

TEMA 6: Utilización avanzada de clases.

Sumario

.....	1
Introducción.....	2
Herencia.....	4
Acceso a miembros heredados.....	6
Constructores y herencia.....	8
Sobreescritura de métodos heredados.....	9
Herencia múltiple.....	10
Clases abstractas.....	11
Declaración de una clase abstracta.....	11
Clases y métodos finales.....	14
Polimorfismo.....	14
instanceof.....	17
Interfaces.....	19
Implementación de interface.....	21
Clases Anónimas.....	24
lambdas.....	27
Prototipo de las expresiones lambda.....	30
Interfaz funcional.....	31
Stream.....	33

Introducción

A la hora de diseñar un conjunto de clases para modelar el conjunto de información cuyo tratamiento se desea automatizar, es importante establecer apropiadamente las posibles relaciones que puedan existir entre unas clases y otras.

En algunos casos es posible que no exista relación alguna entre unas clases y otras, pero lo más habitual es que sí exista: una clase puede ser una especialización de otra, o bien una generalización, o una clase contiene en su interior objetos de otra, o una clase utiliza a otra, etc. Es decir, que entre unas clases y otras habrá que definir cuál es su relación (si es que existe alguna).

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).

La relación de **clientela** la llevas utilizando desde que has empezado a programar en Java, pues desde tu clase principal (clase con método **main**) has estado declarando, creando y utilizando objetos de otras clases. Por ejemplo: si utilizas un objeto **String** dentro de la clase principal de tu programa, éste será cliente de la clase **String** (como sucederá con prácticamente cualquier programa que se escriba en Java). Es la relación fundamental y más habitual entre clases. Tenemos que entender que estamos usando esos objetos pero no son parte de la estructura de nuestra clase (que fueran atributos de nuestra clase) porque si fueran atributos ya no sería clientela sino composición.

Observar el siguiente ejemplo:

```
class Complejo{
    double real;
    double imag;
    void mostrar(){
        String texto = "**** Número Complejo ****\n";
        texto = texto + "("+real+", "+imag+")";
        System.out.println(texto);
    }
}

public class NewClass {
    public static void main(String[] args) {
        Complejo c = new Complejo();
        c.real = 4;
        c.imag = 5;
        c.mostrar();
    }
}
```

Si se observa en el método mostrar() de la clase Complejo se usa la clase String pero no aparece el objeto como un atributo (los atributos son double) luego es clientela

Por otro lado la clase NewClass dentro de su método main() utiliza un objeto de la clase Complejo y tampoco es un atributo. Así que también es una relación de tipo clientela

La relación de **composición** es posible que ya la hayas tenido en cuenta si has definido clases que contenían (tenían como atributos) otros objetos en su interior, lo cual es bastante habitual. Por ejemplo, si escribes una clase donde alguno de sus atributos es un objeto de tipo **String**, ya se está produciendo una relación de tipo **composición** (tu clase “tiene” un **String**, es decir, está compuesta por un objeto **String** y por algunos elementos más).

```
class Complejo{
    double real;
    double imag;
    String texto;
    Complejo(int re, int img){
        real = re;
        imag = img;
        texto = "("+real+", "+imag+")";
    }
}
```

en la versión de Complejo que acabamos de poner aparece un atributo String y por tanto sería composición

La relación de **anidamiento** (o **anidación**) es quizá menos habitual, pues implica declarar unas clases dentro de otras (**clases internas** o **anidadas**). En algunos casos puede resultar útil para tener un nivel más de encapsulamiento y ocultación de información.

Podría decirse que tanto la **composición** como la **anidación** son casos particulares de **clientela**, pues en realidad en todos esos casos una clase está haciendo uso de otra (al contener atributos que son objetos de la otra clase, al definir clases dentro de otras clases, al utilizar objetos en el paso de parámetros, al declarar variables locales utilizando otras clases, etc.).

Finalmente está herencia donde unas clases derivaban de otras. Es este tipo de relación el que vamos a trabajar ahora

Herencia

La herencia es un poderosísimo modo de reutilizar código. Cuando una clase deriva de otra ("es hija de otra", suele decirse), hereda todos sus atributos y métodos. Es decir, hereda datos y código, como en la vida real heredamos rasgos de nuestros padres.

Una clase derivada (o clase hija) puede sobrescribir (**override**, que queda más técnico) algunos atributos o métodos, y también puede sobrecargarlos, es decir, hacer nuevas versiones de un método heredado pero conservando el original. Todo ello para adaptar los métodos de su madre a sus necesidades, o para ampliarlos de algún modo.

En Java, cada clase sólo puede heredar de otra. Es lo que se denomina herencia simple. Otros lenguajes, como C++, permiten a cada clase heredar de varias, es decir, herencia múltiple.

Veamos un ejemplo:

```
class Animal{
    String nombre;
    int edad;
    int peso;
}

class Perro {
    String nombre;
    int edad;
    int peso;
    int dientes;
}

class Pajaro {
    String nombre;
    int edad;
    int peso;
    double ala;
}
```

Es fácil ver que estamos repitiendo todos los atributos de Animal tanto en Perro como en Pajaro. El anterior código quedaría establecido con herencia de la siguiente forma:

```
class Animal{
    String nombre;
    int edad;
    int peso;
}

class Perro extends Animal{
    int dientes;
}

class Pajaro extends Animal{
    double ala;
}
```

Ahora es fácil ver el mecanismo. Usamos la palabra clave: extends y los atributos y métodos se heredan.

En general tendríamos:

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

Otro ejemplo:

```
class Persona {  
  
    String nombre;  
    String apellidos;  
}  
  
public class Alumno extends Persona {  
    String curso;  
    double notaMedia;  
}
```

● **Práctica 1:** Crear una clase Profesor, que hereda de Persona y contará con atributos como nombre, apellidos, salario y especialidad

Acceso a miembros heredados

Como ya has visto anteriormente, no es posible acceder a miembros privados de una superclase. Para poder acceder a ellos podrías pensar en hacerlos públicos, pero entonces estarías dando la opción de acceder a ellos a cualquier objeto externo y es probable que tampoco sea eso lo deseable. Para ello se inventó el modificador `protected` (protegido) que permite el acceso desde

clases heredadas, pero no desde fuera de las clases (estrictamente hablando, desde fuera del paquete), que serían como miembros privados.

En la unidad dedicada a la utilización de clases ya estudiaste los posibles modificadores de acceso que podía tener un miembro: sin modificador (acceso de paquete), público, privado o protegido. Aquí tienes de nuevo el resumen:

Cuadro de niveles accesibilidad a los atributos de una clase

	Misma clase	Subclase	Mismo paquete	Otro paquete
Sin modificador (paquete)	X		X	X
public	X	X	X	X
private	X			
protected	X	X	X	

Observar que cuando no decimos nada no hay problema de visibilidad desde las clases heredadas pero observemos lo que ocurre si estableciéramos private.

● **Práctica 2:** Poner los atributos de la clase persona a private. Tratar de acceder desde la clase Alumno a esos atributos que antes heredaba. Toma captura de pantalla de los mensajes del IDE. Para ello, por ejemplo, puedes crear un método en alumno y llamar a this.nombre desde el método.

Al definir la clase Alumno como heredera de Persona, no habrías tenido acceso a esos atributos, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar esos atributos como protected

Con los métodos resulta similar. Por ejemplo si tuviéramos getter y setter definidos en Persona, Alumno los heredaría

● **Práctica 3:** Crear getter() y setter() en Persona Hacer uso desde desde estos métodos para un objeto de la clase Alumno Tomar captura de pantalla del IDE aparte del código

Constructores y herencia

Recuerda que un constructor de una clase puede llamar a otro constructor de la misma clase, previamente definido, a través de la referencia `this`. En estos casos, la utilización de `this` sólo podía hacerse en la primera línea de código del constructor.

Un constructor de una clase derivada puede hacer algo parecido para llamar al constructor de su clase base mediante el uso de la palabra **super**. De esta manera, el constructor de una clase derivada puede llamar primero al constructor de su superclase para que inicialice los atributos heredados y posteriormente se inicializarán los atributos específicos de la clase: los no heredados.

Nuevamente, esta llamada también debe ser la primera sentencia de un constructor (con la única excepción de que exista una llamada a otro constructor de la clase mediante `this`). El uso de `this()` y `super()` es mutuamente excluyente. Si se llama a `super()` dentro de un método no se puede llamar también a `this()` en ese método


En el caso del constructor por defecto (el que crea el compilador si el programador no ha escrito ninguno), el compilador añade lo primero de todo, antes de la inicialización de los atributos a sus valores por defecto, una llamada al constructor de la clase base mediante la referencia `super`.

Si la clase `Persona` tuviera un constructor de este tipo:

```
public Persona (String nombre, String apellidos) {  
    this.nombre= nombre;  
    this.apellidos= apellidos;  
}
```

y ahora la llamada desde `Alumno`:

```
public Alumno(String nombre, String apellidos, String grupo, double notaMedia) {  
    super(nombre, apellidos);  
    this.grupo= grupo;  
    this.notaMedia= notaMedia;  
}
```

 **Práctica 4:** Crear un constructor para la clase `Profesor` que también heredaba de `Persona`. Hacer uso de `super()` para este constructor

● **Práctica 5:** Crear un constructor para Animal y un constructor para Pajaro y Perro. Hacer uso de super() para esos constructores

Sobreescritura de métodos heredados

Una clase puede redefinir algunos de los métodos que ha heredado de su clase base. En tal caso, el nuevo método (especializado) sustituye al heredado.

En cualquier caso, aunque un método sea sobrescrito o redefinido, aún es posible acceder a él a través de la referencia super: **super.metodo()** aunque sólo se podrá acceder a métodos de la clase padre y no a métodos de clases superiores en la jerarquía de herencia.

Los métodos redefinidos pueden ampliar su accesibilidad con respecto a la que ofrezca el método original de la superclase, pero nunca restringirla. Por ejemplo, si un método es declarado como protected o de paquete en la clase base, podría ser redefinido como public en una clase derivada.

Los métodos estáticos o de clase no pueden ser sobrescritos. Los originales de la clase base permanecen inalterables a través de toda la jerarquía de herencia.

Cuando sobrescribas un método heredado en Java puedes incluir la anotación **@Override**. Esto indicará al compilador que tu intención es sobrescribir el método de la clase padre. De este modo, si te equivocas (por ejemplo, al escribir el nombre del método) y no lo estás realmente sobrescribiendo, el compilador producirá un error y así podrás darte cuenta del fallo. En cualquier caso, no es necesario indicar @Override, pero puede resultar de ayuda a la hora de localizar este tipo de errores (crees que has sobrescrito un método heredado y al confundirte en una letra estás realmente creando un nuevo método diferente).

Ej. Ya conocemos toString() que es un método que se hereda desde la clase raíz: Object. Veamos como tenemos uno creado en Animal y la sobreescritura en Pajaro

```
class Animal{
    private String nombre;
    int edad;
    int peso;
    @Override
    public String toString(){
        return nombre +" "+edad+" años "+peso+"kg";
    }
}
```

```

class Pajaro extends Animal{
    double ala;
    @Override
    public String toString(){
        return super.toString() + " ala: " + ala + "cm";
    }
}

```

● **Práctica 6:** Crear un método: void imprimirDatos() en Persona que mostrará en pantalla todos los atributos de Persona. Sobreescibir el método en Alumno y que haga uso de super.imprimirDatos()

● **Práctica 7:** Tomando la clase Cuenta que definimos en la UT sobre clases y objetos: atributos: String numero, String titular, double saldo, ingresar(), retirar() Crear una clase hija llamada CuentaBCTT que refleja las cuentas del banco BCTT Este banco por cada ingreso retiene en comisión 0.50€ y por cada retirada de efectivo un 0.02% del dinero retirado. Hacer uso de la palabra reservada super

● **Práctica 8:** Tomando la clase Coche con atributos marca, modelo, potencia, puertas, matricula Crear la clase CocheSEAT Los modelos de esta marca siempre empezarán por “se-” agregando ese texto siempre al modelo que se le pase al constructor o al setter (salvo que ya se le esté pasando, cosa que se debe validar)
Observar que el atributo marca en coche no tiene mucho sentido que sea heredado por CocheSEAT

● **Práctica 9:** Crear la clase Libro con atributos String []autor, String titulo, String resumen, int paginas aparte de getter y setter un método boolean libroGrande() que comprueba que el número de páginas es mayor de 500. Crear VolumenLibro que hereda de Libro y tiene adicionalmente String propietario, estadoConservacion que será un enum: bueno, regular, malo y un id de tipo entero. Sobreescibir el toString() en ambas clases . VolumenLibro también tiene un atributo boolean variosTomos que está a true si el libro es de más de 500páginas

Herencia múltiple

En determinados casos podrías considerar la posibilidad de que se necesite heredar de más de una clase, para así disponer de los miembros de dos (o más) clases disjuntas (que no derivan una de la otra). La herencia múltiple permite hacer eso: recoger las distintas características (atributos y métodos) de clases diferentes formando una nueva clase derivada de varias clases base.

El problema en estos casos es la posibilidad que existe de que se produzcan ambigüedades, así, si tuviéramos miembros con el mismo identificador en clases base diferentes, en tal caso, ¿qué miembro se hereda? Para evitar esto, los compiladores suelen solicitar que ante casos de ambigüedad, se especifique de manera explícita la clase de la cual se quiere utilizar un determinado miembro que pueda ser ambiguo.

Ahora bien, la posibilidad de herencia múltiple no está disponible en todos los lenguajes orientados a objetos. Se puede encontrar en C++ por ejemplo, pero no en Java

Clases abstractas

En determinadas ocasiones, es posible que necesites definir una clase que represente un concepto lo suficientemente abstracto como para que nunca vayan a existir instancias de ella (objetos). ¿Tendría eso sentido? ¿Qué utilidad podría tener?

Imagina una aplicación para un centro educativo que utilice las clases de ejemplo Alumno y Profesor, ambas subclases de Persona. Es más que probable que esa aplicación nunca llegue a necesitar objetos de la clase Persona, pues serían demasiado genéricos como para poder ser utilizados (no contendrían suficiente información específica). Podrías llegar entonces a la conclusión de que la clase Persona ha resultado de utilidad como clase base para construir otras clases que hereden de ella, pero no como una clase instanciable de la cual vayan a existir objetos. A este tipo de clases se les llama clases abstractas.

Declaración de una clase abstracta

Las **clases abstractas** se declaran mediante el modificador **abstract**:

```
[modificador_acceso] abstract class NombreClase [herencia] [interfaces] {  
    ...  
}
```

- Una **clase abstracta** sólo puede usarse para crear nuevas clases derivadas. **No se puede hacer un `new` de una clase abstracta. Se produciría un error de compilación.**
- Una **clase abstracta** puede contener **métodos totalmente definidos (no abstractos)** y **métodos sin definir (métodos abstractos)**.

Observar el siguiente ejemplo

```
public class PruebaAbstracto {
    public static void main(String[] args) {
        Pajaro p = new Pajaro();
    }
}

abstract class Animal{
    protected String nombre;
    protected int edad;
    protected int peso;
    public Animal(){
        System.out.println("jajaja! soy un animal!!!!");
    }
    //private abstract String emitirSonido();
}

class Perro extends Animal{
    int dientes;
}

class Pajaro extends Animal{
    double ala;
}
```

● **Práctica 10:** Utiliza el código del ejemplo de abstract Animal en el IDE y ejecútalo. Toma captura de pantalla ¿hay alguna salida de pantalla ? ¿se puede utilizar el constructor de una clase abstracta ? Ahora trata de crear un objeto: `new Animal()` y toma captura de pantalla del error que da el IDE ¿ qué significa el error ?

● **Práctica 11:** Quita los comentarios del método `emitirSonido()` ¿ da algún error el IDE ? Toma captura de pantalla. ¿Se puede combinar `private` y `abstract`? Cambia `private` a `public` ¿ el IDE da algún mensaje? Pon el cursor en la línea que declaras la clase `Pajaro` y pulsa: `ALT + INTRO` y selecciona la opción “implement all abstract methods” Haz lo propio con la clase `Perro` pero esta vez utiliza `ALT + INSERT` (toma captura de pantalla de las ventanas que salgan) . Quita el modificador `abstract` de donde se define la clase `Animal`

pero deja ese modificador en el método emitirSonido(). Toma captura de pantalla del mensaje contextual del IDE. Pulsa ALT + INTRO toma captura de pantalla de la ventana que muestra y sigue su recomendación de hacer la clase abstracta

Haz podido comprobar con la actividad anterior que:

Cuando una clase contiene un método abstracto tiene que declararse como abstracta obligatoriamente.

Adicionalmente debemos tener en cuenta:

- Los métodos abstractos deben ser redefinidos
- Puede haber métodos abstractos y métodos no abstractos en una misma clase.

● **Práctica 12:** Crear una clase Persona que es abstracta y que tiene atributos: nombre, apellido1, apellido2, int edad, int altura, double peso. La altura está pensada en cm y el peso en kg debe tener al menos un constructor con esos 6 parámetros. Sobreescibir toString() para que muestre los datos de la persona. Crear el método protected double calcularIMC() que devuelve el IMC de la persona. Crear también un método abstracto calcularPesoIdeal(). Crear las clases Hombre y Mujer y para obtener calcularPesoIdeal tener en cuenta la fórmula:

$$\text{peso ideal} = \text{altura} - 100 - ((100 - 150)/k)$$

siendo k=4 si hombre y k=2 si mujer

el toString() de hombre y mujer debe hacer uso del toString() de Persona y agregar el resultado del peso ideal

Clases y métodos finales

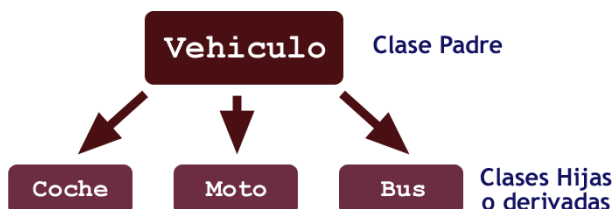
Mediante la palabra clave: **final** sabemos que podemos hacer que un atributo no pueda ser modificado. Si lo aplicamos a una clase, esa clase no puede tener clases derivadas(clases hijas) Si definimos un método como final no puede ser redefinido en una clase derivada

- **Práctica 13:** Toma capturas de pantalla modificando Persona para que no sea abstracta y establécela a final. Muestra en captura de pantalla los errores en Hombre y Mujer. Quita el modificador final de Persona pero ponerlo en calcularPesoIdeal() Tratar de redefinirlo en Hombre o en Mujer. Tomar captura de pantalla del error ¿ qué dice el mensaje?

Polimorfismo

El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, ...).

Conseguimos implementar polimorfismo en jerarquías de clasificación que se dan a través de la herencia. Por ejemplo, tenemos una clase vehículo y de ella dependen varias clases hijas como coche, moto, bus, etc.



Si seguimos el ejemplo anterior podríamos tener un apuntador de la clase Vehículo (una variable de tipo Vehículo) que podría soportar objetos (instancias) de Coche Moto Bus

Lo primero que debemos entender es que podemos crear un apuntador (variable) de una clase aunque sea **abstracta**

```
abstract class Vehiculo{
    ...
}

public class Polimorfismo {
    public static void main(String[] args) {
        Vehiculo apuntador;
    }
}
```

Observar que **NO INSTANCIAMOS NINGÚN OBJETO** simplemente definimos una variable del tipo Vehiculo. Si tratáramos de crear una instancia generaría un error

Lo segundo que debemos entender es que en el mundo real si un objeto es de tipo Moto también es un Vehiculo. De esa forma podemos usar el apuntador anterior hacia una moto:

```
Vehiculo apuntador=new Moto("xmn-2532",230);
```

Veamos un ejemplo completo:

```
abstract class Vehiculo{
    public String matricula;
    public double cilindrada;

    public Vehiculo(String matricula, double cilindrada){
        this.matricula = matricula;
        this.cilindrada = cilindrada;
    }
}

class Moto extends Vehiculo{
    public boolean sidecar;
    public Moto(String matricula, double cilindrada) {
        super(matricula, cilindrada);
        sidecar=false;
    }
}

class Bus extends Vehiculo{
    public int asientos;
    public Bus(String matricula, double cilindrada,int asientos) {
        super(matricula, cilindrada);
        this.asientos=asientos;
    }
}
```



```

public class Polimorfismo {
    public static void main(String[] args) {
        Vehiculo array[] = new Vehiculo[2];
        Moto moto=new Moto("xmn-2532",230);
        Bus bus = new Bus("jtr-8932",3200,40);
        Vehiculo vehiculo = moto;
        System.out.println(vehiculo.getClass());
        array[0]=vehiculo;
        vehiculo = bus;
        array[1]=vehiculo;

        for(Vehiculo elemento : array){
            if(elemento instanceof Moto){
                Moto m = (Moto)elemento;
                System.out.println("tiene sidecar: " + m.sidecar);
            }
        }
    }
}

```

En el ejemplo anterior hemos marcado 3 líneas en amarillo porque precisan explicación adicional.

La primera de ellas:

```
Vehiculo array[] = new Vehiculo[2];
```

¿ si no podemos instanciar objetos de una clase abstracta cómo es que creamos un array de Vehiculo ?

Lo cierto es que en el array lo que se van a guardar son apuntadores, realmente con la sentencia anterior el: **new** se hace sobre un array y adicionalmente estamos informando que tendrá apuntadores del tipo Vehiculo, pero eso, apuntadores, NO INSTANCIAS PORQUE ESO NO ES POSIBLE

La segunda sentencia:

```
if(elemento instanceof Moto){
```

vemos una nueva palabra clave: **instanceof** aprovechamos para definirla:

instanceof

Cuando utilicemos el operador instanceof, debemos recordar que sólo puede usarse con variables que contengan la referencia a un objeto. Es decir, variables que contendrán un conjunto de bytes que representarán a la dirección en memoria en la que está almacenado el objeto.

El objetivo del operador instanceof es conocer si un objeto es de un tipo determinado. Por tipo nos referimos a clase o interfaz (interface),

La forma de uso general es:

nombreObjeto instanceof NombreClase

Nos devuelve true o false según el objeto pertenezca o no a la clase. Observar que por el mismo motivo que antes decíamos que si es una Moto también es un Vehiculo, si preguntáramos por:

moto instanceof Vehiculo

nos devolvería true

La tercera sentencia:

```
if(elemento instanceof Moto){  
    Moto m = (Moto)elemento;
```

Observar que una vez que hemos comprobado que el apuntador de tipo Vehiculo, que hemos llamado elemento es realmente una Moto entonces **hacemos un cast para poder usar toda la funcionalidad que tiene una Moto**, porque de otra forma únicamente podríamos acceder a la información que es común con Vehiculo, NO la exclusiva de una Moto. Pensar al respecto de eso un momento: un Vehiculo como tal no tiene por qué poder llevar un sidecar sin embargo una moto si debiera tener esa capacidad. Es por eso por lo que Java **no permite acceder a los atributos y métodos de la clase heredada (Moto) desde un apuntador de tipo del padre (Vehiculo)**

Así pues: No se puede acceder a los **miembros específicos** de una **subclase** a través de una **referencia** a una **superclase**. Sólo se pueden utilizar los miembros declarados en la **superclase**, aunque la definición que finalmente se utilice en su ejecución sea la de la **subclase**.

● **Práctica 14:** Pon el código del ejemplo en el IDE y tratar de acceder directamente desde elemento (la variable de tipo Vehiculo del ejemplo) al atributo propio de Moto: Ej. elemento.sidecar = false;
Tomar captura de pantalla que muestre el error ¿qué dice el IDE?
Prueba a escribir en el IDE: elemento.
Observa las opciones que te permite. Te aparecerá sidecar, seleccionala y toma captura de pantalla del código que te haya generado ¿ qué es lo que ha hecho el IDE ?

● **Práctica 15:** Toma las clases Persona, Hombre, Mujer de la práctica 12 y crea un pequeño programa que le de la posibilidad al usuario de introducir tantos hombres y mujeres como quiera. Esos datos se insertarán en un ArrayList de tipo Persona. Finalmente recorrer el ArrayList y mostrar el toString() de cada elemento. ¿ qué toString() está usando ? ¿ el de Persona, el de Hombre, el de Mujer ?

● **Práctica 16:** Crear la clase ProfesionalBaloncesto que representa a cualesquier profesional de baloncesto. Así tiene como atributos: nombre, apellidos, edad, ingresosAnuales, numeroFederado, con un constructor como mínimo y un toString()
La clase Jugador que extiende a ProfesionalBaloncesto con atributos propios: posición (enum para base, pivot,..) , partidosJugados , equipo
La clase Entrenador que extiende a ProfesionalBaloncesto con atributos propios: porcentajeVictorias (un número de 0 a 100)
La clase Arbitro que extiende a ProfesionalBaloncesto con atributos propios: int faltasPitadas
Todos los datos de los atributos propios hacen referencia a una temporada completa
Poner en una LinkedList de tipo ProfesionalBaloncesto objetos de tipo Jugador, Entrenador, Arbitro. Recorrer la lista y mostrar **únicamente** los atributos propios de la clase de cada objeto (no los atributos comunes con la superclase)

Interfaces

Nota: Ya hemos usado en su momento sin entrar en detalle los interface cuando escribíamos: `implements Comparable`, `implements Comparator` en la UT de Estructuras de Almacenamiento

Un ejemplo bastante nítido de interfaz para objetos sin una relación clara de herencia lo podemos tener en el interfaz: `Ordenable`

podemos ordenar fracciones y podemos ordenar clientes por orden alfabético de su nombre y son clases muy dispares entre sí

Pareciera que debiera haber algún mecanismo que no fuera el de herencia para implementar lo anterior. Que tuviéramos unos nombres de métodos que fueran conocidos y comunes para todas las clases que queremos ordenar sin que tuvieran una clara relación de herencia entre ellas.

Un ejemplo más completo:

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieres que lleven a cabo están relacionadas con el hecho de que algunos animales sean depredadores (por ejemplo: observar una presa, perseguirla, comérsela, etc.) o sean presas (observar, huir, esconderse, etc.). Si creas la clase `León`, esta clase podría tener lo que corresponde a depredador. Mientras que otras clases como `Gacela` tendría las acciones de Presa. Por otro lado, podrías tener también el caso de la clase `Rana`, que implementaría las acciones de Depredador (ya que caza pequeños insectos), pero también la de Presa (ya que puede ser cazado). Adicionalmente `León` hereda de `Felino`, `Gacela` de `Bóvido` y `Rana` de `Anfibio`

Se diría que nos interesa poder asignar/establecer esas funcionalidades de Depredador indistintamente de la herencia. Por de pronto en Java sabemos que no es posible heredar de dos clases y como vimos `León` ya hereda de `Felino`, ya no sería posible pues que heredara de `Depredador`. Por otro lado también nos pudiera interesar disponer de un artificio para utilizar conjuntos de acciones/métodos sin que haya una relación de herencia de por medio.

Lo anterior se puede modelar mediante interface. En su origen interface establecía un conjunto de **métodos abstractos y atributos constantes**. Posteriormente se incluyó la posibilidad de métodos default y static. Veremos de momento la idea original de interface para más adelante en el tema introducir las otras variantes entendiendo la razón del cambio.

Una interfaz consiste principalmente en una lista de declaraciones de métodos sin implementar, que caracterizan un determinado comportamiento. Si se desea que una clase tenga ese comportamiento, tendrá que implementar esos métodos establecidos en la interfaz. En este caso no se trata de una relación de herencia (la clase A es una especialización de la clase B, o la subclase A es del tipo de la superclase B), sino más bien una relación "de implementación de comportamientos" (la clase A implementa los métodos establecidos en la interfaz B, o los comportamientos indicados por B son llevados a cabo por A; pero no que A sea de clase B).

Existe un gran **parecido formal** entre una **clase abstracta** y una **interfaz**, pudiéndose en ocasiones utilizar indistintamente una u otra para obtener un mismo fin. Pero, a pesar de ese gran parecido, existen algunas diferencias, no sólo formales, sino también conceptuales, muy importantes:

- ❑ **Una clase no puede heredar de varias clases**, aunque sean abstractas (**herencia múltiple**). Sin embargo sí puede **implementar una o varias interfaces** y además seguir heredando de una clase.
- ❑ **Una interfaz puede hacer que dos clases tengan un mismo comportamiento independientemente de sus ubicaciones en una determinada jerarquía de clases** (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).
- ❑ **Las interfaces tienen su propia jerarquía**, diferente e independiente de la jerarquía de clases.

Prototipo:

```
[public] interface <NombreInterfaz> {  
    [public] [final] <tipo1> <atributo1>= <valor1>;  
  
    [public] [final] <tipo2> <atributo2>= <valor2>;  
  
    ...  
  
    [public] [abstract] <tipo_devuelto1> <nombreMetodo1>([lista_parámetros]);  
  
    ...  
}
```

Observar que en el prototipo aparecen los métodos terminados en “;” al igual que con los métodos abstractos. Ese es el modelo “normal” para los métodos de un interface.

Poner public final en los atributos es opcional, lo pongamos o no, serán de ese tipo

Ejemplo

```
public interface Depredador {  
  
    void localizar (Animal presa);  
    void cazar (Animal presa);  
    ...  
}
```

Implementación de interface

```
class NombreClase implements NombreInterfaz1, NombreInterfaz2, ... {  
    ...  
}
```

Podemos observar que podemos poner varios interface separados por coma. Eso permitiría ver algunas clases del tipo:


```
class Rana extends Anfibio implements Depredador, Presa {  
    ...  
}
```

En el ejemplo anterior vemos que podemos heredar de una clase (extends Anfibio) e implementar varias interface (implements Depredador, Presa)

Como ejemplo vamos a crear la interfaz Ordenable con los métodos boolean: esMayorQue() esMenorQue() y esIgualQue()

```
interface Ordenable{
    boolean esMayorQue(Ordenable elemento);
    ...
}
```

Observar que le decimos que recibe un objeto que implementa la interfaz Ordenable. Aquí también tenemos polimorfismo ya que tomamos un apuntador de algo que NO es instanciable (nosotros no podemos hacer: new Ordenable())

 **Práctica 17:** Crear la interfaz: Ordenable e incluir como métodos: boolean esMayorQue() , esMenorQue(), esIgualQue() Implementar el interfaz en la clase Persona (tomaremos la clase abstracta que ya hemos usado en esta unidad) ¿ qué ocurre en el IDE cuándo escribimos: Persona implements Ordenable ? (tomar captura de pantalla) mediante alt+enter hacer el override

Aún no hemos establecido el criterio de ordenación así que la anterior práctica nos queda incompleta. Utilizaremos el criterio de ordenación alfabética. Para ello nos apoyaremos en el método que tienen los String: compareTo() que devuelve mayor que cero si el String actual es mayor que el String dado:

```
@Override
public boolean esMayorQue(Ordenable elemento) {
    Persona p=(Persona)elemento;
    return this.nombre.compareTo(p.nombre) > 0;
}
```

Observar que en el ejemplo anterior hemos tenido que hacer un cast ya que lo único que teníamos de información del objeto elemento es que implementaba Ordenable. Así que no podríamos acceder a la información de su nombre. Si recordamos cuando veíamos Polimorfismo una superclase podía únicamente acceder a los métodos que eran comunes con la subclase. La única forma era mediante un cast a la subclase para acceder al resto de métodos

- **Práctica 18:** Implementar en Persona correctamente los tres métodos de Ordenable mediante compareTo() . Utilizar el siguiente pequeño programa para probarlo:

```
public class PruebaInterfaz{
    public static void main(String[] args) {
        Ordenable elemento1 = new Hombre("unnom", "unape", "otro", 25, 178, 75);
        Ordenable elemento2 = new Mujer("otronom", "ape1", "ape2", 20, 165, 60);
        if( elemento1.esMayorQue(elemento2))
            System.out.println("elemento1 es mayor que elemento2");
        else
            System.out.println("elemento1 no es mayor que elemento2");
    }
}
```

- **Práctica 19:** Crear la clase Fraccion con atributos: int numerador, int denominador un constructor que recibe los dos paramentros y otro que recibe únicamente el numerador (el denominador será 1) con métodos: double toDouble() que devuelve el resultado de la división y un toString() en el formato: 3/5 donde 3 sería el numerador y 5 el denominador. Fraccion implementará la interfaz Ordenable los métodos: esMayorQue() ... utilizarán la comparación de los double resultantes al hacer la división para comparar dos fracciones, siendo mayor una fracción si el double resultante es mayor

- **Práctica 20:** Crear un pequeño programa con un array de Ordenable:
Ordenable array[] = new Ordenable[2];
donde apuntemos a un Hombre o Mujer y a una Fraccion
Recorrer el array y mediante un condicional con instanceof haremos que:
si el elemento es de tipo Fraccion se comparará con la Fraccion: 4/5 y se mostrará cuál es mayor en pantalla.
Si el elemento es Persona se comparará con la Mujer: Mujer("Ana","Po","Ro",20,165,60)
y se mostrara cuál es mayor en pantalla

Clases Anónimas

En algunas ocasiones precisamos crear un objeto que tenga determinado comportamiento pero realmente es algo puntual y si creáramos la clase correspondiente quizás no la volviéramos a utilizar. Ese podría ser un caso para una clase anónima.

Una clase anónima es una clase sin nombre, definida en la misma línea de código donde se crea el objeto de la clase. Esta operación se lleva a cabo en el interior de un método de otra clase, por ello la clase anónima es considerada como una clase interna anidada.

Una clase anónima siempre debe ser una subclase de otra clase existente o bien de implementar alguna interfaz

Un ejemplo donde sería útil una clase anónima puede darse cuando queremos utilizar la interfaz `Comparator<>`. Veamos el código que teníamos de ejemplo en la UT Estructuras de Almacenamiento para ordenar la clase `Persona` (la hemos modificado a `Mujer` porque en esta unidad `Persona` es abstracta):

```
class ComparadorMujeres implements Comparator<Mujer>{  
  
    @Override  
    public int compare(Mujer p1, Mujer p2) {  
        return Integer.compare(p1.getEdad(), p2.getEdad());  
    }  
}
```

```
ArrayList<Mujer> mujeres = new ArrayList<>(3);  
mujeres.add(new Mujer("Marta", "León", "León", 25, 180, 60));  
mujeres.add(new Mujer("Julia", "Luz", "Luz", 20, 160, 55));  
mujeres.add(new Mujer("Pilar", "Ramos", "Ramos", 29, 165, 58));  
  
Collections.sort(mujeres, new ComparadorMujeres());  
for(Mujer p: mujeres)  
    System.out.println(p);
```

Tuvimos que crear una clase `ComparadorMujeres` únicamente para pasarle un objeto a `Collections.sort()` realmente no necesitábamos una clase más. Lo único que necesitábamos era poder implementar el método: `compare()` que precisa `Collections.sort()` para hacer la ordenación

Podemos crear una clase anónima así:

```
NombreInterfaz var = new NombreInterfaz(){  
    ...  
};
```

Observar que **var** recibe un apuntador a un nuevo objeto de una clase desconocida pero que implementa la interfaz: NuevaInterfaz. También debemos observar que acabamos con: “;” ya que realmente lo que estamos haciendo es una sentencia new

Nota: Se ha puesto el prototipo aludiendo a un interfaz pero podría ser también una superclase

Veamos un ejemplo. Nos han pedido que usemos una interfaz: ImprimirEnMayuscula que la utilizamos para que las clases que la implementen tenga un método llamado: imprimirMayuscula() que nos garantice que la salida en pantalla va a ser toda en mayúsculas. Una solución sería modificar la clase Mujer e implementar la interfaz ¿pero y si no podemos modificar la clase Mujer porque, por ejemplo, está en muchos sitios y obligaría a cambiar muchos programas ?

Otra solución pasaría por usar un “wrapper” que implemente la interfaz y que envuelva al objeto Mujer.

Para ello creamos un objeto de una clase nueva que incorpore la interfaz ImprimirEnMayuscula, así no modificaríamos la clase Mujer. Claro que esa nueva clase no va a tener mayor interés una vez lo hayamos usado. Luego es un candidato para crear un objeto desde una clase anónima:

```
Mujer nuevaMujer=new Mujer("Isabel","Dorta","Jiménez", 45, 165, 60);  
ImprimirEnMayuscula<Mujer> iem=new ImprimirEnMayuscula<Mujer>() {  
    @Override  
    public void imprimirMayuscula(Mujer mujer) {  
        String val = mujer.toString();  
        System.out.println(val.toUpperCase());  
    }  
};  
  
iem.imprimirMayuscula(nuevaMujer);  
ArrayList<ImprimirEnMayuscula> imprimibles = new ArrayList<>();  
imprimibles.add(iem);
```

- **Práctica 21:** Crear la interfaz `ImprimirEnMayuscula<T>` que incluya el método: `void imprimirMayuscula(T objeto)` y probarlo con el código anterior. Tomar captura de pantalla de ese trozo de código y la ejecución

Con lo anterior hemos conseguido crear un objeto que implementa la interfaz como se nos pedía y que puede imprimir en mayúsculas todos los objetos `Mujer` que se le pasen. Ahora el objeto `iem` puede estar en una lista de `ImprimirEnMayuscula` junto con otros objetos que implementen la interfaz. Pero cuando lo usemos imprimirá `Mujer` como queríamos

Todavía mejor que el anterior ejemplo es modificar el que tenemos de la UT estructuras de almacenamiento que necesita un objeto que implemente la interfaz `Comparator<Mujer>` y así vemos el caso de una necesidad real: poder ordenar una lista de `Mujeres` mediante `Collections.sort()`

```
ArrayList<Mujer> mujeres = new ArrayList<>(3);
mujeres.add(new Mujer("Marta", "León", "León", 25, 180, 60));
mujeres.add(new Mujer("Julia", "Luz", "Luz", 20, 160, 55));
mujeres.add(new Mujer("Pilar", "Ramos", "Ramos", 29, 165, 58));

Comparator<Mujer> cm= new Comparator<Mujer>() {
    @Override
    public int compare(Mujer o1, Mujer o2) {
        return Integer.compare(o1.getEdad(), o2.getEdad());
    }
};

Collections.sort(mujeres, cm);
for(Mujer p: mujeres)
    System.out.println(p);
```

- **Práctica 22:** Crear un `ArrayList` con objetos `Fraccion` ordenarlos utilizando una clase anónima que implemente `Comparator<Fraccion>`

- **Práctica 23:** Crear un `ArrayList` con objetos `Jugador` (práctica 16) y ordenarlos utilizando una clase anónima que implemente `Comparator<Jugador>` la comparación será mediante el atributo `partidosJugados`

lambdas

Este apartado tiene lugar con un cambio en Java que abre la puerta hacia la programación funcional. Hasta ahora hemos visto un lenguaje muy orientado a objetos. El motivo de incluirlo en este tema es por el uso de las interfaces funcionales. Que como tales interfaces corresponde al estudio de herencia. Dejando aparte la conveniencia o no de este cambio trabajar la siguiente sección nos permite ver posibilidades que estaban cerradas en Java pero que estaban disponibles en otros lenguajes.

Pensemos en lo último que hemos visto de clases anónimas Les encontramos especial utilidad porque nos permitía utilizar las ventajas de Collections, por ejemplo Collections.sort() Ej.

```
Comparator<Mujer> cm= new Comparator<Mujer>() {  
    @Override  
    public int compare(Mujer o1, Mujer o2) {  
        return Integer.compare(o1.getEdad(), o2.getEdad());  
    }  
};  
Collections.sort(mujeres, cm);
```

Pero ¿qué es lo que realmente se está persiguiendo pasar a Collections.sort() con el código anterior?: Le estamos pasando una **función** de comparación para que Collections.sort() tenga una forma para comparar dos Mujer. Observar que aquí el detalle está en que en lugar de pasarle un número o similar que es lo que hacemos cuando pasamos un parámetro normal lo que le queremos enviar realmente es una función

Hemos armado un poco de lío para poder conseguir pasarle una función a Collections.sort() En una primera aproximación, crear un clase nueva. En una segunda aproximación crear una clase anónima para no estar creando tropecientas clases que no van a tener otro uso.

Observar un código que era válido con la tecnología de hace ya muchos años, mediante el lenguaje C (recordar que Java se inspira mucho en C, C++ ya que su sintaxis deriva de ellos)

```
typedef int (*opBinaria)(int, int);
int suma(int a, int b)
{
    return a + b;
}
int multiplicacion(int a, int b) {
    return a * b;
}
void ejecuta(opBinaria f, int op1, int op2)
{
    int resultado = f(op1, op2);
    printf("%d\n", resultado);
}
int main()
{
    ejecuta(suma, 1, 2);
    ejecuta(multiplicacion, 1, 2);
    return 0;
}
```

El código anterior te da la posibilidad de pasar funciones: suma, multiplicación como parámetro. Cuando conoces de las ventajas de algo no aceptas con agrado prescindir de ello. Es normal que muchos programadores echaran de menos en Java **poder pasar funciones como parámetros**.

En un lenguaje como Java, que seguía de forma tan estrecha la Programación Orientada a Objetos, las variables sólo podían contener datos. Así pues, no es posible declarar un método que acepte como parámetro una función.

La solución que se encontró fue: **el patrón estrategia**.

Tal patrón ya lo hemos visto cuando creamos un objeto de una clase anónima que incorpore la función que queremos enviar como parámetro y pasar el objeto como parámetro. Así se mantenía el mantra del envío únicamente de datos por parámetro y pasar lo que se quería: **una función**

Sigamos con nuestro caso particular de clase Mujer y Collections.sort() ¿ qué ocurre si queremos ordenar por más criterios además de edad ? Hay que crear otra clase anónima:

Ejemplo: Por nombre

```
Comparator<Mujer> cmNombre= new Comparator<Mujer>() {  
    @Override  
    public int compare(Mujer o1, Mujer o2) {  
        return Integer.compare(o1.getNombre(), o2.getNombre());  
    }  
};  
Collections.sort(mujeres, cmNombre);
```

¿ Y si luego queremos ordenar por otro criterio ? Otra clase anónima. Y luego otra y otra,...

Hubo un momento en que se decidió sacrificar la idea de POO de únicamente pasar datos como parámetros y **surgieron las expresiones lambda**:

Una expresión lambda es un bloque de código que representa a una función.

El uso de una expresión lambda (y las interfaces funcionales) reduce la necesidad de clases anónimas:

```
Comparator<Mujer> comp = (Mujer m1, Mujer m2)-> {  
    return Integer.compare(m1.getEdad(), m2.getEdad());  
};  
Collections.sort(mujeres, comp);
```

En la primera línea hemos usado una interfaz funcional para asignar la expresión lambda (ya veremos más adelante las interfaces funcionales)

Veamos el mismo código pasando directamente la lambda como parámetro:

```
Collections.sort(mujeres,  
    (Mujer o1, Mujer o2) -> {  
        return Integer.compare(o2.getEdad(), o1.getEdad());  
    }  
);
```

Mirando el código anterior, en el segundo parámetro que le pasamos a Collections.sort() le estamos enviando una función ¡¡ya no es un dato, es una función!! que dice que maneja dos objetos Mujer y devuelve la comparación mediante Edad

Lo anterior se puede simplificar más, siempre que el tipo de parámetro lo pueda inferenciar el compilador y lo que hace la función sea una sola línea (no es válido para más líneas)

```
Collections.sort(mujeres, (o1, o2) -> o2.getEdad() - o1.getEdad());
```

Realmente nos permite una escritura mucho más compacta y limpia.

Prototipo de las expresiones lambda

Definición: **Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase.**

Las expresiones lambda son una forma de crear funciones anónimas y que puedes utilizar en dónde el parámetro recibido sea una interfaz funcional (más adelante vemos la interfaz funcional)

Sintaxis:

(parámetros) -> { cuerpo-lambda }

1. El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.
2. Parámetros:
 - Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
 - Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
3. Cuerpo de lambda:

- Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula return en el caso de que deban devolver valores.
- Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la cláusula return en el caso de que la función deba devolver un valor .

Ejemplos de expresiones lambda:

- `() -> 5` // directamente devuelve un valor, return 5
- `x -> 2 * x` // duplica el valor de x y lo retorna
- `(x, y) -> x - y` // toma dos valores y retorna su diferencia
- `(int x, int y) -> x + y` // toma dos enteros y retorna su suma
- `(String s) -> System.out.print(s)` // toma un string y lo imprime en la consola

Bien, hemos visto un caso de uso de una expresión lambda mediante Collections.sort() También hemos visto como es el prototipo y algunos ejemplo. En varias ocasiones hemos nombrado durante esas explicaciones las interfaces funcionales. La relación que tienen es: Las expresiones lambda son funciones anónimas que puedes utilizar en donde el parámetro sea una interfaz funcional

Pero ¿qué es una interfaz funcional? Veámoslo:

Interfaz funcional

Las interfaces funcionales son interfaces con la característica de contar con un solo método abstracto.

También pueden contar con más de un método abstracto siempre y cuando estos métodos sobrescriban métodos de la clase Object o sean métodos predeterminados, default o sean métodos static.

Los métodos predeterminados (marcados con "default") no son abstractos, por lo cual una interfaz funcional puede definir varios métodos predeterminados.

Que una interfaz incorpore métodos por defecto (con código) tiene miga... **Fijarse que Java no permitía la herencia multiple, pero sí que permitía tener varias interface Así que de alguna manera se está introduciendo la herencia múltiple en Java**

Importante: Se ha creado una anotación específica: **@FunctionalInterface** que le dará la instrucción al compilador para que verifique que la interfaz anotada cumple con los requisitos de una interfaz funcional.

Como la mejor forma de ver las cosas es mediante ejemplos. Pongamos uno:

```
@FunctionalInterface
interface Operaciones<T>{
    String operacion(T a, T b);
}

public static void mostrarResultado(int x, int y, Operaciones op){
    System.out.println(op.operacion(x,y));
}

public static void main(String[] args) {
    Operaciones<Integer> suma = (a,b) -> "x + y=" + (a+b);
    Operaciones<Integer> resta = (a,b) -> "x - y=" + (a-b);
    Operaciones<Integer> multiplica = (a,b) -> "x * y=" + (a*b);
    mostrarResultado(2,3,suma);
    mostrarResultado(5,1,resta);
    mostrarResultado(4,7,multiplica);
}
```

Observar que este ejemplo es muy parecido al que pusimos al principio en lenguaje C

Por último decir que las interfaces funcionales pueden aceptar tanto una expresión lambda como una referencia a un método ya existente. Esa referencia la denotaremos mediante: “::” Ej.

```
Function<String, Integer> funcion1 = Integer::parseInt;
int valor1 = funcion1.apply("20");
```

Nos falta ver dos conceptos introducidos en interface: default method, static method

Ambos tipos de métodos permiten introducir código dentro del interface y ese código será pues parte de cualquier clase que implemente el interface. De alguna forma ya se establece una especie de “herencia múltiple” en java. **La única diferencia entre estos dos tipos de métodos es que los static no pueden ser sobrescritos por las clases que implementen la interface mientras que los default sí se puede**

Ahora bien, ¿ qué ocurre si dos interface tienen la misma firma(tiene el mismo nombre y recibe los mismos parámetros) para un método y una clases los implementa a ambos ? pues que nos veremos obligados a elegir que método default queremos utilizar:

```
public interface Bicho {
    default void decirHola(){
        System.out.println("Hola bicho!!!!");
    }
}

public interface Planta {
    default void decirHola(){
        System.out.println("Hola planta!!!!");
    }
}

public class Chlorotica implements Bicho, Planta {
    public void decirHola(){
        Bicho.super.decirHola();
    }
}
```

● **Práctica 23:** Copiar y ejecutar el Ejemplo Operaciones. Ya que las interfaces funcionales aceptan métodos con código (métodos default) agregar a la interfaz un método llamado: default public void miNombre() que muestre tu nombre. En el main escribir la sentencia: suma.miNombre(); y ejecutar el código ¿funciona?
Crear otro método default llamado: misApellidos() que en este caso mostrarían tus apellidos ¿puedes ejecutarlo? Entonces ¿ se puede tener más de un método default?

● **Práctica 24:** agrega el método: public abstract String toString(); ¿ lo acepta ? ¿cómo es que lo acepta si únicamente se podía un único método abstracto?
Ahora añade el método: public abstract int otroAbstracto(); ¿lo acepta? (toma captura de pantalla) ¿qué error muestra? Comenta la línea con: @FunctionalInterface ¿sigue quejándose? ¿por qué lo acepta ahora?

● **Práctica 26:** Crear la interfaz funcional: Mates que tiene el método:
public abstract int calc(int x);
En el main crear una lambda para obtener el número al cuadrado, otra para calcular el factorial. Crea un método parecido al anterior: mostrarResultado(int x, Mates func) que nos muestre en pantalla el resultado. Pasarle tanto el factorial como el cuadrado

- **Práctica 28:** Crear un ArrayList de Personas que incluya objetos Hombre y Mujer. Crea una variable:

Comparator<Persona> cmtPersona;

asigna a cmtPersona una expresión lambda que ordena las personas por peso (igual tienes que agregar un getter para peso en Persona) y luego usa Collections.sort() para ordenar mediante cmtPersona el ArrayList

- **Práctica 30:** Modifica el ejercicio 20 de la UT de estructuras de almacenamiento para que el criterio de ordenamiento de las Matriz2x2 se establezca mediante una expresión lambda

- **Práctica 31:** Modifica el ejercicio 21 de la UT de estructuras de almacenamiento para que el criterio de ordenamiento de lTelegrama se establezca mediante una expresión lambda

- **Práctica 32:** Modifica el ejercicio 23 de la UT de estructuras de almacenamiento para que el criterio de ordenamiento de VectorLibre se establezca mediante una expresión lambda

Streams

Los stream corresponderían en el tema de estructuras de almacenamiento. Sin embargo por su estrecha relación con los interfaces funcionales y los lambdas los trabajaremos en esta unidad

Un stream es una "secuencia de elementos de un origen que admite operaciones concatenadas".

Ahora desglosemos la definición:

- **Secuencia de elementos:** Un stream brinda una interfaz para un conjunto de valores secuenciales de un tipo de elemento particular. No obstante, **los streams NO almacenan elementos**; estos **se calculan cuando se recibe la solicitud** correspondiente.
- **Origen:** Los streams toman de un origen de datos, como colecciones, matrices o recursos de E/S.

- **Operaciones concatenadas:** Los streams admiten operaciones estilo SQL y operaciones comunes a la mayoría de los lenguajes de programación funcionales, como filter, map, reduce, find, match y sorted, entre otras

Más aún, las operaciones de los streams tienen dos características fundamentales que las distinguen de las operaciones con colecciones:

- **Estructura de proceso:** Muchas operaciones de stream devuelven otro stream. Así, es posible encadenar operaciones para formar un proceso más abarcador. Esto, a su vez, permite lograr ciertas optimizaciones.
- **Iteración interna:** A diferencia del trabajo con colecciones, en que la iteración es explícita (*iteración externa*), las operaciones del stream llevan a cabo la iteración tras bambalinas

Antes de continuar veamos un ejemplo para saber a qué queremos llegar. Basándonos en el ArrayList de la práctica 29:

```
ArrayList<String> resultados;  
resultados =(ArrayList)personas.stream().filter(x->x.getEdad()>=18)  
    .sorted((p1,p2)->p1.nombre.compareTo(p2.nombre))  
    .map(p->p.nombre)  
    .collect( Collectors.toList());
```

Hemos convertido el ArrayList en un Stream, luego lo hemos filtrado para tomar únicamente a los adultos, después los ordenamos por nombre y finalmente lo hemos vuelto a convertir en una lista

Todas las operaciones, a excepción de collect, devuelven un Stream, por lo que es posible encadenarlas y formar un proceso, que puede verse como consulta respecto de los datos del origen

En realidad **no se lleva a cabo ninguna tarea hasta que se invoca la operación collect**. Esta última comenzará a abordar el proceso para devolver un resultado

Streams vs. colecciones

Tanto la noción de colecciones que ya existía en Java como **la nueva noción de streams se refieren a interfaces con secuencias de elementos**. Entonces, ¿cuál es la diferencia? En resumen, las colecciones hacen referencia a datos mientras que los streams hacen referencia a cómputos.

Pensemos por ejemplo en una película almacenada en un DVD. Se trata de una colección (de bytes o de fotogramas; precisarlo no es importante para el ejemplo) porque contiene toda la estructura de datos. Ahora imaginemos el mismo video, pero esta vez lo reproducimos desde Internet. En este caso hablamos de un stream (de bytes o fotogramas). El reproductor de video por secuencias (*streaming*) necesita descargar solo unos pocos fotogramas más allá de los que está viendo el usuario; así, es posible comenzar a mostrar los valores del comienzo del stream antes de que la mayor parte del stream se haya computado (la transmisión de secuencias o streaming puede pensarse como un partido de fútbol en vivo).

En términos simples, la diferencia entre las colecciones y los streams se relaciona con *cuándo* se hacen los cómputos.

Las operaciones de Streams que pueden conectarse entre sí se llaman *operaciones intermedias*. Se pueden conectar porque la salida que devuelven es de tipo Stream. Las operaciones que cierran un proceso de stream se llaman *operaciones terminales*. A partir de un proceso producen un resultado de tipo List, Integer o incluso void (de tipos distintos de Stream).

¿Por qué es importante la distinción? Bien, las operaciones intermedias no llevan a cabo tareas de procesamiento hasta que se invoca una operación terminal en el proceso del Stream; son "perezosas". Eso se debe a que a menudo **la operación terminal puede "fusionar" y procesar diversas operaciones intermedias en una sola acción**.

Observar lo último que hemos dicho... NO se ejecutan hasta que llega una operación terminal

Observar el siguiente código:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> lista = numeros.stream()
    .filter(n -> {
        System.out.println("filtrando " + n);
        return n % 2 == 0;
    })
    .map(n -> {
        System.out.println("mapping " + n);
        return n * n;
    })
```

```
.limit(2)
.collect(toList());
```

● **Práctica 33:** Probar el código anterior. Agrega el código para mostrar lista que se obtuvo como resultado en pantalla Toma captura de pantalla del código y la ejecución

`limit(2)` hace que sólo se procese el Stream hasta que se obtengan dos resultados. Así únicamente precisa los primeros 4 elementos ya que con ellos se obtienen dos números pares: 2, 4 y finalmente se transforma al cuadrado: 4, 16

Veamos ahora algunas operaciones que pueden usarse con streams. Pueden consultarse el listado completo en la interfaz `java.util.stream`.

Filtrado. Diversas operaciones pueden usarse para filtrar elementos de un stream:

- *filter(Predicate)*: Toma un predicado (`java.util.function.Predicate`) como argumento y devuelve un stream que incluye todos los elementos que coinciden con el predicado indicado.
- *distinct*: Devuelve un stream con elementos únicos (según sea la implementación de `equals` para un elemento del stream).
- *limit(n)*: Devuelve un stream cuya máxima longitud es `n`.
- *skip(n)*: Devuelve un stream en el que se han descartado los primeros `n` números.

Acabamos de nombrar **Predicate** que es una interfaz funcional con un único método llamado: `boolean test()`

```
Predicate<Persona> predicadoNombre = new Predicate<Persona>() {
    @Override
    public boolean test(Persona p) {
        return p.getNombre().equals("pepe");
    }
};
```

Lo anterior usando una lambda nos queda:

```
Predicate<Persona> predicadoNombre = p -> p.getNombre().equals("pepe");
```

En el ejemplo que usamos en la práctica 33 ya usamos un Predicate por medio de una lambda que hacía un println() y seleccionaba los números pares

Búsquedas e identificación de coincidencias. Un patrón común en el procesamiento de datos consiste en determinar si algunos elementos se ajustan a una propiedad dada. Es posible usar las operaciones

- *anyMatch(Predicate)*
- *allMatch(Predicate)*
- *noneMatch(Predicate)*

Todas toman como argumento un predicado y devuelven un valor boolean (es decir que son operaciones terminales). Por ejemplo, se puede usar allMatch para verificar que todos los elementos de un stream de personas tengan todos más de 18 años

```
boolean adultos = personas.stream().allMatch(t -> t.getEdad() >= 18);
```

● **Práctica 34:** haciendo uso de anyMatch(), noneMatch() obtener true si para el ArrayList personas hay algún objeto de tipo Mujer.
Obtener true si ninguna persona tiene un peso anormalmente bajo (menor de 50kg)

Adicionalmente se dispone de:

- *Optional<T> findAny()*
- *Optional<T> findFirst()*

Optional<T> es una clase contenedora que representa la existencia o ausencia de un valor. Mediante findFirst() y findAny() son terminales (y por tanto se procesa ya el stream) para devolver el primer objeto (si hubiera) o uno cualesquiera (si hubiera). El motivo para devolver Optional es precisamente por ese “hubiera” Mediante: ifPresent() toma el dato y hace con él lo que queramos: Ej.

```
List<Integer> numericos = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);  
Optional<Integer> opcional=numericos.stream().filter(x->x>3 && x<6).findFirst();  
opcional.ifPresent(System.out::println);
```

Observar: `System.out::println` Como se dijo al finalizar las interfaces funcionales, éstas aceptan tanto lambdas como referencias a métodos existentes usando el operador: “::”

● **Práctica 35:** Utiliza `findFirst()` y `Optional.ifPresent()` para mostrar a un Hombre llamado: "Luis" que debes primero agregar al `arraylist` personas.

Mapecto. Los streams admiten el método `map`, que emplea una función (`java.util.function.Function`) como argumento para proyectar los elementos del stream en otro formato. La función se aplica a cada elemento, que se "mapea" o asocia con un nuevo elemento. Se debe de entender que con el mapeo ya no estaremos trabajando con un stream de los objetos que empezamos, ahora será de otra cosa

```
List<String> words = Arrays.asList("Oracle", "Java", "Magazine");
List<Integer> wordLengths =
words.stream()
.map(String::length)
.collect(toList());
```

Observar que en el ejemplo anterior se ha pasado de una lista de `String` a una lista de `Integer` mediante `map()`

● **Práctica 36:** Utiliza `findFirst()` y `Optional.ifPresent()` para mostrar a un Hombre llamado: "Luis" que debes primero agregar al `arraylist` personas.

Reducción. Las operaciones terminales que vimos hasta ahora devuelven objetos boolean (`allMatch` y similares), `void` (`forEach`) u `Optional` (`findAny` y similares). También hemos usado `collect` para combinar todo el conjunto de elementos de un `Stream` en un objeto `List`.

También podemos combinar todos los elementos de un stream para formular consultas de procesos más complicadas, como "¿cuál es la transacción con la identificación más alta?" o "calcular la suma de los valores de todas las transacciones". Para ello, se puede usar la operación `reduce` con streams; esta operación aplica reiteradamente una operación (por ejemplo, la suma de dos números) a cada elemento hasta que se genera un resultado

Por ej. en la forma clásica podemos calcular la suma de una lista mediante:

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

El equivalente con Streams:


```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```


`reduce()` recibe dos parámetros: el valor inicial que es cero, y el segundo parámetro es la función a aplicar

La función que recibe es del tipo `BinaryOperator<T>` que simplemente trabaja con dos parámetros de tipo `T` y devuelve un valor de tipo `T`

Otro Ej. Calcular el máximo:

```
int max = numbers.stream().reduce(0, Integer::max);
```

 **Práctica 37:** Calcular el máximo utilizando `reduce()` pero sin usar `Integer::max` , esto es: mediante una función lambda que generemos nosotros

Acabamos de ver cómo usar el método `reduce` para calcular la suma de un stream de números enteros. No obstante, ese enfoque tiene una desventaja: se llevan a cabo muchas operaciones de *boxing* para sumar repetidamente objetos `Integer`

Java incorpora 3 interfaces que transforman streams primitivos en especializados para abordar ese problema: *IntStream*, *DoubleStream* y *LongStream*; cada una de ellas convierte los elementos de un stream de manera especializada para que sean de tipo *int*, *double* o *long*, respectivamente.

Los métodos más habituales para convertir un stream en una versión especializada son `mapToInt`, `mapToDouble` y `mapToLong`. Estos métodos funcionan exactamente igual que el método `map` que vimos anteriormente, pero devuelven un stream especializado

Para ilustrar esto observar: `Integer.sum()` mediante este método podríamos hacer la suma que antes hicimos con `reduce()`

Si intentáramos hacer esto fallaría:

```
int suma = numericos.stream().sum();
```

El motivo del fallo es que estamos tratando de aplicar un método que existe en Integer llamado sum() a datos primitivos int que no tienen tal método

La solución sería:

```
int suma = numericos.stream().mapToInt(Integer::new).sum()
```

Utilizamos mapToInt() para convertir nuestros primitivos int en Integer y ya podemos

Nota: Integer::new es una referencia al constructor de Integer. Si quisiéramos tener una referencia al constructor de Coche haríamos: Coche::new

● **Práctica 38:** Probar los dos ejemplos de sum() el que debe fallar y el que debe funcionar mediante mapToInt() Tomar captura de pantalla del mensaje de error y de la ejecución bien realizada

● **Práctica 39:** A partir de la práctica 30 con la clase Matriz2x2 hacer un ArrayList con varias matrices y obtener utilizando Stream desde el ArrayList:

- el máximo determinante (no la matriz obtener el número)
- una matriz con determinante cero (si la hay)

Nota: Netbeans tiene una forma cómoda de crear getter() y setter():
Refactor → Encapsule fields
(o con atajo de teclado: CTRL+ALT+SHIFT+E)

 **Práctica 43:**