

TEMA 4: Diagramas de Clases

Módulo

Entornos de Desarrollo

para los ciclos

Desarrollo de Aplicaciones Multiplataforma

Desarrollo de Aplicaciones Web



ED FP-GS; Tema4:DiagramasClases

© Gerardo Martín Esquivel, Abril de 2023

Algunos derechos reservados.

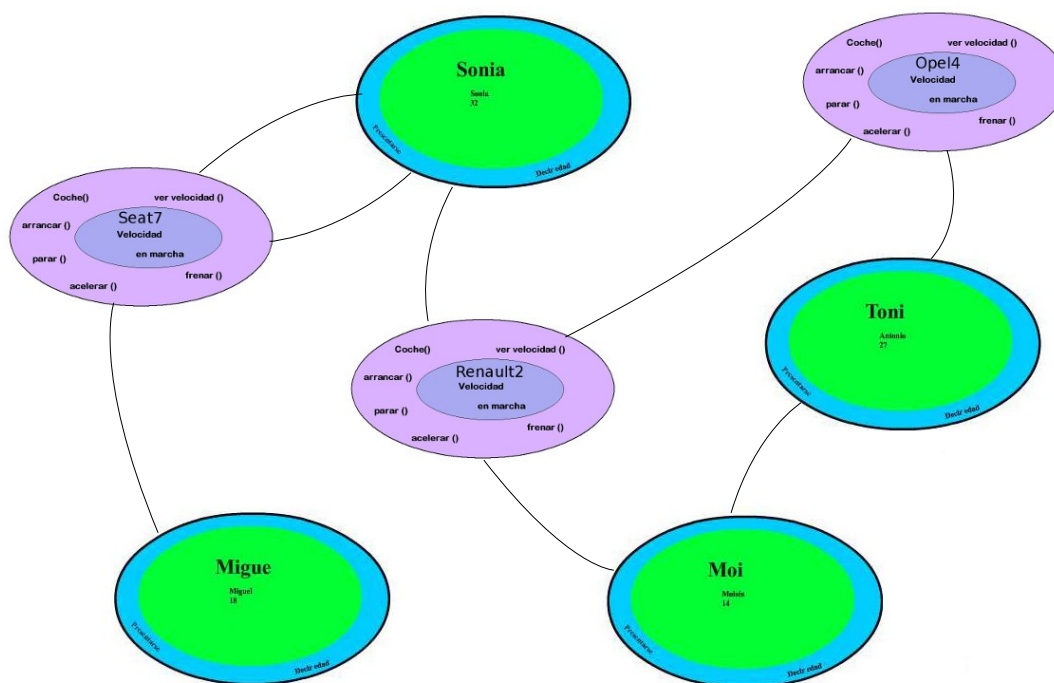
Este trabajo se distribuye bajo la Licencia "Reconocimiento-No comercial-Compartir igual 3.0 Unported" de Creative Commons disponible en <http://creativecommons.org/licenses/by-nc-sa/3.0/>

4.1 Orientación a Objetos.....	3
4.1.1 Conceptos importantes en la orientación a objetos.....	4
4.1.2 Características de un modelo orientado a objetos.....	6
Abstracción.....	6
Modularidad.....	6
Encapsulación.....	6
Herencia (jerarquía).....	7
Polimorfismo.....	9
4.2 UML.....	10
4.2.1 Diagramas de clases.....	11
4.3 Clases.....	12
Atributos.....	12
Métodos.....	13
4.3.1 Representación de clases con ArgoUML.....	13
Crear una clase.....	14
Críticas.....	14
Configuración.....	14
Generación del código.....	14
Exportar los diagramas.....	15
Ingeniería inversa.....	15
4.4 Relaciones entre clases.....	16
4.4.1 Herencia.....	16
4.4.2 Asociación.....	16
Multiplicidad.....	17
Navegabilidad.....	17
Asociación reflexiva.....	18
Crear una asociación en ArgoUML.....	18
Los atributos de navegabilidad en ArgoUML.....	18
4.4.3 Composición (o composición fuerte).....	19
4.4.4 Agregación (o composición débil).....	19
4.4.5 Dependencia.....	20
4.4.6 Realización.....	21
Método abstracto.....	21
Clase abstracta.....	21
Interfaces.....	23
Herencia múltiple.....	25

4.1 Orientación a Objetos

La **programación estructurada** presenta un programa como un guión con principio y final que permite estructuras secuenciales, condicionales e iterativas. Añade además la **modularidad** como recurso para reutilizar software creando módulos que, según el lenguaje de programación, pueden llamarse funciones, procedimientos, subprogramas, subrutinas, etc.

En la **programación orientada a objetos** se entiende un programa como un modelo del universo (trozo del mundo real) que pretende emular. Cada uno de los elementos que intervienen en ese modelo (personas, clientes, cuentas bancarias, etc) son objetos que tienen unas características y unas capacidades y que son capaces de interactuar entre ellos mediante mensajes.



En este dibujo tenemos 4 objetos de la clase **Persona** que son un reflejo de personas reales (Migue, Sonia, Toni y Moi). También tenemos 3 objetos de la clase **Coche** que corresponde a coches concretos (Seat7, Opel2, Renault4). Todos los objetos de la misma clase tienen las mismas características (están hechos con el mismo molde). Los objetos tienen una parte privada, inaccesible dibujada en el centro del objeto con un color distinto. También tienen una parte pública (interfaz) para comunicarse con otros objetos. La comunicación entre objetos se hace mediante mensajes que se envían unos a otros.

4.1.1 Conceptos importantes en la orientación a objetos

- **Clase:** Descripción abstracta de elementos genéricos relevantes en nuestro sistema. Por ejemplo: **Persona**, **Libreta** pueden ser clases necesarias en la aplicación para un banco. La descripción de la clase incluye atributos y métodos.

Una **clase** es una plantilla (un molde) para construir objetos. Cuando se crea un objeto (**instanciación**) se tiene que usar una clase y el objeto pertenecerá a esa clase. De esta manera el compilador comprenderá las características del objeto.

- **Atributos:** Son las características de la clase, sus propiedades, los datos que definen al objeto (variables de clase en **Java**). A la hora de determinar los atributos hay que procurar que sean lo más genéricos posible. Atributos de la clase **Persona** podrían ser **nombre**, **edad** y **dni**.
- **Métodos:** Definen el comportamiento (las cosas que puede hacer) de la clase (métodos en **Java**). Por ejemplo, una de las cosas que puede hacer una persona es **presentarse** ("Hola, me llamo Daniel") o **informarte de su edad** ("Tengo 27 años").

Todas las clases tienen al menos un **método constructor**, que es el que se usa para generar un objeto de esa clase. El constructor tiene unas reglas muy precisas:

- ◆ Debe ser público.
- ◆ Su nombre coincide exactamente con el de la clase.
- ◆ No devuelve ningún valor (ni siquiera **void**).

Nota: Al crear la clase en **Java**, por defecto y sin ninguna acción por nuestra parte, existirá un constructor de la clase sin argumentos (**constructor por defecto**). En el momento que creamos un constructor explícito, dejará de existir el **constructor por defecto**, salvo que lo escribamos explícitamente.

- **Objetos** (o **instancias**): Elementos reales (concretos) del sistema que se crean usando una clase como molde. Usamos la **clase Persona** para crear (instanciar) el **objeto Daniel**. Nuestro sistema no trabajará con las clases directamente, sólo trabaja con objetos. Las clases sólo se usan para crear los objetos.
- **Mensajes:** Los objetos se comunican unos con otros mediante mensajes. En realidad un mensaje es la llamada a uno de los métodos públicos de un objeto. Con el mensaje nos dirigimos a un objeto y le pedimos que realice una de las tareas que sabe hacer.

Al crear una **Libreta** necesitamos el nombre del titular, así que mandamos un mensaje a un objeto de la clase **Persona** para obtener su **nombre**.

- **Interfaz:** La interfaz de un objeto es la parte que se ve desde fuera para comunicarnos con él (es como la ventanilla de una oficina). La interfaz será el conjunto de métodos públicos que permiten pedirle que haga tareas. En los dibujos que acompañan este texto, la interfaz sería el borde exterior marcado en color distinto.

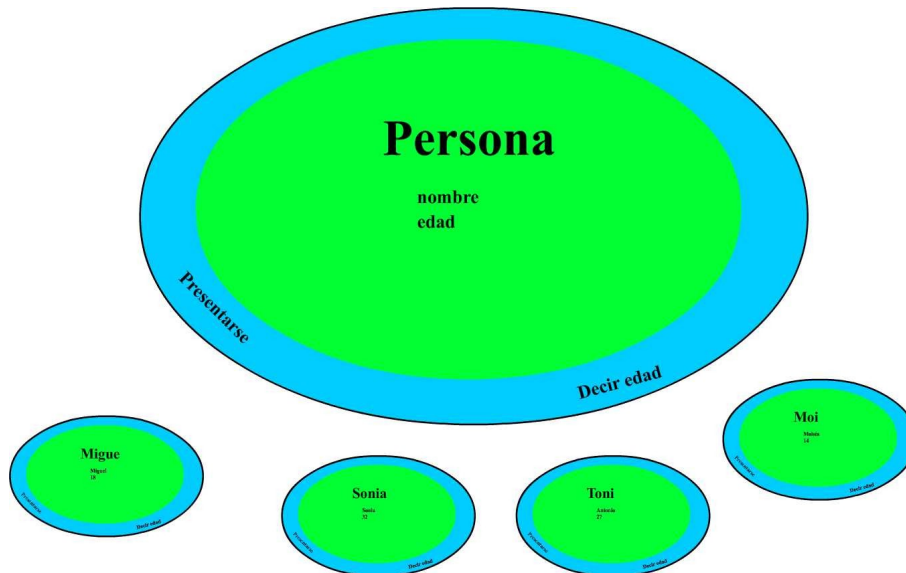
Ejemplo: Clase, atributos, métodos y objetos.

Clase: **Persona**

Atributos de persona: **nombre, edad**

Métodos de persona: **presentarse, decir edad**

Objetos o instancias: **Migue, Sonia,...**



Ejemplo: Definición de la clase Persona usando el lenguaje Java:

```
public class Persona {
    //atributos
    private String nombre;
    private int edad;

    //MÉTODOS
    //constructor
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public void presentar() {
        System.out.println("Hola, mi nombre es " + nombre + ".");
    }

    public void comunicarEdad() {
        System.out.println("Tengo " + edad + " años.");
    }
}
```

Nota: En este ejemplo tenemos los atributos privados (**private**) y los métodos, incluido el método constructor, públicos. Por tanto, el **interface** de esta clase son exclusivamente esos tres métodos. Dicho de otro modo, lo único que se puede usar desde fuera de esta clase son esos tres métodos. No siempre será así, puede haber atributos públicos o privados y métodos públicos o privados. El interface estará formado por los miembros públicos (ya sean atributos o métodos).

4.1.2 Características de un modelo orientado a objetos

ABSTRACCIÓN

Permite describir las características esenciales de los objetos (en clases) sin entrar en las particularidades de objetos concretos. La abstracción permite que la clase se ajuste a todos los objetos que necesitemos. Por ejemplo, al definir la clase **Persona** tenemos que cuidar de no incluir atributos que sólo tienen sentido en una persona concreta o para un subgrupo de personas. Si usamos un atributo como **número de la Seguridad Social** la clase dejaría de ser lo suficientemente general, puesto que ese es un dato que sólo tiene sentido para un subgrupo de personas: aquellas que sean empleados. La abstracción no es exclusiva de la programación orientada a objetos, ya existía en la programación estructurada.

MODULARIDAD

Los métodos se descomponen en unidades pequeñas que se comunican entre sí. Es lo que se conoce como programación modular que también se aplicaba antes de la orientación a objetos.

ENCAPSULACIÓN

Consiste en ocultar los detalles internos de la clase, quedando inaccesibles desde el exterior. Solamente deben escapar a la ocultación los métodos que permiten la interacción con el objeto (lo que se llama **interfaz**). Por ejemplo, en la clase **Persona** debemos definir el atributo **edad** como privado (**private**) para que no pueda ser accedido desde el exterior. Al mismo tiempo, si es necesario, habrá un método público (**public**) **getEdad**, con el que se podrá conocer la edad desde el exterior.

Si aplicamos esta idea de la encapsulación, el objeto se ve como una cápsula cerrada a cuyo interior no se puede acceder, pero en su superficie está la “ventanilla” (**interfaz**) formada por los métodos públicos.

Un ciudadano puede ir a la ventanilla del ayuntamiento (interfaz) y pedir un certificado (método público), pero no puede usar el PC del ayuntamiento para imprimirlo (método privado).

Ejemplo: Ventajas de la encapsulación.

Con la clase **Motorizado** tratamos de representar vehículos con motor. Observa que los atributos de la clase se han declarado como privados de manera que no pueden ser modificados desde fuera de la clase.

```
public class Motorizado extends Vehiculo {
    private int cilindrada;
    private float combustible;
    private int capacidad;
    private float consumo; /* en litros cada 100 km */

    .. . . . . .
    public void setCombustible(float combustible) throws DesbordaCombustException {
        if (combustible >= 0 && combustible <= capacidad) {
            this.combustible = combustible;
        } else {
            throw new DesbordaCombustException();
        }
    }
    .. . . . . .
}
```

El método **setCombustible()** es un método público y permite modificar el valor del atributo **combustible** desde fuera de la clase. Si nos fijamos en el código de este método observamos que antes de aceptar el valor que se le pasa como argumento, hace una comprobación para que no indique una cantidad de combustible inferior a cero ni superior a la capacidad máxima de combustible.

Si se usa la encapsulación correctamente, todas las modificaciones del atributo **combustible** se harán a través de este método con lo que nos aseguramos de que siempre se controlan sus límites superior e inferior sin tener que reescribir la condición que lo comprueba.

Al haber definido el atributo combustible como privado, podemos estar seguros que todas las modificaciones de su valor desde el exterior se harán mediante el método **setCombustible()**. No obstante, aún es posible modificarlo directamente desde el interior de la clase. Si queremos ser estrictos en este control ya será tarea del programador.

Ejemplo NO aconsejado: Acceso directo desde el interior

```
public void repostar(float cantidad) {  
    this.combustible = this.combustible + cantidad;  
}
```

En este ejemplo, el método **repostar()** actúa directamente sobre el atributo **combustible**. A pesar de que el atributo es privado, este método puede hacerlo porque pertenece a la misma clase.

Sin embargo, no se recomienda hacerlo así (al menos en este ejemplo) porque no estamos controlando un valor fuera de rango para el combustible y si queremos controlar ese rango tendremos que hacer la comprobación en todos y cada uno de los métodos que modifiquen directamente el atributo.

Ejemplo aconsejado: Acceso desde el interior usando el método set

```
public void repostar(float cantidad) throws DesbordaCombustException {  
    this.setCombustible(this.getCombustible() + cantidad);  
}
```

En este otro ejemplo, el método **repostar()** no actúa directamente sobre el atributo combustible. Delega en el método **setCombustible()** con lo que puede aprovecharse de la tarea de control que hace ese método.

HERENCIA (JERARQUÍA)

Nos permite definir nuevas clases en base a clases ya conocidas. Esas nuevas clases (clases derivadas, hijas o subclases) heredan las características y los comportamientos de las clases en que se basan (clases base, padres o superclases). Además de lo heredado, las clases hijas incluirán más características y comportamientos. Por ejemplo, a partir de la clase **Persona** podemos definir las clases hijas **Cliente** y **Empleado**. Si las definimos así no será necesario repetir los atributos y métodos que heredan de **Persona** y además podremos añadir aquello que sea específico de las nuevas clases. **Empleado** podrá tener el atributo **número de Seguridad Social** y **Cliente** puede tener el método **dar de alta**.

Ejemplo: La clase Motorizado hereda de la clase Vehiculo.

Disponemos de la clase **Vehiculo** con atributos generales para cualquier vehículo (número de ruedas, kilómetros recorridos, etc) y algunos métodos como **circula(distancia)** que suma la distancia recorrida al kilometraje previo para mantener el cuentakilómetros.

```
public class Vehiculo {
    private float maxVelocidad;
    private boolean requiereLicencia;
    private int numPasajeros;
    private int nRuedas;
    private int kilometraje;
    private Persona propietario;
    -----
    public void circula (int distancia) {
        this.setKilometraje(this.getKilometraje()+distancia);
    }
    -----
}
```

La clase **Motorizado** hace referencia a vehículos con motor que son, por tanto, vehículos. La definimos como heredera de la clase **Vehiculo** (**extends Vehiculo**) lo que hace que ya posea todos los atributos (**nRuedas**, **kilometraje**, ...) y métodos (**circula()**, ...) de la superclase.

La subclase **Motorizado** tiene atributos adicionales (**cilindrada**, **combustible**) que son propios de ella y no deben aparecer en la superclase **Vehiculo** ya que existen vehículos que no los necesitan, por ejemplo, una bicicleta. La subclase **Motorizado** tiene también un método adicional (**repostar()**).

Además vemos que el método **circula()**, que ya existía en **Vehiculo**, es sobrescrito (**@Override**) en la subclase **Motorizado**, porque la acción de circular en estos vehículos implica mas cosas (gasta combustible y para recorrer una distancia necesita disponer de bastante combustible).

```
public class Motorizado extends Vehiculo {
    private int cilindrada;
    private float combustible;
    private int capacidad;
    private float consumo; /* en litros cada 100 km */
    -----
    public void repostar(float cantidad) throws DesbordaCombustException {
        this.setCombustible(this.getCombustible() + cantidad);
    }

    @Override
    public void circula(int distancia) {
        float combustibleNecesario = (distancia / 100) * this.consumo;
        if (this.combustible >= combustibleNecesario) {
            super.circula(distancia);
            try {
                this.setCombustible(this.getCombustible() - combustibleNecesario);
            } catch (DesbordaCombustException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Nota: La palabra reservada **super** hace referencia a la superclase. Así, **super()** sería una llamada al constructor de la superclase y **super.circula()** llama al método **circula()** de la superclase.

POLIMORFISMO

El término **polimorfismo** se refiere a varios comportamientos bajo el mismo nombre. Se trata de que cada objeto responde de forma distinta al mismo mensaje. Es decir, dos objetos pertenecientes a clases distintas pueden tener métodos con el mismo nombre y cada una de ellas reaccionará a esa llamada según su naturaleza. Sólo hablamos de polimorfismo cuando esto ocurre con clases de la misma familia (herencia).

Ejemplo: Polimorfismo.

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

    .....

    public void presentarse() {
        System.out.println("Hola, mi nombre es " + this.getNombre() + ".");
    }

    .....
}
```

Veamos la superclase **Persona** y subclase **Empleado**. Ambas tienen el método **presentarse()**, pero **Empleado** no usa el método heredado de la superclase, sino que sobrescribe su propio método.

```
public class Empleado extends Persona {
    private int numSS;

    .....

    @Override
    public void presentarse() {
        System.out.println("Buenos días. Yo soy Don "+this.getNombre());
    }

}
```

La clase **Pruebas** no forma parte del sistema, sólo la usamos para probar las clases **Persona** y **Empleado**. Creamos dos objetos de la clase **Persona** (**P1** y **P2**) e instanciamos cada uno de ellos con un constructor distinto (esto es posible por su relación de herencia/jerarquía).

```
public class Pruebas {
    public static void main(String args[]) {
        Persona P1 = new Persona("Pepe", 15);
        Empleado P2 = new Empleado("Rodrigo", 30, 111);
        P1.presentarse();
        P2.presentarse();
    }
}
```

El resultado de la ejecución es el siguiente:

```
Hola, Mi nombre es Pepe.
Buenos días. Yo soy Don Rodrigo
```

Son 2 objetos de la misma clase (**Persona**) y usamos el mismo método con ambos (**presentarse()**), sin embargo obtenemos dos resultados distintos. Eso es **polimorfismo**.

Nota: De igual forma, el comportamiento para el método **circula()** será distinto para un objeto definido con el constructor de **Vehiculo** y un objeto definido con el constructor de **Motorizado**.

Nota importante: En el código que acompaña a este tema, en la clase **ProbandoPolimorfismo.java** puedes observar otros ejemplos y usos del polimorfismo.

4.2 UML

UML (Unified Modeling Language - Lenguaje de Modelado Unificado) es un lenguaje basado en diagramas para representar modelos de la realidad siguiendo el paradigma de la orientación a objetos. **UML** define 13 tipos de diagramas divididos en tres categorías:

- **Diagramas de estructura** (parte estática del modelo): Se centran en los elementos que deben existir en el modelo.
 - ◆ **Diagramas de clases:** Muestran las clases del sistema y relaciones entre ellas.
 - ◆ **Diagramas de objetos:** Representan objetos y sus relaciones. Muestra una serie de objetos y sus relaciones en un momento particular de la ejecución del sistema. Son útiles para la comprensión de los diagramas de clases.
 - ◆ **Diagramas de paquetes:** Reflejan la organización de paquetes y sus elementos. El uso más común es organizar diagramas de clases y diagramas de casos de uso, aunque no se limita a eso.
 - ◆ **Diagramas de despliegue:** Especifica el hardware físico sobre el que el sistema se ejecutará y como el software se despliega en ese hardware. Está compuesto de **nodos** (unidad material capaz de recibir y ejecutar software). Los vínculos entre nodos también pueden aparecer. Los nodos contienen **artefactos** (ejecutables, bibliotecas, scripts).
 - ◆ **Diagramas de estructuras compuestas.**
 - ◆ **Diagramas de componentes.**
- **Diagramas de comportamiento** (parte dinámicas del modelo): Se centran en lo que debe suceder en el sistema.
 - ◆ **Diagramas de estado:** Para analizar los cambios de estado de los objetos. Muestra los estados, eventos, transiciones y actividades de los objetos. Se suele usar un círculo relleno para el estado inicial, un círculo relleno dentro de otro círculo para el estado final, rectángulos para el resto de estados y flechas para las transiciones.
 - ◆ **Diagramas de actividad:** Se utiliza para mostrar la secuencia de actividades. Muestran el flujo detallando las decisiones que surgen.
 - ◆ **Diagramas de casos de uso:** Se utiliza para entender el uso del sistema. Muestra un conjunto de actores, las acciones (casos de uso) que realizan y las relaciones entre ellos.
 - ◆ **Diagramas de interacción:**
 - ◆ **Diagramas de secuencia:** Son una representación temporal de los objetos y sus relaciones. Enfatiza la interacción entre objetos y los mensajes que intercambian junto con el orden temporal.
 - ◆ **Diagrama resumen de interacción.**
 - ◆ **Diagrama de comunicación.**
 - ◆ **Diagrama de tiempo.**

4.2.1 Diagramas de clases

Un diagrama de clases describe la estructura de un sistema mostrando sus clases y las asociaciones entre ellas. Sirve para visualizar las relaciones entre las clases que componen el sistema.

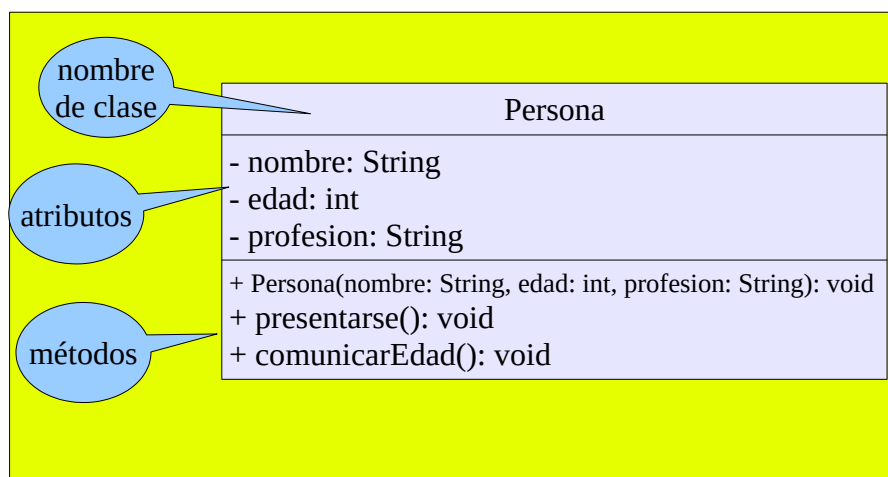
Nota: Un diagrama de clases es muy similar a un diagrama entidad/relación, con la salvedad de que una entidad no tiene capacidad de acción (métodos). De hecho, existe una alternativa para dibujar los diagramas entidad/relación con la notación **UML**, como hemos podido comprobar con las aplicaciones gráficas para modelar bases de datos.

Un diagrama de clases está compuesto por: clases y relaciones entre clases.

4.3 Clases

La clase es una unidad básica en un diseño orientado a objetos. A través de ella podemos representar los elementos del sistema: alumno, vehículo, asignatura, artículo, ... En **UML** una clase se representa con un rectángulo con tres divisiones: en la parte superior aparece el nombre de la clase, en la parte central los atributos y en la parte inferior los métodos.

Ejemplo UML correspondiente a la clase Persona:



ATRIBUTOS

Los **atributos** representan propiedades, características que se encuentran en todas las instancias de la clase. La definición completa de un atributo incluye:

- **Nombre:** del atributo.
- **Tipo:** Los tipos básicos en **UML** son **Integer**, **String** y **Boolean**, aunque también podríamos usar los tipos de cualquier lenguaje de programación.
- **Valor por defecto:** (si fuera necesario).
- **Visibilidad:** Informa desde donde es visible el atributo y puede ser:
 - ◆ **private:** privado. Se ve sólo desde dentro de la propia clase. Se representa con el signo menos (-).
 - ◆ **public:** público. Se ve desde todos lados: desde dentro y fuera de la clase. Se representa con el signo más (+).
 - ◆ **protected:** protegido. Se ve desde dentro de la clase, desde las subclases que deriven de esta (aunque no estén en su paquete) y desde las clases que se encuentren en el mismo paquete. No se ve desde otros paquetes. Se representa con el signo almohadilla (#).
 - ◆ **package:** paquete. Se ve desde todas las clases que pertenezcan a su mismo paquete. Se representa con el signo virgullilla (~). En **Java** la visibilidad por defecto es **package**, no se usa la palabra **package**.

Nota Importante: Al declararlos como **protegidos**, los atributos serán accesibles, no solo desde las subclases, también desde las clases que está en el mismo paquete. CUIDADO CON ESTO, porque si hacemos un mal uso ponemos en peligro la **encapsulación**.

MÉTODOS

Un **método** u operación es la implementación de un servicio de la clase. Definen el comportamiento de los objetos de la clase. La definición de un método incluye:

- **Nombre:** del método.
- **Argumentos:** son los datos de entrada al método. De cada uno de ellos indicaremos nombre y tipo.
- **Tipo de retorno:** tipo de dato que debe devolver el método.
- **Visibilidad:** Informa desde donde es visible el método y puede tomar los mismo valores que la visibilidad de un atributo (privado, público, protegido y paquete)

Nota: **UML** permite omitir atributos y métodos. Esto puede ser necesario cuando tenemos diagramas extensos que incluyen muchas clases. En ese caso cada clase se representaría con un rectángulo con su nombre.

Resumen de los modificadores de visibilidad:

Modificador	desde la clase	desde el paquete	desde subclase	clases ajenas a la jerarquía y al paquete
public	Si	Si	Si	Si
protected	Si	Si	Si	No
package	Si	Si	No	No
private	Si	No	No	No

4.3.1 Representación de clases con ArgoUML

Es una herramienta de modelado **UML** de código abierto, construida en **Java** y puede ejecutarse en cualquier sistema con la máquina virtual **Java**. Desde <http://argouml.tigris.org/> se descarga y tras descomprimir se debe ejecutar **argouml.sh** (**Linux**) o **argouml.bat** (**Windows**). También la puedes descargar desde la **Moodle** de este curso.

Para descomprimir:

```
cd /opt
sudo tar -xzf ruta/nombreFichero.tar.gz
```

Para ejecutar:

```
sh /opt/argouml-0.34/argouml.sh
```

Al abrir la aplicación se distinguen cuatro zonas:

- **Explorador:** arriba izquierda. Muestra los diagramas y sus elementos en árbol.
- **Diseño:** arriba derecha. Donde se dibujan los elementos.
- **Críticas:** abajo izquierda. Muestra consejos y ayudas a los diseños.
- **Propiedades:** abajo derecha. Muestra las propiedades del elemento seleccionado.

En la ventana de diseño dispones de una barra para crear paquetes, clases y los distintos tipos de relaciones. Al pasar el ratón por cada icono aparece el rótulo de ayuda. Al comenzar tienes un paquete (modelo sin título) en el explorador. Lo primero que debes hacer es cambiarle el nombre.

CREAR UNA CLASE

Seleccionamos el icono de la clase (triple rectángulo) y hacemos clic en la ventana de diseño. Aparecerá el dibujo de una clase **UML**. A partir de ahí podemos:

- escribir el nombre de la clase en el rectángulo superior,
- añadir atributos en el rectángulo central,
- añadir métodos en el rectángulo inferior.

En la ventana de propiedades podemos ver y modificar las características del elemento seleccionado: clase, atributo, método, ... Para todos ellos debemos indicar la visibilidad: **público**, **privado**, **protegido** o **paquete**.

Nota: Para trabajar con tipos de datos **Java** hay que incluirlos: **Archivo/Propiedades/Perfiles**.

Nota: Por defecto el diagrama no muestra la visibilidad. Para que lo haga vamos a **Archivo/Propiedades/Notaciones/Mostrar visibilidad**.

Para las clases también debemos seleccionar los modificadores adecuados:

- **isRoot:** Es una clase raíz, no tiene antecesores.
- **isLeaf:** Es una clase hoja, no podrá tener descendientes.
- **isAbstract:** Es una clase abstracta.
- **isActive:** Es una clase activa (programación concurrente).

Para los métodos, en el panel de propiedades establecemos los parámetros y el tipo de retorno.

Nota: Cuando se borra una clase desaparece del diagrama, pero sigue perteneciendo al modelo. Tendrás que borrarla también del explorador.

CRÍTICAS

Si el diseño que estás haciendo no cumple con las reglas **UML** aparecerán críticas y sugerencias en la ventana inferior izquierda. Haciendo doble clic sobre cada una de las críticas obtendrás una descripción detallada de la crítica y sugerencias para solucionarla.

CONFIGURACIÓN

En el menú **Editar/Configuración** tienes acceso a opciones que te permiten personalizar el programa para, por ejemplo, conseguir que se muestre la visibilidad de métodos y atributos o que se muestren u oculten las puntas de flecha.

GENERACIÓN DEL CÓDIGO

En **ArgoUML** podemos ver el código para cada una de las clases en la pestaña "**código fuente**", donde se puede elegir entre varios lenguajes de programación. También podemos generar los ficheros correspondientes a todas las clases desde el menú **Generar/Generar todas las clases** y seleccionando el lenguaje o lenguajes deseados.

EXPORTAR LOS DIAGRAMAS**ARCHIVO/GUARDAR LOS GRÁFICOS****INGENIERÍA INVERSA**

La ingeniería inversa es el proceso contrario al de generación de código, es decir, obtener el modelo **UML** a partir del código en un lenguaje de programación. También se habla de ingeniería inversa cuando se trata de descubrir el código fuente a partir de un ejecutable, pero en este tema nos centramos en la primera definición.

En **ArgoUML**, nos vamos a:

ARCHIVO/IMPORTAR DESDE CÓDIGO

y tras seleccionar la carpeta donde se encuentra el código fuente y establecer las selecciones que nos interesen, pulsamos **Abrir**. Comenzará el proceso de generación que debe finalizar con un diagrama de clases.

Nota: El proceso de ingeniería inversa es muy costoso y puede tardar en completarse.

4.4 Relaciones entre clases

En el mundo real los objetos de distintas clases están relacionados entre sí. Nuestro modelo de la realidad tiene que permitir describir esas relaciones y contempla hasta 6 tipos distintos de ellas:


- Herencia.
- Asociación.
- Composición.
- Agregación.
- Dependencia.
- Realización.

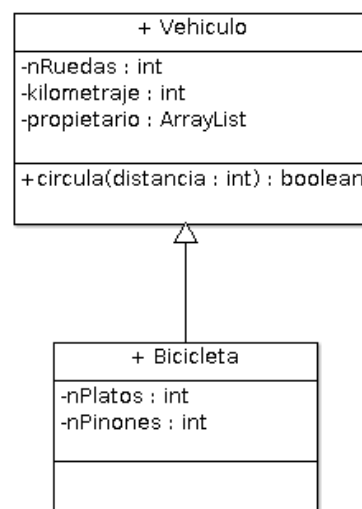
4.4.1 Herencia

La herencia asocia clases con atributos y operaciones comunes. Las superclases contienen los atributos y operaciones más generales (**generalización**) y las subclases son más específicas (**especialización**). La herencia se representa con una flecha con punta en triángulo hueco para diferenciarlas de la asociación (que tiene una flecha con punta en ángulo). El extremo de la flecha en la herencia apunta a la superclase.

Una clase hoja (en inglés *leaf*) es una clase que no puede ser heredada por otras. En **Java** se indica mediante la palabra **final** en la definición de clase:

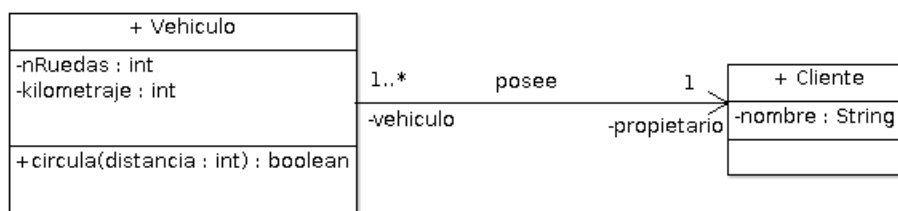
```
public final class Bicicleta {...}
```

Para establecer herencia en **ArgoUML**: Usamos el icono de “**Crear generalización**”  (flecha de punta hueca) en la parte superior del área de diseño.



4.4.2 Asociación

La asociación se indica con una línea que une las clases asociadas. Sobre la línea se escribe el nombre de la asociación y, si fuera necesario, el papel que juega cada una de las clases en la relación. La asociación es el mismo concepto que la relación en el **modelo E/R**. A través de la asociación podemos enlazar a los **Alumnos** con sus **Asignaturas**, a los **Clientes** con los **Artículos**, etc.



MULTIPLICIDAD

Con la asociación aparece el concepto de **multiplicidad**, es decir, la cantidad de objetos de una clase que se asocian con cada objeto de la otra clase (es lo que en el **modelo E/R** se llama cardinalidad). La multiplicidad se expresa indicando en ambos extremos de la línea el valor o la pareja de valores mínimo y máximo.

Notación	Multiplicidad
0..1	Cero o una vez
1	Una y sólo una vez
*	De cero a varias veces
1..*	De una a varias veces
3..7	Entre 3 y 7 veces
4	Exactamente 4 veces

NAVEGABILIDAD

Con la asociación aparece también el concepto de **navegabilidad**. Puede haber navegabilidad en una sola dirección o en ambas y se representa con puntas de flecha en ángulo (para distinguirlo de las puntas de flecha en triángulo que se usan para la herencia).

La navegabilidad indica que la clase origen tiene constancia de la existencia de la otra. Si ambas tienen constancia mutua de la existencia de la otra, hay navegabilidad en ambos sentidos y se dibujan puntas de flecha en los dos extremos.

Nota: Cuando la navegabilidad es bidireccional, **UML** permite dos formas de expresarlo: con flechas en los dos extremos o sin flechas en ningún extremo. Puedes cambiar de una a otra forma en

EDITAR/CONFIGURACIÓN.../APARIENCIA DEL DIAGRAMA/OCULTAR LAS PUNTAS DE FLECHAS...

Cuando se genere el código, en la clase origen habrá atributos que contienen objetos de la clase destino. Dependiendo de la multiplicidad, esos atributos serán un objeto o un conjunto de objetos (array, vector, colección, etc.), pero es importante que quede claro que el concepto de navegabilidad es independiente del de multiplicidad.

En el ejemplo anterior el atributo **propietario** aparecerá en la clase **Vehiculo** como consecuencia de que la clase **Cliente** es **navegable** en esa asociación.

Sin embargo, en la clase **Cliente** NO aparecerá un atributo **vehiculo**, porque la relación no es navegable hacia el otro lado.

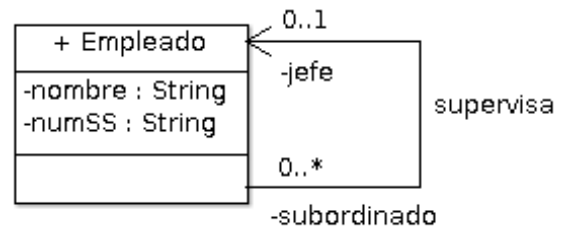
```
public class Vehiculo {
    private int nRuedas;
    private int kilometraje;
    private Cliente propietario;

    public boolean circula(int distancia) {
        return false;
    }
}
```

Nota: A veces las herramientas para crear diagramas **UML** no muestran el atributo que hace que una clase sea navegable desde otra (porque se entiende que al dibujarse la flecha de navegabilidad ya se está diciendo que ese atributo existe). No obstante, al generar el código ese atributo debe aparecer.

ASOCIACIÓN REFLEXIVA

Cuando se establece una asociación entre objetos de la misma clase se dice que es una **asociación reflexiva**. En estos casos es muy importante escribir el rol en cada extremo de la relación para que quede claro el significado de la multiplicidad y de la navegabilidad.



En este ejemplo un objeto de la clase **Empleado** se asocia con otro objeto de la clase **Empleado**. En esa asociación uno ejerce de **jefe** del otro y es necesario escribir el rol en los extremos de la línea para entender que cada **empleado** tiene cero o un **jefe** y que cada **jefe** puede supervisar a varios **empleados**. Por otra parte, la navegabilidad nos indica que cada **empleado** será consciente de quien es su **jefe**, esto es, habrá un atributo **jefe** en la clase **Empleado**.

CREAR UNA ASOCIACIÓN EN ARGOUML

Usamos el icono de “Asociación” en la parte superior del área de diseño. Este icono incluye un desplegable con otros dos iconos:

- línea sin puntas para una asociación navegable en ambos sentidos.
- flecha para una asociación navegable en un solo sentido.



Nota: Para cambiar entre línea sin puntas o línea con dos puntas: **Archivo/Propiedades/Apariencia del diagrama/Ocultar las puntas de flechas.**

Al crear la asociación aparecen varios sombreados en los que debemos escribir el nombre de la asociación y los roles. La multiplicidad y la visibilidad de los roles debes modificarla en las propiedades: Al seleccionar la asociación, en **Propiedades/Conexiones** eliges el extremo que quieres modificar y tendrás la opción de cambiar navegabilidad, visibilidad y multiplicidad.

Nota: En la **Moodle** hay un vídeo minitutorial para crear una asociación con roles y cardinalidad.

LOS ATRIBUTOS DE NAVEGABILIDAD EN ARGOUML

Ya hemos visto que, en el código final, la consecuencia de la navegabilidad es un atributo en la clase origen, a través de la cual la clase destino se hace navegable. **ArgoUML** no muestra esos atributos porque da por hecho que existen al mostrar las puntas de flecha de la navegabilidad. Sin embargo, cuando generamos el código **Java**, si que aparecerán esos atributos.

Atención! Cada vez que borramos una asociación del dibujo **tenemos que borrarla también del esquema** en el navegador de la izquierda. Si no lo hacemos así el código **Java** que se genera no es correcto, porque tendrá los atributos de navegabilidad de todas esas asociaciones "ocultas".

4.4.3 Composición (o composición fuerte)

Un objeto puede estar compuesto de otros objetos. En este caso hablamos de una asociación de composición, que asocia a un objeto complejo con los objetos simples que lo componen, es decir, sus componentes. En la composición fuerte los componentes de un objeto no pueden formar parte de otros objetos, por tanto, la **cardinalidad es 1** y la eliminación del objeto compuesto implica la eliminación de los componentes. Se representa con una línea que tiene un **rombo relleno** en el extremo del objeto compuesto.

Nota: La **composición fuerte** es conocida como **composición** a diferencia de la **composición débil** que es conocida como **agregación**.

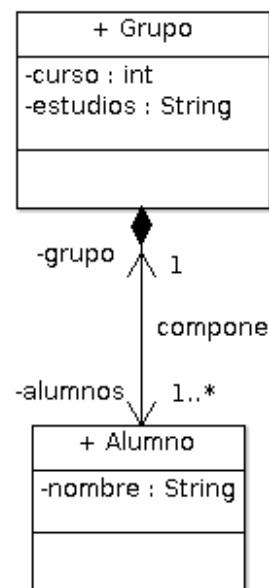
Un ejemplo de composición puede ser el de la imagen, donde un objeto **Grupo** está compuesto por varios objetos **Alumno**. Debemos entender que en este sistema un **Alumno** no puede pertenecer a dos **grupos**.

En la composición también tenemos que tener en cuenta el concepto de navegabilidad.

El código correspondiente es este (en amarillo los atributos que aparecen como consecuencia de la navegabilidad):

```
public class Grupo {
    private int curso;
    private String estudios;
    private ArrayList alumnos;
}
```

```
public class Alumno {
    private String nombre;
    private Grupo grupo;
}
```



4.4.4 Agregación (o composición débil)

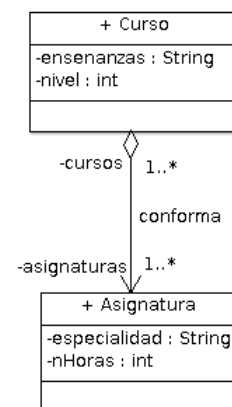
En la agregación (también conocida como composición débil) los componentes pueden ser compartidos por varios compuestos y la destrucción del compuesto no implica la destrucción de los componentes. Se representa con una línea que tiene un **rombo hueco** en el extremo del compuesto. En la agregación puede existir **cualquier cardinalidad**.

En este ejemplo, el **Curso** está formada por **Asignaturas**, pero una **Asignatura** puede seguir existiendo aunque se borre el **Curso** en el que está, porque podría estar en otro **Curso**. Por ejemplo, la asignatura de **FOL** se encuentra en todos los primeros cursos de ciclos.

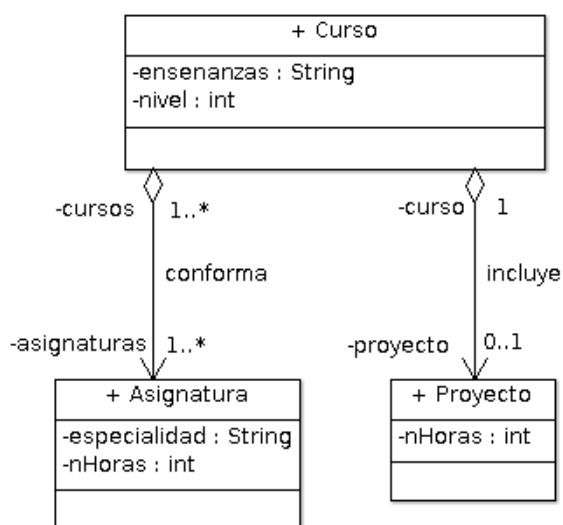
En la agregación también tenemos que tener en cuenta el concepto de navegabilidad. El código correspondiente es:

```
public class Curso {
    private String enseñanzas;
    private int nivel;
    private ArrayList asignaturas;
}
```

```
public class Asignatura {
    private String especialidad;
    private int nHoras;
}
```



En este otro ejemplo vemos que un compuesto puede tener componentes variados. Aquí un **curso** se compone de varias **asignaturas** y, opcionalmente, un **proyecto**.



```

public class Curso {
    private String enseñanzas;
    private int nivel;
    private ArrayList asignaturas;
    private Proyecto proyecto;
}
  
```

```

public class Asignatura {
    private String especialidad;
    private int nHoras;
}
  
```

```

public class Proyecto {
    private int nHoras;
}
  
```

4.4.5 Dependencia

Es la asociación que existe entre dos clases, tales que una usa a la otra. Se representa con una flecha de ángulo y línea discontinua. La flecha apunta a la clase utilizada. Un cambio en la clase utilizada (servidor o **proveedor**) puede afectar a la utilizadora (**cliente**), pero no al contrario.



En este ejemplo, la clase **Ecuacion** usa la clase **Math**, cuando use el método **sqrt()** para calcular la raíz cuadrada. Las modificaciones de **Math** pueden afectar a **Ecuacion**, pero no al revés.

Si el uso que se hace de la clase proveedor es como atributo de la clase cliente, probablemente se trate de una asociación, agregación o composición. La dependencia se da cuando una clase usa a otra, pero no como atributo, sino como un parámetro de método o instanciándola en una variable local o, como en el ejemplo, haciendo uso de sus miembros estáticos.

4.4.6 Realización

Es la asociación entre una **clase interface** y la subclase que la implementa. Se representa igual que la herencia pero con línea discontinua. Para entender el **interface** y la **realización** necesitamos unos conceptos previos sobre abstracción.

MÉTODO ABSTRACTO

Un **método abstracto** es un método sin implementación, es decir, sólo escribimos la cabecera (nombre, parámetros que recibe y tipo que devuelve) pero no escribimos lo que tiene que hacer.

En **UML** los métodos abstractos se representan en cursiva. En **Java** tendría este aspecto:

```
public abstract double area();
```

Observa que, en **Java**, se le antepone la palabra **abstract** y se termina en punto y coma, en lugar de abrir el bloque de implementación del método.

Los métodos abstractos sólo pueden existir en las clases abstractas.

CLASE ABSTRACTA

Una **clase abstracta** es la que se define como tal, es decir, anteponiendo el modificador **abstract** al nombre de la clase:

```
public abstract class PoligonoRegular {
```

La consecuencia inmediata de que una clase sea abstracta es que no se podrán instanciar objetos de esa clase, pero sí podremos extenderla mediante herencia y las subclases ya sí podrán instanciar objetos.

Las clases abstractas pueden (o no) tener métodos abstractos. (En **C++** deberán tener al menos un método abstracto. En **Java** pueden no tener ninguno aunque no tiene mucho sentido).

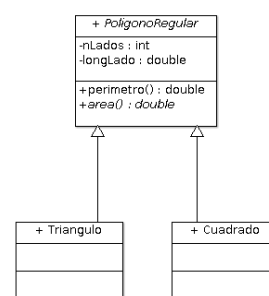
Una **clase concreta** (no abstracta) no puede tener métodos abstractos, lo que implica que si es subclase de una clase abstracta, estará obligada a implementar sus métodos.

Ejemplo: Clase abstracta, subclases concretas.

En el ejemplo de la imagen tenemos una clase llamada **PoligonoRegular** que es abstracta (En **UML** lo sabemos porque está escrita en cursiva). La clase tiene dos atributos y dos métodos:

- **perimetro()**: es un método concreto (no abstracto). El cálculo del perímetro de un polígono es la suma de todos sus lados. Puesto que sabemos el número de lados y la longitud de cada uno de ellos (atributos), se puede crear un método común para calcular el perímetro de todos los polígonos independientemente del número de lados que tengan.

- **area()**: es un método abstracto (en **UML** aparece en cursiva). No tenemos un algoritmo común capaz de calcular el área de todos los polígonos (triángulos, cuadrados, pentágonos, etc).



Nota: En realidad, sí que existe una fórmula para calcular el área de cualquier polígono regular a partir de su número de lados y longitud de lados, pero aquí nos interesa suponer que no.

En **Java**, el código de la clase abstracta **PoligonoRegular** sería el siguiente:

```
public abstract class PoligonoRegular {  
    private int nLados;  
    private double longLado;  
  
    public double perimetro() {  
        return this.getnLados()*this.getLongLado();  
    }  
  
    public abstract double area();  
}
```

Las clases hijas tienen que indicar que extienden a la superclase con la palabra **extends** e implementar **obligatoriamente** sus métodos abstractos:

```
public class Cuadrado extends PoligonoRegular {  
  
    public Cuadrado (double lado) {  
        super();  
        this.setnLados(4);  
        this.setLongLado(lado);  
    }  
  
    /* El área de un cuadrado es el cuadrado de su lado */  
    @Override  
    public double area() {  
        double resultado;  
        resultado = this.getLongLado()*this.getLongLado();  
        return resultado;  
    }  
}
```

INTERFACES

Una **interface** es una clase totalmente **abstracta**, es decir:

- Todos sus métodos son abstractos y públicos. Evidentemente, tampoco se puede instanciar un objeto desde un interface.
- Todos sus atributos serán, por defecto, públicos (**public**), constantes (**final**) y estáticos (**static**), aunque no se pongan estos modificadores. Se recomienda ponerlos para evitar malas interpretaciones.

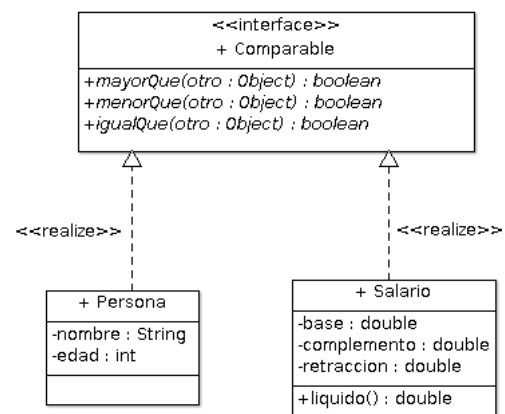
En **UML** los interfaces se reconocen porque llevan la palabra "**interface**" junto al nombre de clase. En **Java**, se usa la palabra **interface** en lugar de la palabra **class** y la palabra **abstract** en los métodos pasa a ser opcional (porque todos los métodos de una interface son abstractos), pero se recomienda ponerla.

Las clases que implementan un interface (**implements**) están obligadas a sobrescribir todos sus métodos.

Ejemplo: Interface y sus implementaciones

En el ejemplo de la imagen tenemos un interface llamado **Comparable** que tiene tres métodos (todos abstractos porque es un interface).

Cualquier clase que implemente esta interface adquiere la capacidad que le otorgan esos métodos que, en este caso, consiste en poder comparar los objetos de la clase entre sí.



La **capacidad de comparar** es un concepto abstracto que permite decir si un objeto es mayor o menor que otro objeto de la misma clase. La comparación es útil en multitud de objetos distintos, así que es un **concepto común**. Sin embargo, la implementación de la comparación es completamente **distinta en objetos de distinta naturaleza**. Cuando se aplica a personas, se dice que una persona es mayor que otra si tiene más edad. Cuando se aplica a salarios, se dice que un salario es mayor que otro si tiene más importe líquido. Cuando se aplica a niveles de alerta, se dice que el nivel rojo es mayor que el nivel amarillo, etc.

Nota: De hecho existe una interface predefinida en **Java** que se llama **Comparable**. Puedes verla en <https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>. Esta interface (que no tiene nada que ver con la de nuestro ejemplo) se aplica a muchas clases, como por ejemplo, a la clase **String** (<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>). Todas las clases que implementan la interface **Comparable** disponen de un método **compareTo()**.

En **Java**, el código de nuestra interface **Comparable** sería el siguiente:

```

public interface Comparable<T> {
    public abstract boolean mayorQue (T otro);
    public abstract boolean menorQue (T otro);
    public abstract boolean igualQue (T otro);
}
  
```

Observaciones:

- Usamos la palabra **interface** en lugar de **class**.
- Todos los métodos son abstractos. No es necesario usar la palabra **abstract** porque en un **interface** siempre será así, pero se recomienda hacerlo.
- La **<T>** que usamos junto al nombre del **interface** hace referencia a un tipo de dato que sólo se conocerá en tiempo de compilación y que no podemos concretar ahora.

¿Qué es **<T>**?: Cuando queramos comparar objetos de la clase **Persona**, el método **mayorQue** tiene que recibir un objeto de la clase **Persona**. Cuando queramos comparar **Salarios**, el método **mayorQue** tiene que recibir un objeto de la clase **Salario**. Usamos la **T** como una referencia “al tipo con el que se defina el interface”.

Aquí tenemos el código de la clase **Persona** que **implementa la interface**:

```
public class Persona implements Comparable<Persona> {  
    private String nombre;  
    private int edad;  
  
    .....  
    @Override  
    public boolean mayorQue(Persona p) {  
        boolean respuesta;  
  
        if (this.getEdad() > p.getEdad()) {  
            respuesta = true;  
        } else {  
            respuesta = false;  
        }  
        return respuesta;  
    }  
    .....  
}
```

Observaciones:

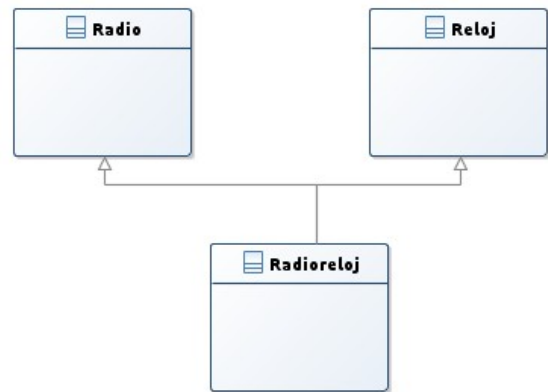
- La clase que implementa un **interface** tiene que indicarlo con la palabra **implements**.
- Al indicar **<Persona>** en el nombre del **interface**, estamos concretando que todas las referencias al tipo genérico **T** deben ser entendidas como el tipo **Persona**.
- Fíjate que en la cabecera del método **mayorQue** el parámetro que recibe es del tipo **Persona**. En la definición del **interface** era del tipo **T**.

HERENCIA MÚLTIPLE

En la programación orientada a objetos existe el concepto de herencia múltiple. Consiste en que una clase puede ser subclase de varias superclases simultáneamente.

Ejemplo: Herencia múltiple entre clases (**Java no la permite**)

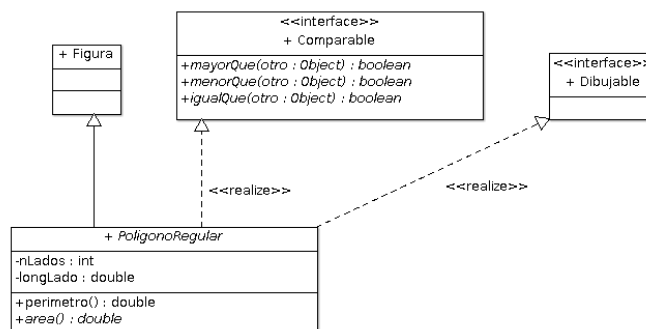
En este ejemplo la clase **Radioreloj** extiende dos superclases: la clase **Radio** y la clase **Reloj**. La consecuencia es que hereda todos los atributos y métodos de ambas. Los objetos de **Radioreloj** tendrán el comportamiento de una **Radio** y también el de un **Reloj**.



Java no permite la herencia múltiple (**C#** tampoco, **C++** si) porque puede ocasionar algunos problemas al heredar dos métodos de igual nombre desde superclases distintas ¿Cuál se debería aplicar?.

Sin embargo **Java** si permite que una clase pueda implementar (**implements**) más de una **interface**.

Ejemplo: Herencia desde una superclase, realización desde varias interfaces



En **Java**, se indicaría así:

```
public abstract class PoligonoRegular extends Figura implements Dibujable, Comparable {
    .. .. .
}
```

Es más, entre **interfaces** (en **Java**) sí puede haber herencia múltiple, porque al tratarse de métodos abstractos, aunque se hereden desde dos superclases no entran en conflicto, ya que sólo tendrán una implementación.

Ejemplo: Herencia múltiple entre interfaces

En **Java**, se indicaría así:

```
public interface Interface3 extends Interface2, Interface1 {
    .. .. .
}
```

