Unidad de trabajo 6. Desarrollo de clases.

RESULTADO DE APRENDIZAJE
4. Desarrolla programas organizados en clases analizando y aplicando los principios de la programación orientada a objetos

Tabla de contenido

1. ESTRUCTURA Y MIEMBROS DE UNA CLASE.

Las clases son el centro de la Programación Orientada a Objetos (OOP - Object Oriented Programming). Algunos de los conceptos más importantes de la POO son los siguientes:

1. Encapsulación. Las clases pueden ser declaradas como públicas (public) y como package (accesibles sólo para otras clases del package). Las variables miembro y los métodos pueden ser public, private, protected y package. De esta forma se puede controlar el acceso y evitar un uso inadecuado.
2. Herencia. Una clase puede derivar de otra (extends), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede añadir nuevas variables y métodos y/o redefinir las variables y métodos heredados.
3. Polimorfismo. Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo. Esto facilita la programación y el mantenimiento del código.

CONCEPTO DE CLASE

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

```
[public] class Classname {

// definición de variables y métodos

...

}
```

donde la palabra public es opcional: si no se pone, la clase tiene la visibilidad por defecto, esto es, sólo es visible para las demás clases del package. Todos los métodos y variables deben ser definidos dentro del bloque {...} de la clase.

Un objeto (en inglés, instance) es un ejemplar concreto de una clase. Las clases son como tipos de variables, mientras que los objetos son como variables concretas de un tipo determinado.

```
Classname unObjeto;

Classname otroObjeto;
```

A continuación, se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de Java deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (extends), hereda todas sus variables y métodos.
3. Java tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.

4. Una clase sólo puede heredar de una única clase (en Java no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de Object. La clase Object es la base de toda la jerarquía de clases de Java.
5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene con extensión *.java. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.
6. Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase.
7. Los métodos de una clase pueden referirse de modo global al objeto de esa clase al que se aplican por medio de la referencia this.
8. Las clases se pueden agrupar en packages, introduciendo una línea al comienzo del fichero (package packageName;). Esta agrupación en packages está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

VARIABLES MIEMBRO

A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está centrada en los datos. Una clase está constituida por unos datos y unos métodos que operan sobre esos datos.

Variables miembro de objeto

Cada objeto, es decir cada ejemplar concreto de la clase, tiene su propia copia de las variables miembro. Las variables miembro de una clase (también llamadas campos) pueden ser de tipos primitivos (boolean, int, long, double, …) o referencias a objetos de otra clase (composición).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de tipos primitivos se inicializan siempre de modo automático, incluso antes de llamar al constructor (false para boolean, el carácter nulo para char y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas también en el constructor.

Las variables miembro pueden también inicializarse explícitamente en la declaración, como las variables locales, por medio de constantes o llamadas a métodos (esta inicialización no está permitida en C++). Por ejemplo,

long nDatos = 100;

Las variables miembro se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Los métodos de objeto se aplican a un objeto concreto poniendo el nombre del objeto y luego el nombre del método, separados por un punto. A este objeto se le llama argumento implícito. Por ejemplo, para calcular el área de un objeto de la clase Circulo llamado c1 se escribirá: c1.area();.

Las variables miembro del argumento implícito se acceden directamente o precedidas por la palabra this y el operador punto.

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: public, private, protected y package (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (public y package), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro.

VARIABLES MIEMBRO DE CLASE (STATIC)

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama variables de clase o variables static. Las variables static se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo PI en la clase Circulo) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como numCirculos en la clase Circulo).

Las variables de clase son lo más parecido que Java tiene a las variables globales de C/C++.

Las variables de clase se crean anteponiendo la palabra static a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, Circulo.numCirculos es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembro static se inicializan con los valores por defecto para los tipos primitivos (false para boolean, el carácter nulo para char y cero para los tipos numéricos), y con null si es una referencia.

Las variables miembro static se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método static o en cuanto se utiliza una variable static de dicha clase. Lo importante es que las variables miembro static se inicializan siempre antes que cualquier objeto de la clase.

VARIABLES FINALES

Una variable de un tipo primitivo declarada como final no puede cambiar su valor a lo largo de la ejecución del programa.

Java permite separar la definición de la inicialización de una variable final. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos.

La variable final así definida es constante (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados final.

Declarar como final un objeto miembro de una clase hace constante la referencia, pero no el propio objeto, que puede ser modificado a través de otra referencia. En Java no es posible hacer que un objeto sea constante.

MÉTODOS (FUNCIONES MIEMBRO)

Los métodos son funciones definidas dentro de una clase. Salvo los métodos static o de clase, se aplican siempre a un objeto de la clase por medio del operador punto (.). Dicho objeto es su argumento implícito. Los métodos pueden además tener otros argumentos explícitos que van entre paréntesis, a continuación del nombre del método.

La primera línea de la definición de un método se llama declaración; el código comprendido entre las llaves {…} es el cuerpo o body del método. Considérese el siguiente método tomado de la clase Circulo:

```
public Circulo elMayor(Circulo c) { // encabezado y comienzo del método

if (this.r>=c.r) // body

return this; // body

else // body

return c; // body

} // final del método
```

El encabezado consta del cualificador de acceso (public, en este caso), del tipo del valor de retorno (Circulo en este ejemplo, void si no tiene), del nombre de la función y de una lista de argumentos explícitos entre paréntesis, separados por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Los métodos tienen visibilidad directa de las variables miembro del objeto que es su argumento implícito, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia this, de modo discrecional (como en el ejemplo anterior con this.r) o si alguna variable local o argumento las oculta.

El valor de retorno puede ser un valor de un tipo primitivo o una referencia. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de interface. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.

Los métodos pueden definir variables locales. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un

valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

BUENAS PRÁCTICAS.

Para clases

- La mayoría de las clases que se crean son públicas.
- Cada fichero .java tendrá solamente una clase pública, con el mismo nombre del fichero.

Para atributos

- La mayoría de los atributos de una clase serán privados.
- Solamente algunas constantes, o casos muy particulares, tendrán otra modificador de acceso.

Para métodos

- Si una clase tiene atributos, seguramente tenga métodos públicos.
- Los métodos privados son interesantes para cálculos auxiliares o parciales (solo se pueden invocar desde la propia clase).

2. CREACIÓN DE CONSTRUCTORES.

Un punto clave de la Programación Orientada Objetos es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. Java no permite que haya variables miembro que no estén inicializadas. Para la inicialización correcta de objetos se usan los constructores.

Un constructor es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del constructor es reservar memoria e inicializar las variables miembro de la clase.

Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase. Su argumento implícito es el objeto que se está creando. Normalmete, una clase tiene varios constructores, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos sobrecargados). Se llama constructor por defecto al constructor que no tiene argumentos. El programa debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

El constructor es tan importante que, si el programador no prepara ningún constructor para una clase, el compilador crea un constructor por defecto, inicializando las variables de los tipos primitivos a su valor por defecto, y los Strings y las demás referencias a objetos a null. Si hace falta, se llama al constructor de la super-clase para que inicialice las variables heredadas.

3. CLASES HEREDADAS.

Todo objeto, de forma directa o indirecta hereda de Object. Esta clase tiene una serie de métodos, entre los que destacan:

- equals: nos permite indicar cuando dos objetos son iguales
- hashCode: nos devuelve un número "único" asociado a una instancia de una clase
- toString: nos devuelve una reperesentación del objeto como una cadena de caracteres.

Una clase que extiende a otra hereda sus atributos y sus métodos (no constructores). Puede añadir atributos y métodos nuevos. Se trata de una asociación *es-un*, ya que la clase Hija es-un(a) (sub)tipo de la clase Base. Por ejemplo un Coche es-un Vehiculo, o un Leon es-un Animal.

Si usamos protected en la clase base, tendremos acceso directo a los atributos. En otro caso, tendremos que acceder vía getters/setters. Hay que tener en cuenta que los constructores no se heredan aunque sean públicos.

```
public class Base {

    private String nombre;
    protected String apellidos;

    //...
}

public class Hija extends Base {

    public void metodo() {
        //this.nombre = "Pepe";  //Imposible acceder. Nos da error
        this.setNombre("Pepe"); //Funciona perfectamente
        this.apellidos = "Perez";
    }

    //...
}
```

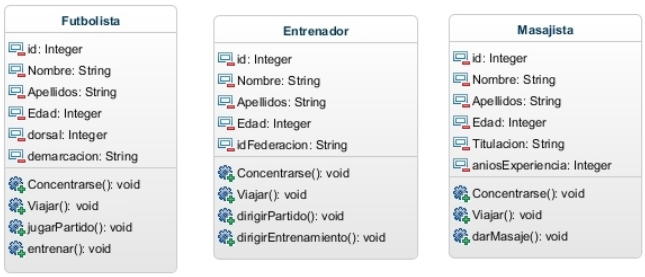
Es un mecanismo que permite definir unas clases a partir de otras heredando todas sus propiedades. El resultado es una nueva clase que extiende la funcionalidad de una clase existente sin tener que reescribir el código asociado a esta última.

La nueva clase, a la que se conoce como **subclase**, puede poseer atributos y métodos que no existan en la clase original. Los objetos de la nueva clase heredan los atributos y los métodos de la clase original, que se denomina **superclase**

La clase A se debe relacionar mediante herencia con la clase B si "A ES-UN B". Por ejemplo, un pájaro es un animal o un coche es un vehículo

En el siguiente ejemplo se muestran 3 clases que modelan el comportamiento que tendrían los diferentes integrantes de la selección española de futbol; tanto los futbolistas como el cuerpo técnico.

De cada uno de ellos vamos a necesitar algunos datos que reflejaremos en los atributos y una serie de acciones que reflejaremos en sus métodos. Estos atributos y métodos los mostramos en el siguiente diagrama de clases:



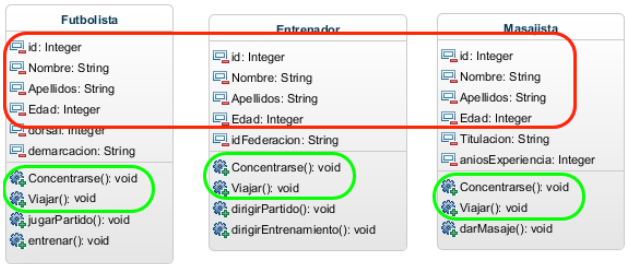
Como se puede observar, en las tres clases tenemos atributos y métodos que son iguales ya que los tres tienen los atributos *id, Nombre, Apellidos y Edad*; y los tres tienen los métodos de *Viajar* y *Concentrarse*

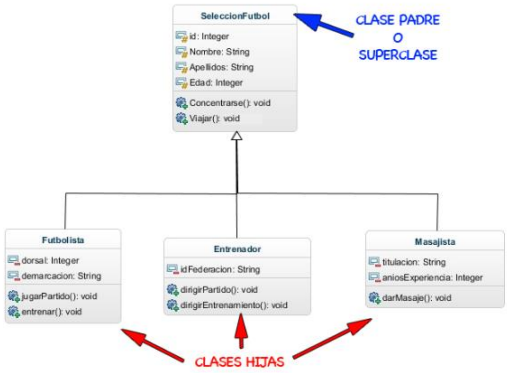
La herencia es un mecanismo que permite definir unas clases a partir de otras heredando todas sus propiedades. Permite reutilizar clases ya existentes.

El resultado es una nueva clase que extiende la funcionalidad de una clase existente sin tener que reescribir el código asociado a esta última.

La nueva clase, a la que se conoce como **subclase**, puede poseer atributos y métodos que no existan en la clase original

Los objetos de la nueva clase heredan los atributos y los métodos de la clase original, que se denomina **superclase**

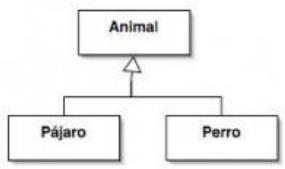




Una mejor solución a este problema será crear una clase con el código que es común a las tres clases que será la Clase Padre o SuperClase y como Clases Hijas o subclases definiremos aquellas que contenga con el código que es especifico.

TIPOS DE HERENCIA

- Herencia Simple: Indica que se pueden definir nuevas clases solamente a partir de una clase inicial



- Herencia de implementación: La implementación de los métodos es heredada. Puede sobreescribirse en las clases derivadas.
- Herencia de interfaz: Sólo se hereda la interfaz, no hay implementación a nivel de clase base.

OPERADOR INSTANCEOF

El operador instanceof nos permite comprobar si un objeto es de una clase concreta.

Por ejemplo, tenemos 3 clases:

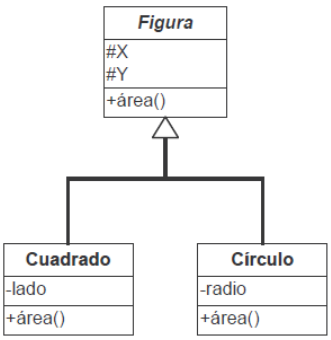
- Empleado (clase padre),
- Comercial (clase hija de Empleado)
- Repartidor (clase hija Empleado).

```
public class EmpleadoApp {

    public static void main(String[] args) {

        Empleado empleados[]=new Empleado[3];
        empleados[0]=new Empleado();
        empleados[1]=new Comercial();
        empleados[2]=new Repartidor();

        for(int i=0;i<empleados.length;i++){
            if(empleados[i] instanceof Empleado){
                System.out.println("El objeto en el indice "+i+" es de la clase Empleado");
            }
            if(empleados[i] instanceof Comercial){
                System.out.println("El objeto en el indice "+i+" es de la clase Comercial");
            }
            if(empleados[i] instanceof Repartidor){
                System.out.println("El objeto en el indice "+i+" es de la clase Repartidor");
            }

        }

    }

}
```

CLASES ABSTRACTAS

Una clase abstracta es una clase que actúa como un depósito de métodos y atributos compartidos para las subclases de nivel inferior y que no tienen instancias directamente. Se usan para agrupar otras clases y para capturar información que es común al grupo.

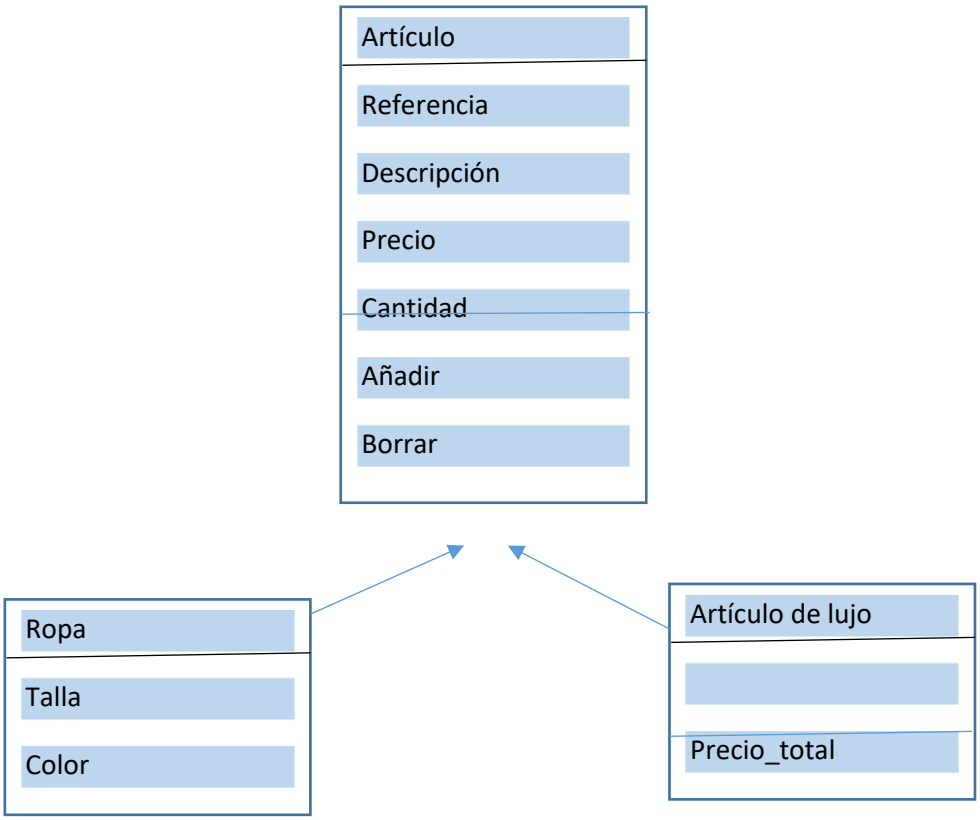
Por ejemplo, Figura es una clase abstracta porque no tiene sentido calcular su área, pero sí la de un cuadrado o un círculo.



No existirán objetos de la clase Figura pero si habrá objetos de la clase cuadrado y de la clase círculo.

ENRIQUECIMIENTO Y SUSTITUCIÓN

Los atributos y los métodos definidos en la superclase, son heredados por la subclases, pero éstas también pueden añadir atributos y modificar métodos. Esto se conoce como enriquecimiento y sustitución respectivamente.



Enriquecimiento

(añade nuevos atributos)

Sustitución

(modifica el método precio_total)

4. ENCAPSULACIÓN Y VISIBILIDAD. INTERFACES DE LAS CLASES.

Se entiende por visibilidad, ámbito o scope de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en una expresión. En Java todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves {}, es decir dentro de un bloque, son visibles y existen dentro de estas llaves.

Por ejemplo, las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque if no serán válidas al finalizar las sentencias correspondientes a dicho if y las variables miembro de una clase (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

MODIFICADORES

DEFAULT:

Si no elegimos ningún modificador, se usa el de por defecto, que sólo puede ser **accedido por clases que están en el mismo paquete**.

PUBLIC:

Este nivel de acceso permite a acceder al elemento **desde cualquier clase**, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento.

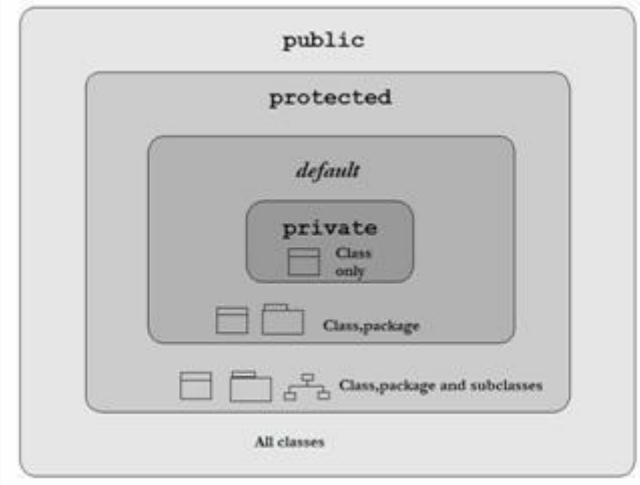
PRIVATE:

Es el modificador más restrictivo y especifica que los elementos que lo utilizan **sólo pueden ser accedidos desde la misma clase en la que se encuentran**. Este modificador sólo puede utilizarse sobre los miembros de una clase y sobre interfaces y clases internas, no sobre clases o interfaces de primer nivel, dado que esto no tendría sentido.

PROTECTED:

Indica que los elementos sólo pueden ser accedidos **desde su mismo paquete y desde cualquier clase que extienda la clase en que se encuentra**, independientemente de si esta se encuentra en el mismo paquete o no. Este modificador, como private, no tiene sentido a nivel de clases o interfaces no internas.

En otras palabras, si determinada clase Hijo hereda el comportamiento de una clase Padre, la clase Hijo tendrá acceso a todos aquellos campos/métodos definidos como *protected* en Padre, pero no aquellos declarados como private en Padre.



Modificadores de acceso

MODIFICADORES DE CLASES Y MÉTODOS: STATIC, FINAL, ABSTRACT.

STATIC:

Sirve para crear miembros que pertenecen a la clase, y no a una instancia de la clase. Esto implica, entre otras cosas, que **no es necesario crear un objeto de la clase para poder acceder a estos atributos y métodos**.

En ocasiones es necesario o conveniente generar elementos que tomen un mismo valor para cualquier número de instancias generadas o bien invocar/llamar métodos sin la necesidad de generar instancias, y es bajo estas dos circunstancias que es empleado el calificador *static*.

Dos aspectos característicos de utilizar el calificador static en un elemento Java son los siguientes:

– **No puede ser generada ninguna instancia** (uso de *new*) de un elemento *static* puesto que solo existe una instancia.

– Todos los elementos definidos dentro de una estructura *static* deben ser *static* ellos mismos, o bien, poseer una instancia ya definida para poder ser invocados.

NOTA: Lo anterior no implica que no puedan ser generadas instancias dentro de un elemento *static*; no es lo mismo llamar/invocar que crear/generar.

FINAL:
Indica que una variable, método o clase no se va a modificar, lo cuál puede ser útil para añadir más semántica, por cuestiones de rendimiento, y para detectar errores.

– Si una **variable** se marca como *final*, **no se podrá asignar un nuevo valor** a la variable.
– Si una **clase** se marca como *final*, **no se podrá extender** la clase.
– Si es un **método** el que se declara como *final*, **no se podrá sobreescribir**.

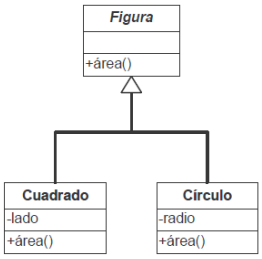
INTERFACES

Un interfaz es una lista de métodos (solamente cabeceras de métodos, sin implementación) que define un comportamiento específico para un conjunto de objetos. Cualquier clase que declare que implementa un determinado interfaz, debe comprometerse a implementar todos y cada uno de los métodos que ese interfaz define. Esa clase, además, pasará a pertenecer a un nuevo tipo de datos extra que es el tipo del interfaz que implementa.

Los interfaces actúan, por tanto, como tipos de clases. Se pueden declarar variables que pertenezcan al tipo del interfaz, se pueden declarar métodos cuyos argumentos sean del tipo del interface, asimismo se pueden declarar métodos cuyo retorno sea el tipo de un interface.

Una clase puede implementar tantos interfaces como desee, pudiendo, por tanto, pertenecer a tantos tipos de datos diferentes como interfaces implemente. En este sentido, los interfaces suplen la carencia de herencia múltiple de Java (y además corrigen o evitan todos los inconvenientes que del uso de ésta se derivan).

En el ejemplo anterior, si no estuviésemos interesados en conocer la posición de una `Figura`, podríamos eliminar por completo su implementación y convertir `Figura` en una interfaz:




```
public interface Figura
{
  public double area ();
}
```

5. LIBRERÍAS DE CLASES.

Al instalar Java (el paquete JDK) en nuestro ordenador, además del compilador y la máquina virtual de Java se instalan una cantidad muy importante de clases y que están a disposición de todos los programadores. Estas clases, junto a otros elementos forman lo que se denomina API (Application Programming Interface) de Java.

La mayoría de los lenguajes orientados a objetos ofrecen a los programadores bibliotecas de clases que facilitan el trabajo con el lenguaje.

La biblioteca estándar de Java se facilita como código cerrado, es decir, como código máquina. No podemos acceder al código fuente. Esto tiene su lógica porque si cada persona accediera al código fuente de los elementos esenciales de Java y los modificara, no habría compatibilidad ni homogeneidad en la forma de escribir programas Java.

Pero, aunque no podamos acceder al código fuente, sí podemos crear objetos de los tipos definidos en la librería e invocar sus métodos. Para ello lo único que hemos de hacer es conocer las clases y la interfaz de los métodos, y esto lo tenemos disponible en la documentación de Java.

6. BIBLIOGRAFÍA

- Aprenda Java como si estuviera en primero García de Jalón, J.
- aprenderaprogramar.com
- OpenWebinars