que no deben utilizarse fuera de clase. Cómo ocultar estos detalles y asegurar que nadie más que la propia clase **`ArrayList`** modifica los datos directamente. Si todos los atributos de la clase **`ArrayList`** fueran públicos, sería muy difícil conseguir que los usuarios actualicen adecuadamente el atributo `size` y `elementData` . Debido a que estos atributos están ocultos, la clase **`ArrayList`** realiza una gestión fiable y sin errores de esos atributos.

Herencia

La *herencia* es uno de los principios fundamentales de la programación orientada a objetos. **El principal fin operativo de la herencia es el de proporcionar un mecanismo para a la reutilización masiva y automatizada de código; de una manera organizada y óptima**. Permite que una clase incorpore el comportamientos y características de otra, la clase más general. Ver la noción de reutilización de código

Por ejemplo, en la familia biológica de los felinos; todos tienen cuatro patas y son depredadores. Esta característica y funcionalidad se puede escribir una sola vez en la clase de **`Felino`** y todas las especies de felinos a través de unas clases específicas (**`Tigre`**, **`Leon`**, **`Puma`**, **`Lince`**, etc.) las pueden reutilizar.

Otro ejemplo, más avanzado técnicamente, es el que se muestra en la siguiente figura donde se plantea unas clases **`OtraClase`** y **`MiFecha`** que aplicarían respectivamente el patrón *Adaptador de objeto* y *Adaptador de clase* para la *reutilización fácil* de la clase **`GregorianCalendar`** de la API de Java. Precisamente, los adaptadores de clase se basan en el mecanismo de la *herencia*.



¿Cómo se utiliza la herencia en Java?

Para la herencia en Java, se utiliza la palabra reservada **`extends`** . En Java, **una clase puede**

heredar sólo de una *clase base* (*herencia simple*) , a diferencia de C + +, que admite la *herencia múltiple*.

La restricción de que la herencia en Java sea simple, se debe al hecho de que la herencia desde varias clases a la vez, con posibles métodos iguales, puede producir ambigüedad y dificultar la utilización de esa multiplicidad. La clase que hereda se llama derivada de otra *clase base* (**`super`** clase).

En Java existe una segunda **modalidad de herencia que consiste en heredar responsabilidades de implementación que sí puede ser múltiple**; se utiliza la palabra reservada **`implements`** para heredar las responsabilidades de implementar varias *interfaces*.

Ejemplo. Extensión de clases

La clase base **`Felino`**:

```
public class Felino {                       // nombre latino de gato

    private boolean macho;

    public Felino() {
        this(false);
    }

    public Felino(boolean m) {
        macho = m;
    }

    public boolean isMacho() {
        return macho;
    }

    public void setMacho(boolean m) {
        macho = m;
    }
}
```

La clase derivada **`Leon`**:

```
public class Leon extends Felino {

      private int peso;

      public Leon(boolean m, int p) {
```

```
            super(m);          //llama al constructor de la clase base
            peso = p;
      }

      public int getPeso() {
            return peso;
      }

      public void setPeso(int p) {
            peso = p;
      }
}
```

Palabra clave super

En el ejemplo anterior en el constructor de **Leon** se ha utilizado la palabra clave **`super`** . Hace referencia a la clase base y permite el acceso a sus métodos, constructores y variables miembro. Con **`super`**`()` se llama al constructor de la clase base. Con **`super`**`.` se puede acceder a los miembros -atributos y métodos- de la clase base.

En Java, los atributos y los métodos heredados de la clase base se pueden reescribir (*override*). Esto significa que se pueden reemplazar las implementaciones originales de la clase base. Para ejecutar la versión de la clase base se puede utilizar **`super`**.

Herencia y constructores

Al heredar de una clase, cada uno de los constructores de la clase derivada debe llamar al constructor correspondiente de la clase base para que se haga cargo de inicializar los atributos heredados. Si no se hace de forma explícita en la primera instrucción del constructor, el compilador hará -de oficio- una llamada al constructor por defecto de la clase base: `super()`. Si la clase base no tiene un constructor por defecto o predeterminado, es obligatorio llamar explícitamente a algunos de los otros constructores de la clase base. La falta de una llamada explícita en la primera instrucción del método constructor provoca un error de compilación.

Ejemplo. Constructor y super

La clase **Leon** anterior, no tiene constructor por defecto o predeterminado. Si se deriva una nueva clase se comprueba cómo puede influir este hecho:

```
public class LeonAfricano extends Leon {

      // ...

      public LeonAfricano(boolean m, int p) {
        super(m, p);  // Si se omite, la clase no se compila
      }
```

```
      public String toString() {
            return String.format("(LeonAfricano, macho: %s, peso: %s)",
                                       this.isMacho(), this.getPeso() );
      }

      // ...
}
```

La llamada al constructor de la clase base debe ser siempre la primera línea del constructor de la clase derivada. De lo contrario, el compilador dará un error. La idea es que los atributos de la clase base deben inicializarse antes de empezar a inicializar los de la clase heredera, ya que pueden sobrescribir alguno de los atributos de la clase base.

Niveles de acceso y la herencia

En el capítulo sobre la *Definición de clases* se revisaron los niveles de acceso a los atributos y los métodos: **`public`**, **`private`** y **por defecto o de paquete**. Pero en Java hay otro nivel de acceso: **`protected`** que está relacionado con la herencia.

Al heredar de una clase base:

- Todos los **atributos y métodos privados son invisibles** en la clase derivada.
- Todos los **atributos y métodos públicos y protegidos son visibles** en la clase derivada.
- Todos los **atributos y métodos por omisión son visibles a la clase derivada** si está en el mismo paquete que la clase base.

```
public class Felino {                       // nombre latino de gato

    private boolean macho;

    public Felino() {
      // llama al otro constructor con el valor por defecto
       this(false);
    }

    public Felino(boolean m) {
        macho = m;
    }

    public boolean isMacho() {
        return macho;
    }
```

```
    public void setMacho(boolean m) {
        macho = m;
    }
}
```

```
public class Leon extends Felino {

      private int peso;

      public Leon(boolean m, int p) {
            super(m);               //llama al constructor de la clase base
            super.macho = true;     //ERROR, macho es invisible
            peso = p;
      }

      public int getPeso() {
            return peso;
      }

      public void setPeso(int p) {
            peso = p;
      }
}
```

Al compilar este ejemplo, se produce un error porque el atributo `macho` es privado de clase **`Felino`** y no es accesible desde la clase **`Leon`**.

la clase Object

La programación orientada a objetos se hizo popular con el lenguaje C + + donde a menudo es necesario manejar objetos de cualquier clase. En C + + se resuelve este problema de una manera que no se considera un estilo muy orientado a objetos; con el uso de punteros.

Los creadores de Java lo resolvieron de otra manera; planteando una super clase que todas las demás clases heredan, directa o indirectamente, al que además se pueden convertir. En esta clase se ubican métodos importantes para su aplicación por defecto en cualquier objeto que pueda crearse. Esta clase se denomina **`Object`**.

En Java, cualquier clase que no deriva de otra, hereda por defecto, directamente de **`java.lang.Object`**. Cualquier clase que deriva de otra, hereda indirectamente la clase **`Object`** de la misma. Así que cada clase dispone de forma explícita o implícita de todos los métodos y

atributos definidos en **`Object`**.

Debido a esta característica cualquier objeto de cualquier clase se puede convertir a un objeto de la clase **`Object`**. Un ejemplo típico de la utilidad de la herencia implícita de **`Object`** se encuentra en la *API* **`Collection`** como se puede ver en el capítulo sobre las *Estructuras dinámicas lineales.* Las colecciones de Java pueden trabajar con cualquier clase de objeto, ya que son considerados como instancias de la clase **`Object`**.

API´s y objetos estándar

En Java hay muchas clases ya pre-escritas que constituyen la plataforma Java. Estas clases se denominan *bibliotecas de clases estándar*.

Java fue la primeras plataformas que dispuso de un amplio conjunto de API's de propósito general disponible en cualquier lugar sin necesidad de recompilación.

En Java, además, se pueden añadir más bibliotecas y API's externas.

Ejemplo: Utilización de Object

```
public class EjemploObject {
    public static void main(String... args) {
        LeonAfricano granGato = new LeonAfricano();
        // casting implicito
        Object obj = granGato;
    }
}
```

En el ejemplo, se produce una conversión implícita de **`LeonAfricano`** a **`Object`**. Esta operación se denomina *upcasting;* está permitido porque **`LeonAfricano`** es indirectamente una clase derivada de **`Object`** a través de la primera clase de la jerarquía, **`Felino`**.

El método Object.toString ()

Uno de los métodos más utilizados procedentes de la clase **`Object`**, es `toString()`. Devuelve una representación en forma de texto del objeto. Todo objeto tiene un método como este, y por lo tanto cualquier objeto de cualquier clase tiene una representación de texto. Este método se utiliza implícitamente cuando se imprime un objeto utilizando `System.out.println()`.

Ejemplo: Utilización de Object.toString ()

```
public class EjemploToString {
      public static void main(String... args) {
            LeonAfricano granGato = new LeonAfricano();
            System.out.println(granGato.toString());
            System.out.println(granGato);
```

```
            // muestra dos lineas iguales con:
            // LeonAfricano@de6ced
      }
}
```

Salida:

```
LeonAfricano@de6ced
LeonAfricano@de6ced
```

Como **LeonAfricano** no reescribe (*override*) el método `toString()`, en este caso se llama a la implementación de la clase base **Object**. Las clases **Leon** y **Felino** tampoco tienen reescrito el método, por lo tanto, se ejecuta el heredado de la clase **Object**. Como resultado se muestra el nombre de la clase de objeto y un valor extraño después del símbolo @. Es un *código hash interno del objeto en notación hexadecimal*. No es una dirección de memoria y varía de un objeto a otro.

La implementación original del método **Object**.`toString()` de Java es:

```
public class Object {
    // ...
    public String toString() {
        return getClass().getName() +
            "@" + Integer.toHexString(hashCode());
    }
    // ...
}
```

Ejemplo: Reescritura de toString()

Reescribir el método `toString()` puede ser especialmente útil como se puede comprobar:

```
public class AfricanLion extends Lion {

      // ...

      public AfricanLion(boolean m, int p) {
            super(m, p);
      }

      @Override
      public String toString() {
            return String.format("[LeonAfricano, macho: %s, peso: %s]",
                              macho, peso);
```

```
        }

        // ...
}
```

El código anterior utiliza el método **`String`**`.format(formato String, Object ... args)` para dar formato apropiado al resultado.

Desde Java 5 hay una manera de indicar al compilador que un método reescribe a otro, utilizando una anotación específica. `@Override` asegura que el compilador verifique que efectivamente se reescribe el método indicado de la clase base; se recomienda esta práctica para reducir riesgo de errores. Si se confunde accidentalmente una letra del nombre del método o los tipos de los argumentos, el compilador avisará del error.

La llamada al método `toString()`, reescrito :

```
public class EjemploToString {
        public static void main(String... args) {
                LeonAfricano granGato = new LeonAfricano(true, 150);
                System.out.println(granGato);
        }
}
```

Salida:

```
[LeonAfricano, macho: true, peso: 150]
```

Transitividad en la herencia

En matemáticas, la transitividad significa la transferencia de las relaciones. Por ejemplo en la relación *mayor que*, si `A>B` y `B>C`, entonces podemos concluir que `A>C`. Esto significa que la relación *mayor que* es transitiva.

Si la clase **`Leon`** deriva de **`Felino`** y la clase **`LeonAfricano`** deriva de **`Leon`** , implica que **`LeonAfricano`** hereda de **`Felino`**. la clase **`LeonAfricano`** cuentan con el atributo `macho`, que se define en **`Felino`**.

Ejemplo uso de la transitividad

```
public class Transitividad {

    public static void main(String... args) {
```

```
        LeonAfricano granGato = new LeonAfricano(true, 150);
        granGato.isMacho();              // método definido en Felino
        granGato.setMacho(false);        // método definido en Felino
    }
}
```

La transitividad de la herencia permite asegurar que todas las clases tienen un método `toString()` además de otros métodos de la clase **`Object`**, sin importar las clase intermedias.

Jerarquía de herencia

Para describir todos los felinos se llega a un número relativamente grandes clases, que heredan entre sí. Todas estas clases junto con la clase base **`Felino`** forman una jerarquía clases. Estas jerarquías pueden describirse más fácilmente utilizando *diagrama de clases*.

Diagramas de clases

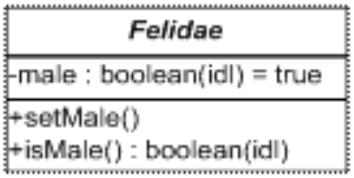
Un diagrama de clases es uno de los tipos de diagramas previstos en *UML*. *UML (Unified Modeling Language*) es una notación para la visualización de los diferentes procesos y objetos relacionados con el desarrollo de software. Un diagrama de clases se utiliza para describir gráficamente las relaciones jerárquicas entre clases.

En los diagrama de clases se ha adoptado una notación que consiste en representar las clases con un rectángulo donde se escribe el nombre de la clase y los nombres de los atributos y los métodos. Las relaciones entre clases se indican con líneas y algunos símbolos adicionales.

Una clase está representada por un rectángulo dividido en tres partes. En la parte superior aparece el nombre de la clase. En la siguiente sección se indican los atributos de la clase. En el tercer recuadro aparecen las operaciones o métodos.

Cada atributo o método lleva un de los siguientes símbolos para indicar el nivel de acceso:

+ público.
– privado.
protegido.

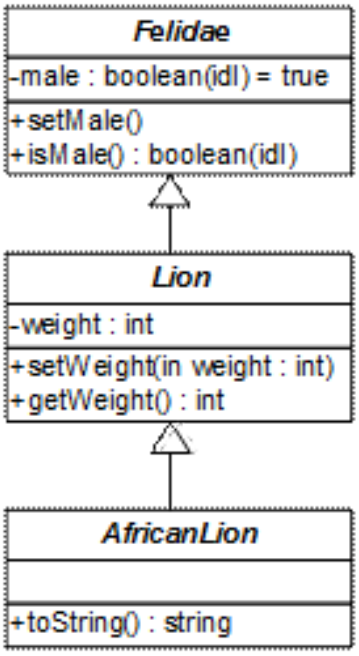


En los diagramas de clases se pueden distinguir dos grandes grupos de relaciones entre clases:

- Relaciones de *generalización* que expresan la noción de herencia o derivación.
- Relaciones de *composición*, *agregación* y *asociación*.

La *generalización*, la *composición,* la *agregación y* la *asociación;* son las formas básicas para la reutilización automatizada de código en la POO

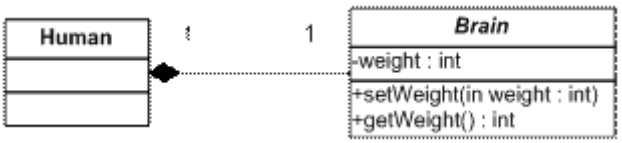
Generalización



Las flechas triangulares indican la *generalización* o *herencia*.

Composición

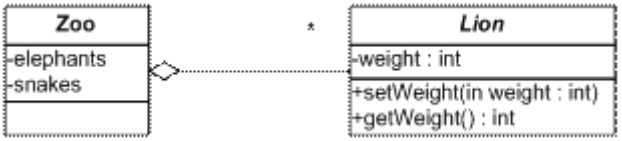
La *composición* es una relación jerárquica de asociación partitiva. Representa también la noción *todo / parte*. A la clase que actúa como parte se le llama *componente.* La clase que actúa como *el todo* lleva en su extremo un rombo lleno.



Agregación

La *agregación* es una relación jerárquica de asociación partitiva. Representa la noción *todo / parte*. A la clase que actúa como parte se le llama *agregada.* La clase que actúa como *el todo* lleva en su extremo un rombo vacío.

La *agregación* es una composición menos estricta, en la que pueden existir los componentes sin el *todo*.



Asociación

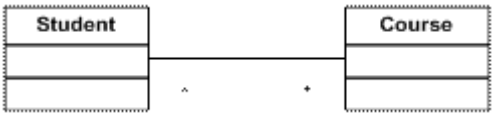
La *asociación* es un tipo de relación jerárquica entre clases. En las asociaciones se puede especificar una cardinalidad en la relación:

1 a 1

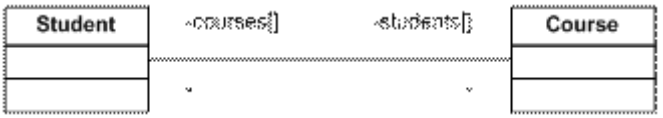
1 a muchos

muchos a 1
1 a 2
...
muchos a muchos

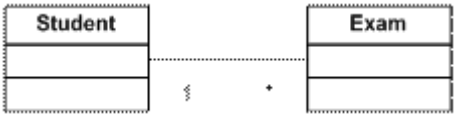
La asociación de *muchos a muchos* entre clases se expresa:



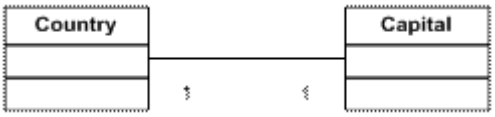
La asociación de *muchos a muchos* en un atributo se expresa:



En este caso, no es obligatorio indicar los nombres de los atributos.
Asociación de *uno a muchos* se expresa:



Asociación de *uno a uno* se expresa:



Abstracción

Abstracción significa utilización de algo que sabemos para qué sirve pero no sabemos cómo funciona en detalle. Por ejemplo, tenemos una televisión; no hace falta saber cómo funciona para poder verla y usarla. Para usar una televisión se necesita sólo el mando a distancia con un determinado número de botones asociados a los distintos comandos u órdenes habituales (interfaz).

Lo descrito para una televisión, sucede con los objetos en la programación orientada a objetos. La abstracción es algo que hacemos todos los días; consiste en dejar de lado los detalles secundarios de un objeto, y tener en cuenta los que son relevantes para resolver un problema determinado. Por ejemplo, hay una abstracción de hardware: Dispositivo de Almacenamiento (Storage Device), que puede ser un disco duro, un pendrive, disquete, CD-ROM, etc. Cada uno de ellos funciona internamente de manera diferente, pero desde la perspectiva del sistema operativo y los programas, se utilizan de la misma manera. Con un explorador de archivos, o con comandos, se puede acceder al dispositivo, como un volumen genérico para leer o escribir datos. Son las implementaciones concretas de cada interfaz de dispositivo, las que resuelven las particularidades, sin que importe cuáles son.

La abstracción es uno de los conceptos más importantes en la programación y la programación orientada a objetos; permite escribir código que funcione con las estructuras de datos abstractos (por ejemplo: listas, colas, pilas, diccionarios, etc.). Dado un tipo abstracto de datos se puede trabajar con su interfaz sin importar la implementación de la misma.

La abstracción permite que, dado un programa, se pueden hacer varios programas más pequeños, adecuadamente coordinados, de manera que cada uno resuelva una tarea menor de manera autónoma.

Ejemplo de datos abstractos

Se define un tipo de dato específico **`LeonAfricano`**, pero después se utiliza de una manera abstracta a través de la abstracción **`Felino`** que no está interesada en los detalles de todos los tipos de leones.

```
public class EjemploDatoAbstracto {
    public static void main(String... args) {

        Leon granGarto = new Leon(true, 150);
        Felino felino1 = granGarto;

        LeonAfricano granGatoAfricano = new LeonAfricano();
        Felino felino2 = granGatoAfricano;
    }
}
```

Interfaces

Una **`interface`** Java **representa la definición o especificación de un rol o funcionalidad que debe realizar de manera abstracta un objeto**. **Define el comportamiento previsto de un objeto sin especificar la forma de realizar o resolver ese comportamiento**.

Un objeto puede tener muchos roles o implementar muchas interfaces que pueden usarse desde diferentes perspectivas.

Por ejemplo, un objeto de la clase **`Persona`** puede tener asociados varios roles y comportamientos:

- Rol de **`Ciudadano`** con un comportamiento de *ir a votar.*
- Rol de **`Comercial`** con un comportamiento de *vender determinados artículos en un comercio.*
- Rol de **`Padre`** con un comportamiento de *jugar con su hijo.*
- Rol de **`Contribuyente`** con un comportamiento de *pagar impuestos*.

Puede haber roles comunes entre diferentes personas, por ejemplo, todos los ciudadanos pueden poner en práctica el comportamiento de *ir a votar* de manera diferente:

- *Pepe vota a…*
- *Luis vota en blanco.*

- *Pedro no vota.*
- ...

Se podría decir que para el comportamiento abstracto de *ir a votar* puede haber implementaciones concretas diferentes.

Signatura de métodos

Una **`interface`** sólo puede tener declaraciones de métodos y constantes. La declaración de un método es la combinación de un tipo de valor de retorno del método y la signatura del del método.

La *signatura de un método* es la combinación de un *identificador* más una *descripción de los argumentos* (tipo y secuencia) del método. En una clase / interface todos los métodos deben tener diferentes signaturas y las signaturas no coincidir con los métodos heredados.

El tipo de retorno no es parte de la signatura de un método; la razón es que si dos métodos se diferencian sólo en el tipo de retorno (por ejemplo, dos clases que heredan entre sí), no se puede identificar de forma exclusiva el método que se debe llamar.

La implementación de un método es un bloque de código fuente dentro de una clase delimitado entre llaves; también llamado cuerpo del método.

Ejemplo: Uso de interface

```
Public Interface Reproducible {
    Mamifero[] reproducir(Mamifero compañero);
}
```

Así sería la clase **`Leon`**, que implementa la interfaz **`Reproducible`**:

```
public class Leon extends Felino implements Reproducible {
    // ...

    public Mamifero[] reproducir(Mamifero compañero) {
        return new Mamifero[]{ new Leon(), new Leon()};
    }
}
```

La clase que implementa una interfaz debe implementar todos los métodos de la misma. Si la clase es abstracta se puede implementar cero, algunos o todos los métodos.

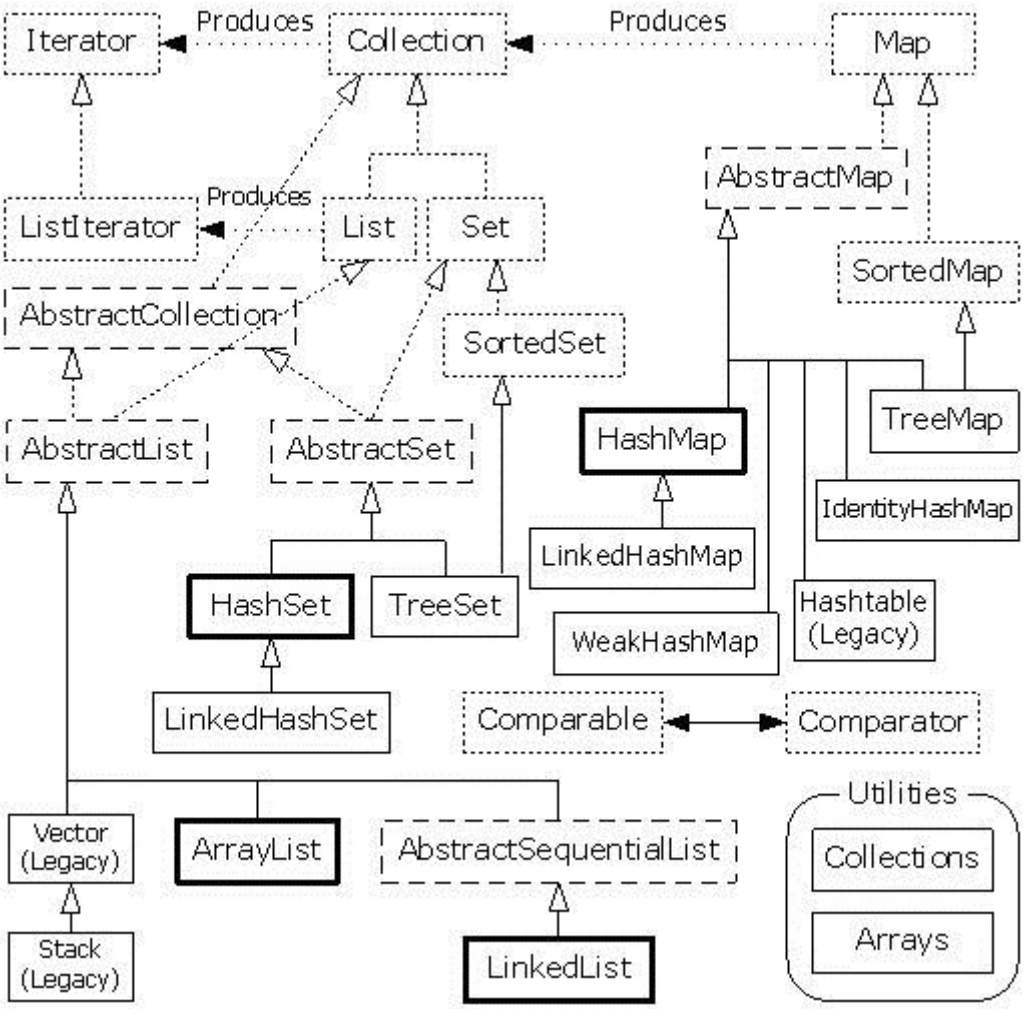
Abstracción e Interfaces

Las interfaces de Java están diseñadas para admitir resolución dinámica de métodos durante la ejecución. Para que un método sea llamado desde una clase a otra, ambas clases tienen que estar

presentes durante la compilación, para que el compilador de Java pueda comprobar que las signaturas de método son compatibles. Debe haber un mecanismo para desconectar la definición de un método o conjunto de métodos de la jerarquía de herencias.

La mejor manera de comprender la abstracción es trabajar con las interfaces. *Un componente funciona con interfaces que otros implementan*. Por lo tanto la sustitución de un componente antiguo por otro nuevo no afectará al funcionamiento, si el nuevo componente implementa la **`interface`** antigua. A una **`interface`** también se llama un *contrato*. Cada componente que implementa una **`interface`**, se dice que se adhieren a un contrato. Así que dos componentes, siempre que cumplan con las reglas del contrato, pueden comunicarse entre sí, sin saber cómo funciona internamente uno u otro.

Pueden verse buenos ejemplos de interfaces Java en el *framework* `java.util.`**`Collection`**. Todas las colecciones estándar implementan estas interfaces y los diversos componentes resuelven las diferentes implementaciones siempre bajo una interfaz común. Esta es la parte de la jerarquía de las colecciones en Java:



Las colecciones son un excelente ejemplo de una biblioteca orientada a objetos de clases e interfaces, que utiliza todos los principios básicos de la programación orientada a objetos:

abstracción, herencia, encapsulación y polimorfismo. Los tipos abstractos de datos se definen como interfaces (**`Collection, List, Set, Mapa`** , etc) y sus implementaciones específicas son su jerarquía de herencia directa o indirecta (**`ArrayList, LinkedList, HashSet, HashMap, TreeMap`** y otros) .

Cuándo utilizar la abstracción con interfaces?

Se debe utilizar la abstracción cada vez que se necesite abstracción de datos o acciones cuya implementación posterior se puede necesitar cambiar. El código escrito sometido a la disciplina de las interfaces es mucho más robusto, frente a los cambios, que el escrito para clases específicas. El trabajo con interfaces es una práctica común y muy recomendable además de una de las reglas básicas para conseguir código de calidad.

Polimorfismo

El polimorfismo permite el tratamiento de objetos de clases derivadas como objetos de la clase base. Por ejemplo, las diferentes especies de **`Felino`** (clase base) cazan de manera diferente, pero todas ellas utilizan alguna técnica de caza. El polimorfismo permite tratar cualquier especie de **`Felino`** como un animal que captura presas, no todos las mismas.

El polimorfismo se basa en la abstracción, pero en la abstracción del comportamiento de las clases. No es necesario conocer a priori, en detalle, cómo se comportará una clase, sino genéricamente, sólo para identificarlo y clasificarlo. Esto hace posible una programación especialmente flexible; uno de los grandes objetivos de la POO.

El polimorfismo utiliza intensamente la posibilidad de reescritura de los métodos, abstractos o no, de las clases derivadas con el fin de adaptar o concretar su comportamiento heredado. La abstracción se asocia con la creación de una interfaz genérica que modela el comportamiento.

Las clases abstractas

Si se quisiera especificar que clase **`Felino`** es incompleta y sólo sus herederos pueden utilizarse para producir directamente objetos, se debería especificar con la palabra clave **`abstract`** antes del nombre de la clase; significa entonces que la clase no está preparada para instanciar objetos con **`new`**. A estas clases se les llama abstractas.

Cualquier clase que tiene al menos un método abstracto debe ser abstracta. También es posible definir una clase como abstracta incluso si no tiene un método abstracto.
Las clases abstractas son algo intermedio entre una clase convencional completa y una de **`interface`**, que sería equivalente a una *clase abstracta pura* en la que todos sus métodos son abstractos.

En una clase abstracta se pueden definir métodos completos (implementados) y métodos abstractos (vacíos). Los métodos abstractos se dejan para ser implementada por las clases derivadas siendo la base del polimorfismo.

Ejemplo: Uso de una clase abstracta

```
public abstract class Felino {
	// ...

	private boolean macho;

	public boolean isMacho() {
		return macho;
	}

	public setMacho(boolean m) {
		macho = m;
	}

	public abstract boolean capturaPresa(Object presa);
 }
```

Se puede observar en el ejemplo anterior que los métodos `isMacho()` y `setMacho()son` de la forma normal, pero `capturaPresa()` va sin implementar.

```
public class Leon extend Felino {
	// ...

	public abstract boolean capturaPresa(Object presa) {
		super.ocultarse();
		this.acechar();
		super.correr();
		// ...
	}
 }
```

En el siguiente ejemplo se puede ver un caso de comportamiento abstracto implementado por una clase abstracta y una llamada polimórfica a un método abstracto. Primero definimos clase abstracta **`Animal`**:

```
public abstract class Animal {
	public void mostrarInformacion () {
		System.out.println("SOY UN:"
		+ this.getClass().getSimpleName() + ".");
		 System.out.println(getSonidoTipico());
```

```
	}

	protected abstract String getSonidoTipico();
}
```

Si definimos la clase **Gato**, que hereda la clase abstracta **Animal** y se implementa el método abstracto:

```
public class Gato extend Animal {

	public void main(String[] args) {
		Gato gato1 = new Gato();
		gato1.getSonidoTipico();
	}

	@ Override
	protected String getSonidoTipico() {
		return "Miauuuuu";
	}
}
```

Salida:

```
SOY UN: Gato.
Miauuuuu
```

En el método `mostrarInformacion()` de la clase abstracta se muestra la salida; utiliza el método abstracto `getSonidoTipico ()`, que debe ser implementado de diferentes formas para diferentes animales. Los diferentes animales tienen diferentes sonidos pero a la hora de mostrarlo es siempre la misma funcionalidad para todos los animales, y se proporciona por adelantado en la clase base.

Clases abstractas puras

Las clases abstractas e interfaces no permiten crear instancias. Si se intenta crear una instancia de una clase abstracta, se obtendrá un error en tiempo de compilación. A veces, una clase se puede declarar como abstracta incluso sin métodos abstractos, sólo para impedir la utilización directa, sin crear una clase derivada.

Las clases abstractas puras son clases abstractas que no implementan ninguno de sus métodos ni asignan valor a ninguno de sus atributos. Se asemejan a las interfaces. La principal diferencia es que una clase puede implementar muchas interfaces y heredar de una sola clase (incluso si es una clase abstracta pura).

Cuando se empezó a utilizar la herencia múltiple no había necesidad de interfaces; se resolvía con clases abstractas puras. Con la herencia simple, tuvieron que aparecer las interfaces.

Los métodos virtuales

En POO en general, los métodos que se puede volver a escribir en las clases derivadas se llaman métodos virtuales.Todos los métodos en Java son virtuales, no se definen explícitamente como tales. Si no se quiere que un método sea virtual, se debe indicar con la palabra clave **`final`**. Una clase derivada no podrá declarar ni reescribir un método con la misma signatura.

Los métodos virtuales son esenciales para la redefinición o rescritura de métodos, concepto que está en la esencia del polimorfismo.

Ejemplo: Los métodos virtuales

Tenemos una clase que deriva de otra, ambas clases tienen un método para escribir en la consola:

```
public class Leon extends Felino {
    // ...

    public void capturaPresa(Object presa) {
        System.out.println("Leon.capturaPresa()");
    }
}

public class LeonAfricano extends Leon {
    // ...
    public void capturaPresa(Object presa) {
        System.out.println("LeonAfricano.capturaPresa()");
    }
}
```

```
public class EjemploMeotodoVirtual {

    public static void main(String... args) {
        {
            Leon granGato = new Leon();
            grangato.capturaPresa(null);
            //Muestra "Leon.caturaPresa()
        }

        {
            AfricanLion granGato = new AfricanLion();
            granGato.capturaPresa(null);
            //Muestra: LeonAfricano.caturaPresa()
        }
```

```
        {
            Leon granGato = new LeonAfricano();
            granGato.capturaPresa(null);
             //Muestra: LeonAfricano.caturaPresa()
             //porque la variable granGato tiene valor de tipo LeonAfricano
        }
    }
}
```

Se hacen tres intentos de crear instancias llamando al método `capturaPresa()`. En este último caso, se comprueba que en realidad se llama al método redefinido, no al de la clase base. Esto sucede debido a que el objeto real es de la clase con el método reescrito.

Como los métodos virtuales y abstractos serán normalmente solapados, los métodos abstractos en realidad representan métodos virtuales sin aplicación específica. Todos los métodos que se definen en una interfaz son abstractos y por lo tanto virtuales, aunque esto no se especifica explícitamente en Java.

Ocultación y selección dinámica de métodos

En el ejemplo anterior se mantienen ocultas y sin usar las implementaciones de la clase base. Se puede usar lo antiguo como parte de la nueva aplicación (en caso de que no se desee reemplazar, sino para complementarla):

```
public class Leon extends Felino {
    // ...

    public void capturaPresa(Object presa) {
        System.out.println("Leon.capturaPresa()");
    }
}
```

La clase **`LeonAfricano`** queda:

```
public class LeonAfricano extends Leon {
    // ...

    public boolean capturaPresa(Object presa) {
        System.out.println("capturaPresa()");
        System.out.println("llama a: super.capturaPresa(presa)");
        super.catchPray(presa);
    }
```

```
}
```

Salida:

```
LeonAfricano.capturaPresa()
llama a: super.capturaPresa(presa)
Leon.capturaPresa()
```

Durante la ejecución, una referencia a un objeto se puede referir a una instancia de alguna clase derivada del tipo de referencia declarado. En estos casos, Java utiliza la instancia real para decidir a qué método llamar, en el caso que la clase derivada haya redefinido el método.

El polimorfismo dinámico durante la ejecución es uno de los mecanismos más poderosos que ofrece el diseño orientado a objetos para soportar la reutilización flexible de código; resuelto de una manera fiable y robusta.

Ejercicios

1. (*)Define en Java:
   - ➢ Una clase **Persona** con los atributos `nombre` y `apellidos`.
   - ➢ Una clase **Estudiante**, que derive de **Persona**, con un atributo llamado `evaluacion`.
   - ➢ Definir una clase **Trabajador** , que derive **Persona**, con los atributos `salario` y `horasTrabajadas`. Además dispone un método `sueldoPorHora()` , que se calcula la cantidad que recibe un trabajador por una horas de trabajo, en base a los salarios y las horas trabajadas.
   - ➢ En todas las clases definidas, escribe constructores, métodos de acceso a los atributos y sobreescribe el método `toString()` de manera adecuada.

2. (**) Crea y carga con datos de prueba un ArrayList de 10 estudiantes utilizando la clase **Estudiante** y ordenarlos por según la nota de `evaluacion`. Se debe implementar en la clase **Estudiante** la interfaz `java.lang.`**Comparable**.

3. (**)Crea y carga con datos de prueba un ArrayList de 10 trabajadores y ordenarlos inversamente según su `salario`. Se debe implementar en la clase **Trabajador** la interfaz `java.lang.`**Comparable**.

4. (**)Define en Java:
   - ➢ Una clase **FiguraGeometrica**, con los atributos `alto` y `ancho`. Además dispone de un método abstracto `calcularArea()`.
   - ➢ Dos clases llamadas **Triangulo** y **Rectangulo** derivadas de **FiguraGeometrica**. Las dos clases deben hacer una Iimplementan respectivamente el método `calcularArea()`.
   - ➢ Una clase **Circulo** derivada de **FiguraGeometrica**, con un constructor para inicializar los atributos heredados con el valor del radio e implementar método para calcular el área.

5. (**) Define una clase **Humano** con propiedades "nombre" y "apellidos". Define la clase **Estudiante** que hereda de **Humano**, que tiene el atributo `listaNotas`. Define la clase **Trabajador** que hereda de **Humano** con los atributos `sueldo` y `horasTrabajadas`. Implementa un método para calcular el salario por hora de cualquier trabajador. Escribe los constructores correspondientes y encapsula todos los atributos.

6. (***) Inicializa una lista o un array de 10 estudiantes y ordenarlos por el resultado de la media de sus notas, en orden ascendente. Utiliza la interfaz **Comparable**.

7. (***) Inicializar una lista o array con 10 trabajadores y ordenarlos por salario en orden descendente. Utiliza la interfaz **Comparable**.

8. (***) Define una clase abstracta **Figura** con el método abstracto `CalcularArea()` y los atributos `anchura` y `altura`. Define dos clases adicionales para **Triangulo** y

Rectangulo, que implementan `CalcularArea()`. Define una clase **Circulo** con un constructor adecuado, que inicializa los dos campos `anchura` y `altura` con el mismo valor (el radio) y aplicar el método abstracto para el cálculo del área. Crear una serie de clases para diferentes figuras y calcular el área de cada una.

9. (***) Implementa las siguientes clases: **Perro**, **Rana**, **Gato**, **Gatito** y **Tomcat** . Todos ellos son animales. Los animales se caracterizan por tener una edad , nombre y sexo. Cada animal emite un sonido utilizando un método abstracto. Crea un conjunto de diferentes animales e imprime en la consola su nombre, la edad y el sonido correspondiente a cada uno.

10. (***) Un banco tiene diferentes tipos de cuentas para sus clientes:
    a. Depósitos.
        i. Las cuentas de depósito permiten depositar y retirar dinero.
        ii. El cálculo de su interés se realiza por meses utilizando la fórmula: *numeroMeses * tipo de interés.*
        iii. Las cuentas de depósito no tienen ningún tipo de interés si su saldo es inferior a 1000.
    b. Crédito.
        i. Sólo permiten depositar.
        ii. El cálculo de su interés se realiza por meses utilizando la fórmula: *numeroMeses * tipo de interés.*
        iii. Las cuentas de crédito no tienen ningún tipo de interés durante los primeros 3 meses, si son clientes particulares; si son de empresas, es durante los 2 primeros meses.
    c. Hipotecas.
        i. Sólo permiten depositar.
        ii. El cálculo de su interés se realiza por meses utilizando la fórmula: *numeroMeses * tipo de interés.*
        iii. Las cuentas hipotecarias tienen la mitad de la tasa de interés durante los primeros 12 meses, para las empresas, y 6 meses para los particulares.

    Los clientes pueden ser particulares o empresas.
    Todas las cuentas tienen:
    - ❏ cliente
    - ❏ Saldo.
    - ❏ Tipo de interés mensual.

    Escribe en Java un modelo orientado a objetos del sistema bancario descrito. Se deben identificar las clases, interfaces, clases base abstractas e implementar la funcionalidad de para el cálculo de intereses.

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en línea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/