DESARROLLO DE APLICACIONES WEB

Unidad 11

Utilización avanzada de clases

1r DAW

IES La Mola de Novelda

Departament d'informàtica

Profesor: Joan Carbonell Picó

ÍNDEX

Unidad 11: Utilización Avanzada de Clases

1.- RELACIÓN ENTRE CLASES

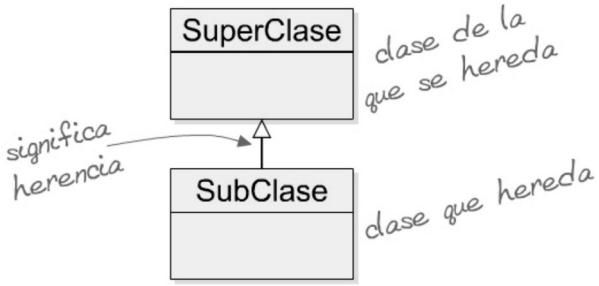
Se pueden distinguir diversos tipos de relaciones entre clases:

- **Clientela**: Cuando una clase utiliza objetos de otra clase (por ejemplo al pasarlos como parámetros a través de un método).
- **Composición**: Cuando alguno de los atributos de una clase es un objeto de otra clase.
- **Anidamiento**: Cuando se definen clases en el interior de otra clase.
- **Herencia**: Cuando una clase comparte determinadas características con otra (clase base), añadiéndole alguna funcionalidad específica (especialización).

2.- HERENCIA

La herencia (también se denomina extensión o generalización) es una de las grandes cualidades de la POO, que permite, al igual que en la vida real, que las características pasen de padres a hijos. Cuando una clase hereda de otra, adquiere sus **atributos y métodos visibles**, permitiendo reutilizar el código y las funcionalidades, que en la clases que heredan se pueden ampliar o extender.

La clase de la que se hereda se denomina clase padre o superclase, y la clase que hereda es conocida como clase hija o subclase.



Una subclase dispone de los miembros heredados de la superclase y, habitualmente, se amplia añadiéndole nuevos atributos y métodos. Esto aumenta su funcionalidad, a la vez que evita la repetición innecesario de código., Por ejemplo, en la API de Java, la mayoría de las clases no se definen desde cero, si no que se construyen heredando de otras, lo cual simplifica su desarrollo.

2.1.-SUPERCLASE

La forma de expresar las clases base (superclase) y las clases derivadas (subclases) es mediante la palabra reservada ***extends***, de la siguiente manera:

```
[modificador] class ClasePadre {
        // Cuerpo de la clase
        ...
}
[modificador] class ClaseHija extends ClasePadre {
        // Cuerpo de la clase
        ...
}
```

- Ejemplo: Supongamos que tenemos la clase **Persona** (nombre, edad y estatura), y necesitamos construir la clase **Empleado**. Un empleado no es mas que una persona (nombre, edad y estatura) con un salario. Generar un proyecto con un paquete donde estarán las dos clases, y después cuando la clase Persona está en un paquete (persona) y la clase Empleado en otro paquete (empleado). En la clase Persona genera el método *mostrarDatos*() que visualizará por pantalla la información de una persona. Comprueba como un objeto empleado cuando se construye a qué constructor llama (en el mismo paquete y en diferente) y si puede visualizar su información como persona. Prueba generar el constructor Empleado con el IDE Intellij y deduce su significado.

El proceso de herencia puede continuar ampliando la biblioteca de clases a partir de las existentes. Por ejemplo, podemos definir a partir de Empleado la clase Jefe, ya que un jefe no es más que un empleado con unas propiedades añadidas.

> Existen lenguajes como C++ que permiten que una clase herede de otras muchas, lo que se conoce como herencia múltiple. **Java** sólo permite **herencia simple**, donde cada clase tiene como padre una única superclase, cosa que no impide que a su vez tenga varias clases hijas.
>
> Es posible, aunque no tiene mucho sentido, dejar la definición de una subclase vacía, con lo cual sería una copia exacta de la superclase de la que hereda.

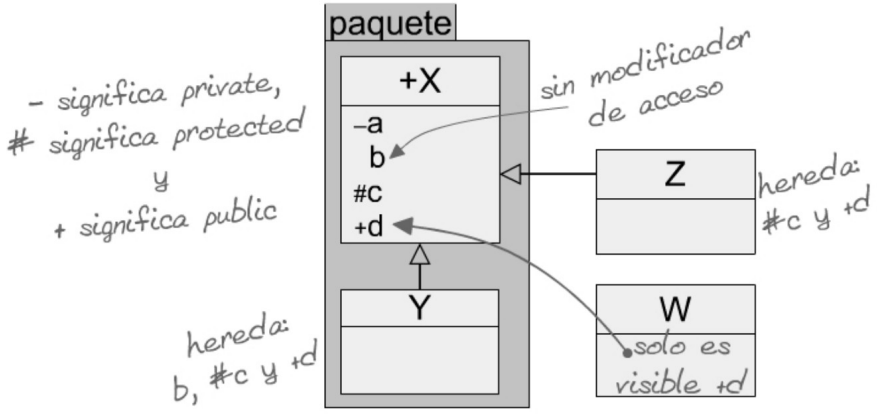
NOTA: Si queremos prohibir que una clase pueda ser extendida (sellar la clase), deberemos añadir el modificador final en la declaración de la clase. De este modo, no se podrá crear una clase que herede de ésta: [modificador] **final** class ClaseFinal { // Cuerpo de la clase}

2.2.-MODIFICADORES DE ACCESO

Cuando una clase hereda de otra, hemos de tener claro que miembros de la clase padre (superclase) se heredan en la clase hija (subclase) a partir de los modificadores que tengan éstos. Por ejemplo, se heredan todos, aunque los miembros que tengan el modificador ***private*** no serán visibles desde la subclase, aunque se pueden acceder a ellos con un método no privado.

A partir de los diferentes tipos de visibilidad que hemos visto hasta ahora (*public*, *private*, por defecto) indicaremos que miembros serán visibles des de una subclase. Pero para tener una mayor flexibilidad, deberemos hacer uso del modificador de acceso ***protected***, el cual está pensado para facilitar la herencia. Funciona de forma muy similar a la visibilidad por defecto, con la diferencia de que los miembros protegidos serán siempre visibles para las clases que heredan, indistintamente de si la superclase y la subclase son vecinas o externas

En la siguiente figura podemos ver los diferentes tipos de visibilidad:



En resumen, un miembro ***protected*** es visible en las clases vecinas, no es visible para las clases externas, pero siempre es visible, indistintamente del paquete, desde una clase hija. La siguiente figura ilustra la visibilidad de cada uno de los modificadores:

	Visible desde...			
	la propia clase	clases vecinas	subclases	clases externas
`private`	✓			
sin modificador	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

La implementación de la clase X anterior sería:

```
public class X {
        private int a;          // invisible fuera de la clase
        int b;                  // visibilidad por defecto: visible en el paquete
        protected int c ;       // visibilidad en el paquete y por las subclases
        public int d;           // visibilidad total
        ...
}
```

El atributo **a** es invisible desde fuera de la clase; el atributo **b** visible sólo desde el mismo paquete, sólo será visible por subclases vecinas; **c** sólo es accesible desde el mismo paquete, pero en caso de herencia, siempre es visible desde las subclases; y por último **d** es visible desde cualquier lugar.

2.3.-REDEFINICIÓN DE MIEMBROS HEREDADOS

Cuando una clase hereda una serie de miembros, en alguna ocasión puede ocurrir que interese modificar el tipo de algún atributo o redefinir un método. Este mecanismo se conoce como ocultación cuando es un **atributo** y sustitución u *overrriding* para un **método**. Consiste en declarar un miembro con igual nombre que uno heredado, lo cual hace que éste sea ocultado o sustituido por el nuevo.

- Ejemplo: En la clase Empleado podemos reescribir el método *mostrarDatos(),* ya que con este método de la superclase no podemos imprimir el salario de una instancia empleado.

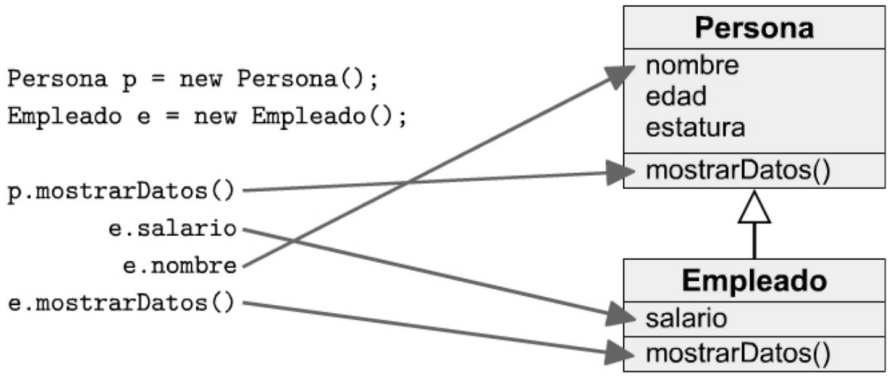
  Redefinir métodos es opcional, y los métodos sustituidos en las subclases se suelen marcar con la anotación ***@Override***, que indica que el método es una sustitución u *overriding* de un método de la superclase. La redefinición del método mostrarDatos() en la subclase **Empleado** será:

```
class Empleado extends Persona {
        double salario:
        @Override
         void mostrarDatos() {
                 System.out.println("Nombre: "+this.getNombre());
                System.out.println("Edad: "+this.getEdad());
                System.out.println("Estatura: "+this.getEstatura());
                System.out.println("Salario: "+this.getSalario());
        }
}
```

- Ejemplo: Como ejemplo de ocultación de atributo podemos indicar que la estatura de una empleado, como longitud, no es un dato relevante para la empresa. Pero si es interesante conocer la estatura como talla del uniforme. En este caso, vamos a redefinir el atributo como una cadena para que contenga la talla del uniforme, con los valores: “XXL”, “XL”, “L”, etc. La redefinición de la clase **Empleado** será:

```
class Empleado extends Persona {
        double salario:
        String estatura;
        @Override
         void mostrarDatos() {
                ...
        }
}
```

En este caso, en la clase Empleado se oculta el atributo **estatura** de tipo *double*, por otro de tipo **String**. En la siguiente figura tenemos un ejemplo de qué miembro es el que se utiliza en cada caso



2.4.-SUPER Y SUPER()

Como hemos indicado en otra unidad, la palabra reservada ***this*** se utiliza para indicar la propia clase. De forma análoga, disponemos de ***super***, que hace referencia a la superclase de la clase donde se utiliza.

- Ejemplo: En el siguiente ejemplo, en **SubClase** se ha redefenido el atributo **b**. Cada vez que se utiliza dicho atributo en **SubClase** estaremos utilizando un **String**, pero si deseamos utilizar el atributo **b** de tipo entero de la superclase desde la subclase, tendremos que escribir **super.b**.

```
class SuperClase{
        int a;
        int b:
}
```

```
class SubClase extends SuperClase {
        String b;
}
```

Algo habitual es tener un método que muestre los datos de una clase. En caso de que la clase haya heredado de otra, dicho método tiene que redefinirse casi obligatoriamente, para mostrar también los datos de la clase hija. Si utilizamos el método ***mostrarDatos()*** de la superclase, dejaremos sin mostrar los nuevos atributos definidos en la subclase. En ***mostrarDatos()*** de la subclase no hace falta copiar el código que muestra los atributos de la superclase; se puede invocar el método ***mostrarDatos()*** de la superclase, utilizando ***super.mostrarDatos()***.

- Ejemplo: En la clase Empleado, quedaría de la siguiente manera:

```
public class Empleado extends Persona {
        double salario;
        …
         @Override
        void mostrarDatos() {
                super.mostrarDatos();                          // método de la superclase
                System.out.println("Salario: "+this.getSalario());     // muestra el atributo propio
         }
}
```

Algo análogo ocurre con los constructores propios de la superclase. Para ellos disponemos del método ***super()***, que invoca un constructor de la superclase. En caso de estar éstos sobrecargados, podemos variar sus parámetros de entrada en número y en tipo. Una restricción a tener en cuenta con ***super()*** es que si lo utilizamos, tiene que ser forzosamente la primera instrucción de un constructor.

- Ejemplo:

```
public class Persona {
        String nombre;
        int edad;
        double estatura;
}
Persona(String nombre,int edad,double estatura){
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
  }
```

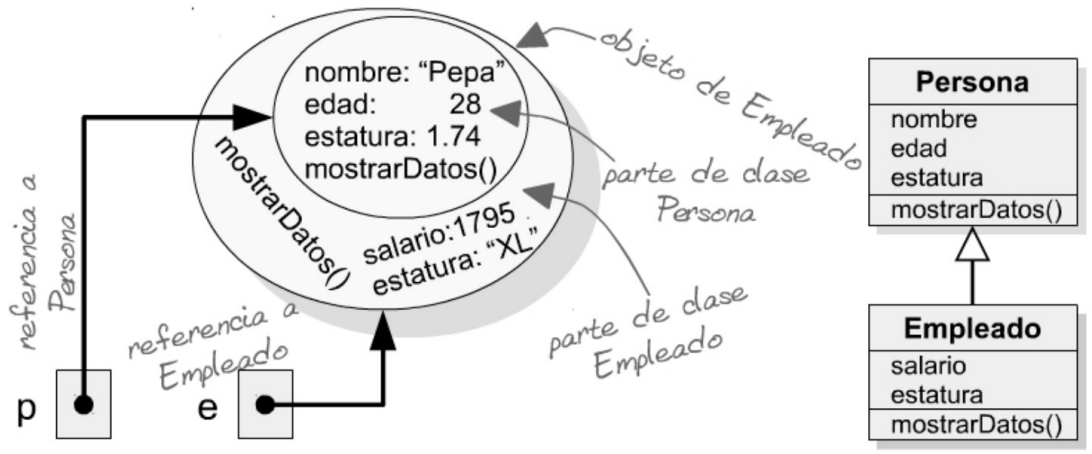
```
public class Empleado extends Persona {
        double salario;
}
public Empleado(String nombre, int edad, double estatura,
double salario) {
        super(nombre, edad, estatura); // constructor de
                                                Persona
        this.salario = salario;
  }
```

2.5.-SELECCIÓN DINÁMICA DE MÉTODOS

Cuando definimos una clase como subclase de otra, los objetos de la subclase son también objetos de la superclase.

- Ejemplo: Un objeto **Empleado** será al mismo tiempo un objeto **Persona**, ya que posee todos los miembros de **Persona**, además de otros específicos de **Empleado**. Por lo tanto, podemos referenciar un objeto **Empleado** usando una variable **Persona**.

```
Empleado e = new Empleado();
Persona p = e;
```



A partir de la anterior figura podemos preguntarnos: ¿Es lo mismo una variable **Empleado** que una variable **Persona** para referenciar a un **Empleado**? No, ya que hay una sutil pero importante diferencia.

1. Los atributos accesibles son los definidos en la clase de variable, por lo que no se produce ocultación:

```
p.estatura; //atributo de Persona de tipo double
e.estatura; // atributo de Empleado de tipo String
```

2. En cambio, con los métodos ocurre lo contrario. Se ejecuta la versión del objeto referenciado, es decir, de la subclase **Empleado**.

```
p.mostrarDatos(); // método Empleado
e.mostrarDatos(); //método Empleado
```

Con este ejemplo, podemos ver que Java nos proporciona una de las herramientas más potentes de las que dispone para ejercer el **POLIMORFISMO**: la selección de métodos en tiempo de ejecución.

- Ejemplo: Supongamos una tercera clase **Cliente** que hereda de **Persona**.

```
public class Cliente extends Persona {
	@Override
	void mostrarDatos() {
		super.mostrarDatos();
		System.out.printf("Soy un cliente");
	}
}
```

Si creamos una variable de tipo **Persona**, con ella podemos referenciar tanto objetos de la clase **Empleado** como de la clase **Cliente**, como de la clase **Persona**. Para todos ellos disponemos del método ***mostrarDatos()***, pero se ejecutará una u otra versión, según el objeto referenciado, que puede cambiar en tiempo de ejecución:

```
Persona p = new Persona();
p.mostrarDatos();
p = new Empleado();
p.mostrarDatos();;
p = new Cliente();
p.mostrarDatos();
```

```
PERSONA
========
Nombre: null
Edad: 0
Estatura: 0.0
PERSONA - EMPLEADO
================
Nombre: null
Edad: 0
Estatura: 0.0
Salario: 0.0
PERSONA - CLIENTE
========
Nombre: null
Edad: 0
Estatura: 0.0
```

2.6.-LA CLASE *OBJECT*

La clase ***Object*** del paquete ***java.lang*** es una clase especial de la que heredan, directa o indirectamente, todas las clases de Java. Es la superclase por excelencia, ya que se sitúa en la cúspide de la estructura de herencias entre clases.

Todas la clases que componen la API de Java descienden de la clase ***Object***. Esta herencia se realiza por defecto sin necesidad de especificar nada.

- Ejemplo: La definición de la clase **Persona** es en realidad una herencia de la clase ***Object***.

```
public class Persona {
	...
}
```

```
public class Persona extends Object {
	...
}
```

Los objetivos de que todas las clases hereden de ***Object*** son:

1. Que todas las clases implementen un conjunto de métodos (en ***Object*** sólo se han definido métodos) que son de uso universal en Java, como realizar comparaciones entre objetos, clonarlos o representar un objeto como una cadena.

2. Poder referenciar cualquier objeto, de cualquier tipo, mediante una variable de tipo ***Object***.

- Ejemplo: Si queremos ver los métodos de ***Object*** que ha heredado la clase **Persona**, escribiremos en un IDE una variable del tipo **Persona**, seguida de una punto. Aparecerán todos los atributos y métodos disponibles: los propios de la clase **Persona** más los heredados de ***Object***.

2.6.1.- Método toString()

Este método devuelve una cadena que representa al objeto desde el que se llama. Su prototipo es:

```
public String toString();
```

La implementación de ***toSting()*** en ***Object*** consiste en devolver el nombre cualificado de la clase a la que pertenece el objeto, seguida de una arroba (@) junto a la referencia del objeto. Para un objeto **Persona** se devuelve algo similar a:

paquete.Persona@2a139a55

Esta implementación por defecto no es útil para representar la mayoría de los objetos, por lo que estamos obligados a realizar un ***overriding*** de ***toString()*** en cada clase (que es la única que conoce cómo será la cadena que representa a sus objetos).

- Ejemplo: Para reimplementar ***toString()*** en **Persona**, podemos elegir diversas formas de cómo queremos representar una persona. En este ejemplo, hemos decidido que una representación adecuada consistirá en el nombre junto a la edad,omitiendo la estatura. Con el IDE que utilicemos, podemos generar automáticamente esta sobreescritura del método heredado ***toString()*** de la clase ***Object***.

```
@Override
  public String toString() { // siempre utilizar public
     return "Persona{" +
          "nombre='" + nombre + '\'' +
          ", edad=" + edad +
          '}';
  }
```

En este ejemplo podemos ver que hemos vuelto a escribir un método heredado, en este caso **toString()**, de la superclase ***Object***, para adecuarlo a las necesidades de la clase hija **Persona**, es decir, nos devolverá un ***String*** pero a nuestro gusto. Cuando una clase hija sobrescribe un método heredado de su superclase, anula el heredado y vale solo el nuevo método.

Cuando una clase sobrescribe un método, se añade antes del nombre del método la línea @***Override***. A los elementos de Java que comienzan por @ se le llaman anotaciones. Esta anotación le dice al compilador que lo que viene a continuación es un método que intencionadamente se desea sobreescribir.

> ***Override*** no es obligatorio, pero es aconsejable. No se puede sobreescribir un método si es ***static*** o ***final***.

2.6.2.- Método equals()

Su prototipo es:

public boolean equals(Object o) ;

Compara dos objetos y decide si son iguales, devolviendo ***true*** en caso afirmativo y ***false*** en caso contrario. Destacar que el operador == es válido para comparar tipos primitivos, pero no sirve para comparar objetos, ya que sólo examina sus referencias, sin fijarse en su contenido. Por ejemplo, el siguiente código:

```
Persona a = new (“Claudia”,8,1.20);
Persona b = new (“Claudia”,8,1.20);
System.out.println (a == b ); // false
```

Dará como resultado ***false*** ya que la comparación se hace atendiendo a las referencias de los objetos, que son distintas, al ocupar lugares distintos en la memoria, y no al contenido de sus atributos.

Vamos a reimplementar ***equals()*** para la clase **Persona**. Tendremos que decidir qué significa que dos personas sean iguales. Para este ejemplo, vamos a considerar dos personas iguales si tienen el mismo nombre y la misma edad (en la práctica, a la hora de comparar es muy útil utilizar atributos que identifiquen a cada objeto, como el DNI, el número de socio de una biblioteca, etc.)

```
@Override
  public boolean equals(Object otraPersona) { // compara this con otraPersona
     Persona otra = (Persona)otraPersona;    // hacemos un cast
     if((this.getNombre().equals(otra.getNombre()))&&(this.getEdad() == otra.getEdad()))
        return true;
     else
        return false;
  }
```

Ahora podemos comparar:

```
Persona a = new Persona("Claudia", 8, 1.20);
Persona b = new Persona("Claudia", 8, 0.0);
Persona a = new Persona("Pepe", 24, 1.89);
System.out.println(a.equals(b)); // true: compara atributos, no referencias
System.out.println(a.equals(c)); // false
```

La mayoría de las clases de la API tiene su propia implementación (***overriding***) de ***equals()***, que permite compara sus objetos entre sí.

2.7.-ABSTRACCIÓN

2.7.1.- Introducción

En el mundo real, no de la programación, algo abstracto es algo que existe sólo en el mundo mental. Existen perros, gatos, lobos, y todos ellos son Animales. Animal es un concepto abstracto, que contiene a los otros indicados, y que indica ciertas características de todos ellos.

En programación, una clase abstracta es una clase sobre la que no se va a crear nada real (no podemos crear objetos de una clase abstracta) y que define las características de todas las que heredan de ella.

- Ejemplo: En el mundo real, normalmente no vamos a crear objetos de "Animal", sino que crearemos objetos de animales concretos (Perros, Gatos, etc)

Por lo tanto, en ocasiones es útil definir clases de las que no pretendemos crear objetos, ya que su único objetivo es que sirvan de superclases a las clases "*reales*". Estas superclases nos valdrán para crear moldes que obliguen a las subclases a actuar como se haya dicho en la

superclase.

Las clases abstractas están solamente para ser heredadas y sobreescribir métodos, y así servir de guía a las clases hijas. Es como si una clase “obliga” a sus hijas a hacer algo que ella misma no sabe aún hacer.

Para construir clases abstractas, es necesario primero saber cómo declarar métodos abstractos.

2.7.2.- Métodos abstractos

Se denominan métodos abstractos a aquellos que se construyen con la intención de que no se definan donde se crean, sino que se espera que las clases heredadas lo hagan. Un método abstracto es un método declarado pero sin código alguno. Sirven para identificar qué han de hacer las cosas, no cómo han de hacerla. Para declarar un método abstracto se utiliza la expresión general:

public abstract tipo-nombre (lista_de_parametros);

aunque la sintaxis es la siguiente:

[modificador_acceso] abstract <tipo> <nombreMetodo> ([parámetros]) [excepciones];

Con el modificador del tipo ***abstract***, estamos precisando que las subclases sobrescriban ciertos métodos especificados con este modificador. Se suele hacer referencia a estos métodos como métodos de responsabilidad de la subclase, ya que no están implementados específicamente en la superclase. Así, la subclase debe sobrescribirlos, ya que no se puede utilizar la versión de la superclase.

- Estos métodos tendrán que ser obligatoriamente redefinidos (en realidad “definidos”, pues aún no tienen contenido) en las clases derivadas. Si en una clase derivada se deja algún método abstracto sin implementar, esa clase derivada será también una clase abstracta.
- Cualquier clase que contenga uno o más métodos abstractos debe ser declarada abstracta.
- Un método nunca puede ser declarado ***abstratc*** y ***final*** a la vez. Parece bastante evidente, ya que con ***final*** impedimos que un método se sobreescriba, por lo que un método

abstracto que no se puede sobreescribir tiene poco sentido.

- Un método no puede ser declarado ***abstract*** y ***private*** a la vez. Se aplica el mismo razonamiento que antes, ya que un método ***private*** no se puede sobreescribir.

2.7.3.- Clases abstractas

Una clase abstracta es aquella que no va a tener instancias de forma directa, aunque si que pueden haber instancias de las subclases siempre y cuando esas subclases no sean también abstractas. Por lo tanto, una clase abstracta es la que contiene al menos un método abstracto.

Se identifican por llevar el identificador ***abstract*** en la declaración de la clase, antes del nombre de la clase.

```
public abstract class Figura{
    ...
}
```

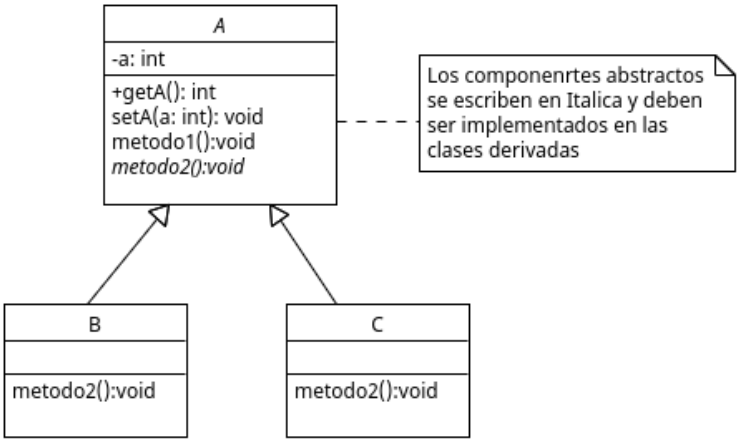
Las características de las clases abstractas son:

- Una clase abstracta puede tener lo mismo que una clase normal: atributos, métodos y constructores. De hecho es obligatorio que, como toda clase, tengan constructor, explícito o implícito.
- Los métodos pueden ser tanto abstractos como no abstractos.
- Si una clase tiene algún método abstracto, ha de ser declarada abstracta obligatoriamente. Sin embargo, una clase definida como abstracta puede no tener métodos abstractos (es opcional).
- Una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de ella. Gracias al polimorfismo, si que se puede declarar un objeto e inicializarlo con un objeto de una clase hija, pero no de la propia clase abstracta.
- Una clase abstracta se usa heredándola desde otra clase, y en esta clase, se sobreescriben los métodos abstractos. Se ha de sobreescribir obligatoriamente todos los métodos abstractos. Si no se desea sobreescribir un método abstracto en la clase hija, se puede definir en ésta última el método nuevamente como abstracto, solo que en este caso, la clase hija también habrá de ser declarada como abstracta.

- Ejemplo:

A.java

```
package abstracta;

abstract class A {
   int a = 1;

    public int getA() {
      return a;
   }

   void setA(int a) {
      this.a = a;
   }

   public void metodo1(){ // método implementado
      System.out.println("Método definido en A");
   }
   // método abstracto para ser implementado por las subclases
   //En UML se pone en cursiva
   abstract void metodo2();
}
```



B.java

```
package abstracta;

public class B extends A{
   @Override
   public void metodo2() {
      System.out.println("Método2 definido en B");
   }
}
```

C.java

```
package abstracta;

public class C extends A{
   @Override
   public void metodo2() {
      System.out.println("Método2 definido en C");
   }
}
```

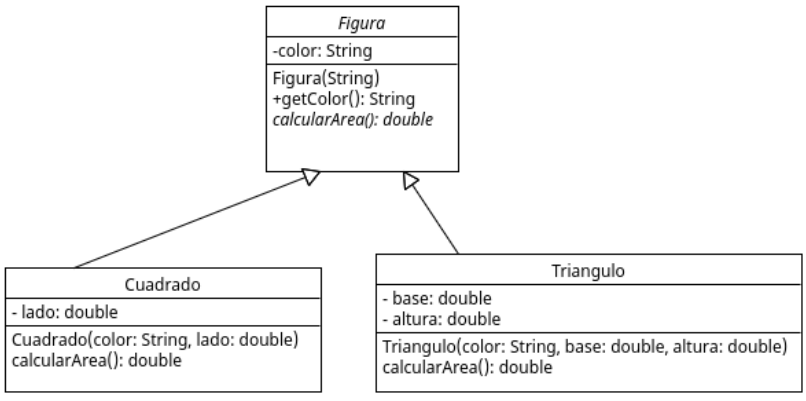
PruebaAbstracta.java

```
package pruebas;

import abstracta.B;
import abstracta.C;

public class PruebaAbstracta {
   public static void main(String[] args) {
      B b = new B();
      C c = new C();
      System.out.println("Valor de a en la clase B: "+b.getA()); // Heredado de A
      b.metodo1();
      b.metodo2();
      c.metodo1();
      c.metodo2();
   }
}
```

Salida por pantalla

```
Valor de a en la clase B: 1
Método definido en A
Método2 definido en B
Método definido en A
Método2 definido en C
```

- Ejemplo: Implementa en Java el siguiente diagrama UML.



- Ejemplo: Implementar en UML primero y después en Java el siguiente enunciado:

Jaume es un niño pequeño de 8 años amante de los animales. De todos los animales que existen en el mundo, sus preferidos son los gatos, las jirafas, los pingüinos y las águilas. Por eso su asignatura favorita son las ciencias naturales. De hecho, de mayor quiere ser veterinario.

La semana pasada Miquel, su profesor de ciencias naturales les explicó que los animales se suelen clasificar atendiendo a diferentes criterios. Como deberes, les propuso que hicieran una clasificación de sus animales favoritos.

Esa misma tarde, Jaume se fue a la biblioteca a hacer los deberes con su amiga Anna. Estaba ansioso de consultar “La Gran Enciclopedia Animal”, una de las referencias bibliográficas más importantes del mundo. En ella pudo comprobar que tanto el gato como la jirafa pertenecen a la categoría de mamíferos, ya que ellos dos tienen pelos en su cuerpo. Además, la mayoría de los mamíferos tienen mamas con las que alimentan a sus crías, aunque la gestación de cada uno de ellos es diferente. Por ejemplo, una gata suele tener una gestación de 65 días, mientras que en una jirafa es de 450 días. FASCINANTE!!!! exclamo Jaume al saber el dato. Unas páginas más adelante descubrió que el pingüino y el águila pertenecen a la categoría aves ya que tienen plumas en su cuerpo, aunque no se interesó por su gestación. Le pareció aburrida. En cambio, le pareció muy interesante los orígenes de las aves: Madagascar, Nepal,...Además, pudo comprobar que no todas las aves vuelan. Por ejemplo, el pingüino no vuela.

Otra cosa sorprendente que leyó es que existen tres sexos: hembra, macho y hermafrodita, y que cualquier animal pude tener uno de esos tres sexos.

Ya finalizando el libro de 3892 páginas, comprobó que una de las características más importante que tienen los mamíferos es que se pelean entre ellos, mientras que las aves se besan entre ellas.

Por último, dependiendo de lo que comen, los animales se pueden agrupar en: Herbívoros, si su base alimenticia está compuesta de vegetales; Carnívoros, si se alimentan de carne; Insectívoros, si se alimentan de insectos; y Omnívoros, si se alimentan de animales y vegetales. Como no iba a poner ese dato en su libreta de ciencias naturales? Miró que los gatos son omnívoros, la jirafa herbívora y el pingüino y el águila carnívoros.

Al día siguiente, su profesor Miquel le felicitó por el buen trabajo que hizo. Le comenta que seria buena idea poder crear una aplicación de gestión de sus animales favoritos y que lo podría presentar en el Congreso Jupitariano de animales terrenales. Dicho y hecho. Durante el fin de semana se puso manos a la obra. Y pensó, que también le gustaría conocer de sus animales favoritos el sonido que hacen, qué comen y visualizar la información de cada uno de ellos. Además, para cada mascota favorita le pondría un nombre, la edad que tiene, su sexo y el tipo de alimentación a partir de la información que obtuvo del “La Gran Enciclopedia Animal”. Y de sus gatos su raza y de las jirafa su tamaño. Leyó que tienen un cuello tan largo para poder alcanzar las hojas de las copas de los árboles.

El que pasa es que algunas veces sólo conoce el nombre y el sexo de las mascotas favoritas, dejando el resto de información a unos valores que indiquen que no se saben. También les gustaría saber que mascotas de la que tiene se pueden pelear y besar entre ellas.

PUEDES AYUDARLE A MODELAR EL PROBLEMA PLANTEADO Y POSTERIORMENTE IMPLEMENTARLO EN JAVA?

3.- EJERCICIO 1

1. Diseñar la clase Hora que representa un instante de tiempo compuesto por una hora (de 0 a 23) y los minutos. Dispone de los métodos:
    - Hora (hora, minuto) --> Construye un objeto con los datos pasados como parámetros.
    - inc () → Incrementa la hora en un minuto.
    - setMinutos (valor) → Asigna un valor, si tiene sentido, a los minutos.
    - setHora (valor) → Asigna un valor, si tiene sentido, a la hora.
    - toString () → Devuelve un String con la representación del reloj.
2. Escribir la clase Hora2, que funciona de forma similar a la clase Hora, con la diferencia de que las horas solo pueden tomar un valor entre la 1 y las 12; y se distingue la mañana de la tarde mediante “am” y “pm”.
3. A partir de la clase Hora implementar la clase HoraExacta, que incluye en la hora los segundos. Además de los métodos visibles de Hora dispondrá de:
    - HoraExacta (hora, minuto, segundo) --> Construye un objeto con los datos pasados como parámetros.
    - setSegundo (valor) → Asigna, cuando es posible, el valor indicado a los segundos.
    - inc () → Incrementa la hora en un segundo
4. Añadir a la clase HoraExacta un método que compare si dos horas (la invocante y otra pasada como parámetro de entrada al método) son iguales o distintas.
5. Crear la clase Instrumento que es una clase abstracta que almacena un máximo de 100 notas musicales. Mientras haya sitio es posible añadir nuevas notas musicales, al final, con el método add(). La clase también dispone del método abstracto interpretar() que en cada subclase que herede de Instrumento, mostrará por consola las notas musicales según las interprete. Utilizar enumerados para definir las notas musicales.
6. Crear la clase Piano y la clase Campana que heredan de Instrumento.
7. Las empresas de transporte, para evitar daños en los paquetes, embalan todas sus mercancías en cajas con el tamaño adecuado. Una caja se crea expresamente con un ancho,

un alto y un fondo y, una vez creada, se mantiene inmutable. Cada caja lleva pegada una etiqueta con información útil como el nombre del destinatario, dirección, etc. Se pide implementar la clase Caja con los métodos:

- Caja (int ancho, int alto, int fondo, Unidades u) → Construye una caja con las dimensiones especificadas, que pueden encontrarse en “cm” o en “m”.
- double getVolumen () → Devuelve el volumen de la caja en metros cúbicos.
- String toSring() → Devuelve una cadena con la representación de la caja.

8. La empresa de mensajería BiciExpress que reparte en bicicleta, para disminuir el peso a transportar por sus empleados solo utiliza cajas de cartón. Por motivos de privacidad, las etiquetas (con texto) que se pegan en las cajas normales se sustituyen por una etiqueta con un número (que determina el cliente, la dirección de envío, etc.). Además las cajas de cartón se caracterizan porque su volumen se calcula como el 80% del volumen real, ya que si las cajas se llenan mucho, se deforman y se rompen. Otra característica de las cajas de cartón es que sus medidas siempre están en centímetros. Por último, la empresa, para controlar costes, necesita saber cuál es la superficie total de cartón utilizado para construir todas la cajas enviadas. Se pide implementar a partir de la clase Caja la clase CajaCarton.

4.- EJERCICIO 2

1. A partir de la clase Punto implementada en la unidad 8, se pide diseñar, mediante herencia, la clase Punto3D que representa un punto en tres dimensiones (con tres componentes x, y, z)

2. A partir de la clase Calendario, implementada en la unidad 8, se pide escribir la clase CalendarioExacto que determine un instante de tiempo exacto formado por un año, un mes, un día, una hora y un minuto. Implementar los métodos toString(), equals() y aquellos necesarios para manejar la clase.

5.- INTERFACES

5.1.-INTRODUCCIÓN

Anteriormente implementamos una aplicación para ayudar al pequeño Jaume a gestionar sus animales favoritos. Sin embargo, si queremos trabajar con todos los animales, podemos ver que unos tienen la capacidad de emitir un sonido (por ejemplo los perros, los gatos o los lobos) y otros no (por ejemplo una lagartija o un caracol). Por lo tanto, los primeros tendrán el método **void sonido()** y lagartija y caracol no. Eso significa que no todos los animales han de heredar el método **sonido()**.

Sin embargo, la forma en que se ejecuta dicho método es distinta, según la clase. En un objeto Gato, la ejecución de soido() hará que aparezca en la pantalla la cadena “Miau!”. En cambio, un un objeto Perro mostrará “Guau!”. Pero todos ellos tienen en común que implementan el método sonido().

Hay una forma de expresar en Java esa capacidad común de algunas clases, en este caso aquellas que tienen implementado el método sonido(). A dichas funcionalidades comunes las definimos en lo que llamamos ***interfaces***. En el ejemplo hablaríamos de la **interfaz Sonido**, que consiste en implementar el método sonido(). Diríamos que las clases Perro, Gato y Lobo implementan la interfaz Sonido ya que, entre sus métodos está sonido(), mientras que las clases Caracol y Lagartija no.

La sintaxis en Java es:

```
[public] interface <NombreInterfaz> {

          [public] [final] <tipo1> <atributo1>= <valor1>;
          [public] [final] <tipo2> <atributo2>= <valor2>;
          ...

          [public] [abstract] <tipo_devuelto1> <nombreMetodo1> ([lista_parámetros]);
          [public] [abstract] <tipo_devuelto2> <nombreMetodo2> ([lista_parámetros]);
          ...
}
```

Por lo tanto, la definición de la interfaz y método para el ejemplo visto anteriormente sería:

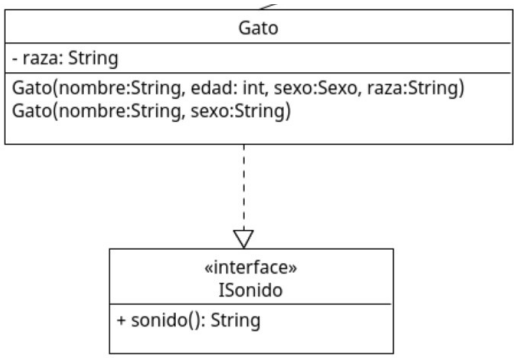
Definición interfaz

```
interface ISonido{
        //métodos de la interface
        String sonido();
}
```

Implementación en una clase del método de la interfaz

```
public class Gato extends Mamifero implements ISonido{
        public String sonido(){
                return (“Guau!!”);
        }
        … //resto de implemenatición de la clase Gato
}
```

Su representación en UML és:



El archivo de una interfaz se usa en el proyecto igual que una clase normal: se guarda en un fichero .java , su bytecode en un fichero .class, y existe en los paquetes con las mismas reglas que una clase.

En una interfaz se definen los métodos en qué consiste la interfaz. En el ejemplo, la interfaz ISonido consiste en este caso de tan solo un método: String sonido(). Su implementación dependerá de la clase concreta que implemente este método. Al ejemplo vemos el cuerpo de la implementación del método sonido para la clase Gato. Cada clase que implemente esta interfaz definirá una implementación particular para cada una de ellas.

Una interfaz puede consistir en más de un método, e incluso puede contener atributos. Diremos que una clase implementa una interfaz si implementa todos sus métodos.

En general, una interfaz es una declaración de funcionalidades, una especie de compromiso asumido por una clase. Cuando una clase declara que tiene implementado una interfaz concreta, se está comprometiendo a tener implementadas en su definición todas las funcionalidades de esa interfaz.

Desde el punto de vista sintáctico, una interfaz se parece mucho a una clase; tiene atributos y métodos. Pero las interfaces no son instanciables, es decir, no se pueden crear objetos de una interfaz. En la definición de una interfaz hay métodos (llamados métodos abstractos) que son declarados, pero no implementados, igual que ocurre en las clases abstractas. En el ejemplo, sonido() es el único método de la interfaz y además es abstracto.

Además, en una interfaz también pueden declararse atributos. Por ejemplo, imaginemos que queremos llevar la cuenta de las sucesivas versiones de nuestra interfaz ISonido. Podemos definir el atributo entero versión. La definición de la interfaz tendría la siguiente forma:

```
interface ISonido{
        int version = 1;
        String sonido();
}
```

En este caso, el atributo **version** se convertirá automáticamente en un atributo de las clases

que implementan la interfaz ISonido. En el ejemplo, la clase Gato tendrá el atributo **version**. Esta clase no lo podrá cambiar ya que los atributos definidos en una interfaz son ***static*** y ***final*** por defecto.

A parte de los métodos que se declaran en la interfaz y que no son implementados (**métodos abstractos**), también están los **métodos por defecto** y los **métodos estáticos**, los cuales si que están implementados en la propia interfaz. Los métodos abstractos de una interfaz deben ser implementados por las clases que implementan dicha interfaz, pero los métodos por defecto y estáticos, no.

Para ser exactos, se dice que una clase implementa una interfaz cuando implementa sus métodos abstractos. En cambio, los **métodos no abstractos** de una interfaz (por defecto y estáticos) **se llaman de extensión**. Éstos son implementados en la definición de la interfaz, donde también se declaran e inicializan los atributos. Tanto unos como otros son incorporados por las clases de forma automática (por defecto), aunque los métodos por defecto pueden se reimplementados por las clases, haciendo *overriding* de ellos.

Los métodos por defecto se deben declara como ***default***. Siguiendo con el ejemplo, supongamos que, entre los sonidos que emiten algunos animales, queremos incluir los ruidos que hacen durmiendo, y que todos emiten el mismo sonido cuando duermen. En este caso, podemos incluir e implementar la función **String sonidoDurmiendo()** en la interfaz, sin tener que esperar a la definición de ninguna clase particular.

```
interface ISonido{
        int version = 1;
        String sonido();
        default String sonidoDurmiendo(){
                System.out.println("Zzzzzzzzzzz");
        }
}
```

Este método, **String sonidoDurmiendo()** será incorporado a las clases que implementen la interfaz ISonido y será accesible por cualquier objeto de una de esas clases. En nuestro caso, será accesible desde la clase Gato.

```
Gato g = new Gato("Pepito", Sexo.MACHO);
System.out.println(g.sonidoDurmiendo());
//Suponemos que la clase Perro implementa
//también la interfaz ISonido
Perro p = new Perro("Marujita",Sexo.HEMBRA);
System.out.println(p.sonidoDurmiendo());
```

En ambos casos se muestra por pantalla el mensaje: Zzzzzzzzzz

No obstante, sonidoDurmiendo() se puede reimplementar en cada clase haciendo overriding de la implementación que se ha hecho en la interfaz. Supongamos que descubrimos que los leones (clase que también implementan la interfaz ISonido) son una excepción, y rugen hasta cuando duermen. Entonces la clase Leon sería de la siguiente manera:

```
Class Leon extends Mamifero implements ISonido{
        public String sonido() {
                return "Grrrrr!!";
        }
        @Override //de la implementación en ISonido, debe ser public
        public String sonidoDurmiendo() {
                //return Sonido.super.sonidoDurmiendo();
                return "Grrrrr!!";
        }
}
```

```
Leon le = new Leon();
le.sonidoDurmiendo();

Mostraría por consola "Grrrrr!!"
```

No se debe de olvidar el modificador *public* para hacer el overriding.

En cambio, si implementamos un método estático en una interfaz, pertenecerá a la interfaz y no a las clases que la implementan, y mucho menos a los objetos instanciados. Por ejemplo, supongamos que todos los animales, sin excepción, emiten el mismo sonido al bostezar. Entonces podríamos implementar un métodos estático para los bostezos como se indica a continuación:

```
interface ISonido{
        int version = 1;
        String sonido();
        default String sonidoDurmiendo(){
                System.out.println("Zzzzzzzzzzz");
        }
        static String bostezo(){
                return "Aaaauuuh!";
        }
}
```

Este método será accesible directamente desde la interfaz ISonido. Para invocarlo, tendremos que escribir:

```
System.out.println(ISonido.bostezo());
```

```
Aaaauuuh!
```

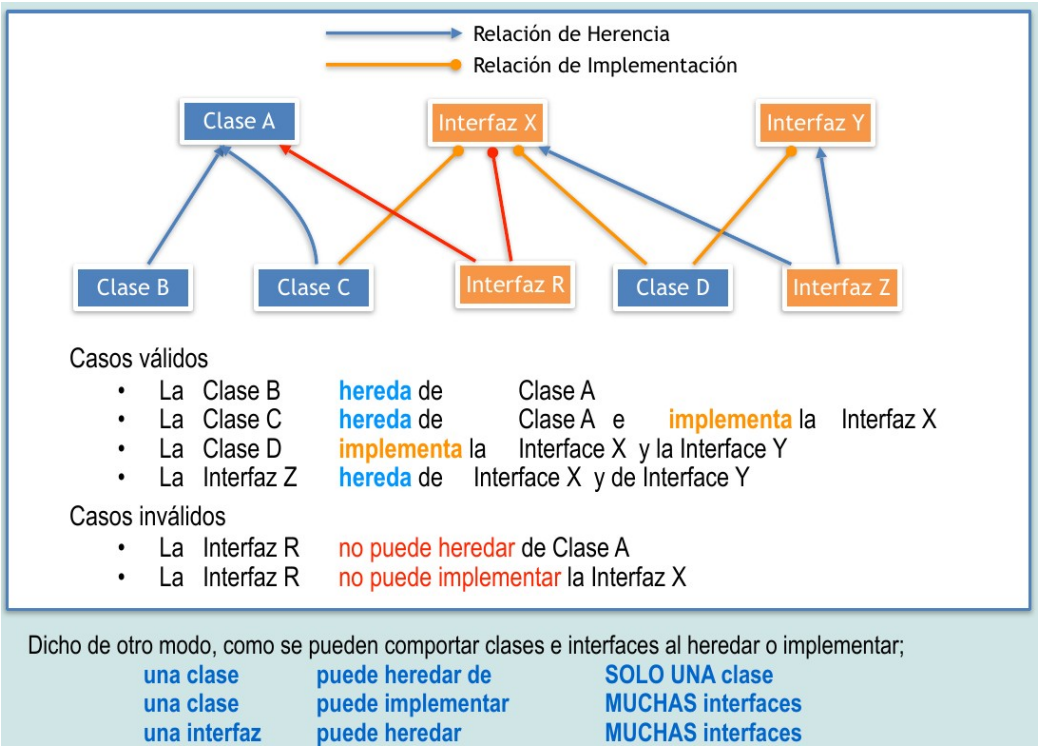
Las interfaces pueden heredar unas de otras y se pueden crear variables cuyo tipo es una interfaz, con las que podremos referenciar cualquier objeto de cualquier clase que la implementa. Por ejemplo, podremos crear variables de tipo ISonido.

```
ISonido so;
so = new Gato("Migueli",Sexo.MACHO);
System.out.println(so.sonido());
//La misma variable puede referenciar a otra clase
//que implemente la misma interfaz ISonido
so = new Perro("Milú",Sexo.MACHO);
System.out.println(so.sonido());
```

```
MIAU!!
BUFFF!!
```

Este es un ejemplo de una de las formas de polimorfismo de Java: la selección dinámica de métodos. La misma línea de código produce efectos distintos (ejecuta métodos distintos), dependiendo del objeto referenciado por la variable **so**. Se decide en tiempo de ejecución qué implementación concreta del método se va a ejecutar. Es importante saber que cuando usemos una variable de tipo ISonido para referenciar un objeto, solo tendremos acceso a los métodos de ese objeto que pertenezcan a la interfaz ISonido.

Como puede verse a partir de su definición, una interfaz es semejante a una clase abstracta. La diferencia es que una clase solo puede heredar de una clase abstracta, mientras que puede implementar más de una interfaz. Las clases abstractas están para ser heredadas por otras clases, mientras que las interfaces son implementadas por las clases. Por otra parte, mientras que una clase solo se puede heredar de una clase, una interfaz puede heredar de más de una interfaz (entre las interfaces es posible la herencia múltiple). Todo esto lo podemos ver en el siguiente gráfico:



A partir de lo visto anteriormente, la sintaxis general de la definición de una interfaz es:

```
tipoDeAcceso interface NombreInterface{
        //atributos, son public, static y final por defecto:
        tipo atributo1 = valor1;
        … //otros atributos

        //métodos abstractos, sin implementar
        tipo metodo1 (listaPar1);
        … //otros métodos abstractos

        //métodos static, implementados:
        static tipo metodoEstatico (listaPar1){
                //cuerpo del método estático
        }
        … //otros métodos estáticos

        //métodos por defecto, implementados:
        default tipo metodoDefault1(listaPar1){
                //cuerpo de metodoDefault1
        }
        … //otros métodos por defecto
}
```

tipoDeAcceso puede omitirse, en cuyo caso el acceso está restringido al paquete en el que está incluida, o es *public* para que la interfaz pueda ser importada desde otro paquete mediante una sentencia *import*.

Los atributos por defecto son *public*, *final* y *static*, sin que haya que especificarlo de forma explícita en la definición. Se accede a ellos a través del nombre de la interfaz o de una clase que la implemente, pero no desde una instancia. Por ejemplo, si queremos mostrar la versión actual de la interfaz ISonido, pondremos:

```
System.out.println(Sonido.version);
```

o bien:

```
System.out.println(Gato.version);
```

Para que una clase implemente una interfaz, debe declararla en el encabezamiento, usando la palabra reservado *implements*, e implementar todos los métodos abstractos de la interfaz en el cuerpo de la definición de la clase.

```
tipoDeAcceso class NombreClase implements NombreInterfaz {
        …
        public tipo metodoAbstracto1(listaPar1){
                … // cuerpo del método 1
        }
        … // otros métodos abstractos
}
```

Una clase puede implementar más de una interfaz, para lo cual deberá escribirlas separadas por comas, en el encabezamiento, e implementar todos los métodos abstractos de cada una de ellas.

```
tipoDeAcceso class NombreClase implements Interfaz1, Interfaz2, … {
        …
}
```

5.2.-CLASES ANÓNIMAS

Lo más común es que las interfaces sean implementadas por distintas clases de las que, por otra parte, se crearán diversos objetos. Sin embargo, hay ocasiones en que una determinada implementación de una interfaz es necesaria en un solo lugar. En ese caso no merece la pena definir una clase nueva para crear un solo objeto.

En su lugar podemos crear un objeto de una clase sin nombre, es decir, anónima, donde se implementan los métodos de la interfaz. Como no disponemos de nombre para la clase, tampoco lo tendremos para el constructor. Por eso usaremos un constructor con el nombre de la interfaz. Asimismo, el objeto será referenciado por una variable del tipo de la interfaz. Como ejemplo vamos a crear una clase anónima que implemente la interfaz **ISonido**. Imaginemos que alguien ha encontrado un animal de una especie desconocida, que emite un extraño sonido. Mientras se identifica o no, para describir su sonido crearemos un objeto de una clase anónima que implementa la interfaz ISonido,

```
ISonido son = new ISonido(){
        public void sonido(){
                System.out.println("Jajejijojuuuuuu!");
        }
};

System.out.printf(son.sonido());
```

En realidad no hemos definido ninguna clase. Lo que hemos hecho es crear un objeto sin clase, aunque es costumbre hablar de clases anónimas.

5.3.-INTERFAZ COMPARABLE

En programación hay operaciones tan necesarias y tan frecuentes, que merece la pena definir una interfaz que las declare. Una de esas operaciones es la de comparar dos valores para hacer búsquedas u ordenarlos. Con este fin, los desarrolladores de Java han implementado un par de interfaces, **Comparable** y **Comparator**. Además, está el método **equals()**, que aunque no pertenezca a ninguna interfaz especial, sino a la clase raíz ***Object***, también nos servirá para hacer comparaciones.

La interfaz **Comparable** consta de un único método abstracto:

int compareTo (Object ob);

Para que una clase implemente la interfaz **Comparable**, deberá implementar este método, que se comportará de la siguiente forma. Supongamos que queremos comparar dos objetos **ob1** y **ob2** de una determinada clase. Deberemos de ejecutar la sentencia:

ob1.compareTo (ob2);

Los posibles resultados son:

ob1.compareTo (ob2) < 0 si ob1 va antes que ob2.

ob1.compareTo (ob2) > 0 si ob1 va después que ob2.

ob1.compareTo (ob2) == 0 si ob1 es igual que ob2

- Ejemplo: Supongamos que queremos ordenar una lista de objetos de la clase **Persona**. Naturalmente, tendremos que escoger un criterio de ordenación. En este caso queremos ordenar por los números de identificación (**id**) crecientes.

```
// la clase Persona tendrá que implementar los métodos de Comparable
class Persona implements Comparable{
        int id;
        String nombre;
        int edad;

        Persona (int id, String nombre, int edad){ // constructor
                this.id = id;
                this.nombre = nombre;
                this.edad = edad;
        }
        @Override // la implementación debe declararse public
        public int compareTo(Object o) {
                if(this.id < ((Persona)o).id) //this va antes que o
                        return -1; // devolvemos cualquier número negativo
                else if(this.id > ((Persona)o).id) // this va después que o
                        return 1; // devolvemos cualquier número positivo
                else // this es igual que o
                        return 0;
        }
}
```

Ahora ejecutamos el código para ver el resultado:

```
Persona p1 = new Persona(3,"Anselmo",14);
Persona p2 = new Persona(1,"Josefa",15);
System.out.println(p1.compareTo(p2));
System.out.println(p1.compareTo(p1));
System.out.println(p2.compareTo(p1));
```

Como estamos comparando un atributo (**id**) de tipo numérico para comparar, hay una implementación mucho más sencilla para calcular el resultado de la comparación:

```
@Override // la implementación debe declararse public
public int compareTo(Object o) {
        return (this.id – ((Persona)o).id);
}
```

La interfaz **Comparable** ha sido creada por los desarrolladores de Java y es reconocida por otras clases de la API. Entre ellas la clase **Arrays**, donde se implementa el método **sort()**, que sirve para ordenar un *array*.

- Ejemplo: Declaramos e inicializamos el *array* de números enteros:

  int [ ] t = {5, 3, 6, 9, 3, 4, 1, 0, 10};

  la sentencia **Arrays.sort(t)** ordena el *array* **t** por el orden natural de sus elementos que, en el caso de los números enteros, es el orden creciente.

  El método **sort()** no devuelve una copia ordenada de **t**, sino que ordena el *array* original. Para mostrarlo podemos recurrir a otra función de la clase **Arrays**:

  System.out.println(Arrays.toString(t));

  que mostrará por pantalla: [0, 1, 3, 3, 4, 5, 6, 9, 10]

El método **sort()** también sirve para ordenar un *array* de objetos de cualquier clase, con tal de que esta tenga implementada la interfaz **Comparable**. Utilizará el orden natural de la clase que, por definición será el establecido por el método **compareTo()**. Por lo tanto, para usar la utilidad **sort()** con objetos de una determinada clase, ésta deberá tener implementada la interfaz **Comparable**, que será la que establezca el criterio de ordenación natural de la clase.

- Ejemplo: Vamos a ordenar el *array* de objetos **Persona**, por medio del método **sort()** ya que implementamos anteriormente la interfaz **Comparable**. Escribiríamos el siguiente código:

```
Persona[] p = new Persona[]{
        new Persona(2,"Anna",19),
        new Persona(5,"Jordi",12),
        new Persona(1,"Joan",10),
        new Persona(3,"Amancio",18),
};

Arrays.sort(p);
System.out.println(Arrays.deepToString(p));
```

```
[Id: 1 Nombre: Joan Edad: 10
, Id: 2 Nombre: Anna Edad: 19
, Id: 3 Nombre: Amancio Edad: 18
, Id: 5 Nombre: Jordi Edad: 12
]
```

En vez de utilizar **toString()**, válido para *arrays* de tipos primitivos, hemos utilizado **deepToString()** para mostrar un *array* de objetos **Persona**. Este método llamará al método

toString() que hemos implementado en la clase **Persona**, para mostrar los objetos individuales, obteniendo por pantalla lo mostrado anteriormente.

Muchas clases implementadas en la API de Java, como la clase **String**, implementan la interfaz **Comparable**, con lo cual podemos comparar sus objetos y ordenarlos por medio de **sort()**. En particular, las cadenas se comparan por medio de **compareTo()** siguiendo el orden alfabético creciente.

5.4.-INTERFAZ COMPARATOR

Hemos visto que la interfaz **Comparable** proporciona un criterio de ordenación a la clase que la implemente, llamada ordenación natural, que es la que usan por defecto los distintos métodos de la API, como **sort()**. Pero es frecuente que tengamos que ordenar los objetos de la misma clase con distintos criterios, según las circunstancias. Por ejemplo, puede ser que necesitemos un listado de objetos **Persona** por orden alfabético de nombres, o por edades, en sentido creciente o decreciente. Para resolver este problema existe la interfaz **Comparator**. Para usarla habrá que importarla con la sentencia:

```
import java.util.Comparator;
```

Esta interfaz tiene un único método abstracto:

```
int compare(Object ob1, Object ob2);
```

Negativo si ob1 va antes de ob2

Positivo si ob1 va después de ob2

Cero si ob1 y ob2 son iguales

Llamamos comparador a cualquier objeto de una clase que implemente la interfaz **Comparator**. Necesitaremos una clase de comparadores distinta para cada criterio de comparación que queramos emplear para objetos de una clase determinada.

- Ejemplo: Para la clase **Persona**, vamos a implementar una clase comparadora para el atributo edad.

ComparaEdades.java

```
import persona.Persona;

import java.util.Comparator;

public class ComparaEdades implements Comparator {
  @Override
  public int compare(Object o1, Object o2) {
    Persona p1 = (Persona) o1;
    Persona p2 = (Persona) o2;
    return p1.edad - p2.edad;

  }
}
```

PruebaPersonas.java

```
Persona[] p = new Persona[]{
        new Persona(2,"Anna",19),
        new Persona(5,"Jordi",12),
        new Persona(1,"Joan",10),
        new Persona(3,"Amancio",18),
    };

    //Comparar edades
    ComparaEdades e = new ComparaEdades();
    Arrays.sort(p,e);
    System.out.println(Arrays.deepToString(p));
```

RESULTADO:

```
[Id: 1 Nombre: Joan Edad: 10
, Id: 5 Nombre: Jordi Edad: 12
, Id: 3 Nombre: Amancio Edad: 18
, Id: 2 Nombre: Anna Edad: 19
]
```

En vez de declarar la variable **e**, podríamos haber creado el comparador en la propia llamada a la función **sort()**:

Arrays.sort(p, new ComparaEdades());

Otra opción, en caso de que se vaya a hacer la ordenación una sola vez, es crear una clase anónima en la llamada al método **sort()**:

```
Arrays.sort(p, new Comparator()){
        public int compare (Object o1, Object o2){
                return ((Persona) o1).edad – ((Persona) o2).edad;
        }
}

System.out.println(Arrays.deepToString(p));
```

Si en algún otro lugar del programa tuviéramos que mostrar las personas ordenadas por orden alfabético de nombres, tendríamos que definir otra clase comparadora:

ComparaNombres.java

```
package persona;

import java.util.Comparator;

public class ComparaNombres implements Comparator {
  @Override
  public int compare(Object o1, Object o2) {
    String nombre1 = ((Persona)o1).nombre;
    String nombre2 = ((Persona)o2).nombre;
    return nombre1.compareTo(nombre2)
  }
}
```

PruebaPersonas.java

```
Persona[] p = new Persona[]{
        new Persona(2,"Anna",19),
        new Persona(5,"Jordi",12),
        new Persona(1,"Joan",10),
        new Persona(3,"Amancio",18),
    };

  //Comparar nombres
    Arrays.sort(p, new ComparaNombres());
    System.out.println(Arrays.deepToString(p));
```

RESULTADO:

```
[Id: 3 Nombre: Amancio Edad: 18
, Id: 2 Nombre: Anna Edad: 19
, Id: 1 Nombre: Joan Edad: 10
, Id: 5 Nombre: Jordi Edad: 12
]
```

Podemos ver que para obtener el resultado, se llama al método **compareTo()** de la clase **String**, ya que se están comparando **nombre1** y **nombre2**, que son cadenas.

En clases más complejas, con más atributos, podrá haber más criterios posibles de ordenación. Para cada uno de ellos, si se va a usar, habría que definir una clase comparadora apropiada.