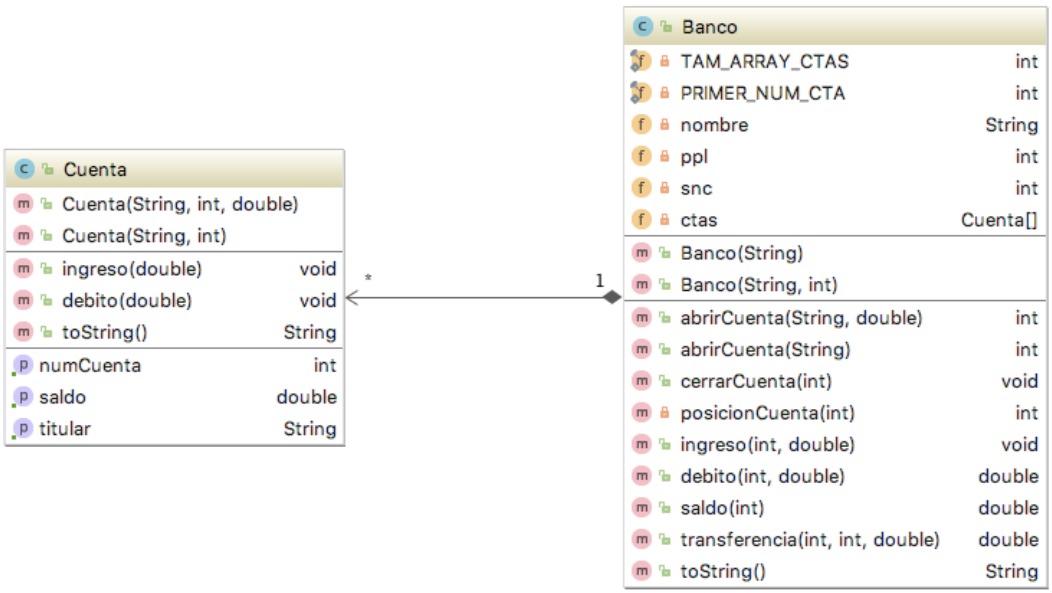
MÓDULO INICIAL. PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

Relación de Problemas Nº 2

Módulo mdBancoV1 (arrays, excepciones)

El objetivo de esta práctica es el de crear clases para representar bancos y cuentas bancarias (en el paquete `banco`).

Defínase una clase `Cuenta` de cuentas bancarias. Toda cuenta bancaria vendrá dada por un número de cuenta (de tipo `int`), un titular de la cuenta bancaria (de tipo `String`) y un saldo (de tipo `double`). Sobre una cuenta bancaria se podrán realizar operaciones de ingreso (añadir dinero a la cuenta) y de débito (extraer una cantidad de dinero de la misma, no importando que el saldo quede negativo), además de operaciones de consulta del número de la cuenta, su titular y saldo.



Defínase una clase `Banco` para representar bancos. Un banco tendrá un nombre (dado en el constructor) y dispondrá de una colección de cuentas bancarias (representadas en un `array` de cuentas). Además, la clase `Banco` contendrá las variables de instancia `ppl` y `snc` que representan respectivamente la primera posición libre en el array de cuentas (están en posiciones contiguas) y el siguiente número de cuenta libre, de forma que una vez asignado el número a una nueva cuenta este contador se debe incrementar. El valor inicial de `snc` será el de la constante `PRIMER_NUM_CTA` (`1001`)

La clase Banco debe proporcionar métodos para llevar a cabo operaciones de ingreso, de débito o de consulta de saldo sobre cualquiera de sus cuentas dado su número de cuenta, así como una operación de transferencia entre dos cuentas bancarias indicando los números de cuenta de la cuenta origen y de la cuenta destino y la cantidad a transferir. Los métodos `public int abrirCuenta(String, double)` y `public cerrarCuenta(int)` se encargarán de crear una nueva cuenta bancaria y de cerrar una existente. Para

abrir una cuenta bancaria bastará con proporcionar el nombre del titular de la cuenta (suponemos un único titular por cuenta) y un saldo inicial. Si no se indica saldo inicial la cuenta se creará con saldo cero. El banco se encargará de asignar un número de cuenta a la nueva cuenta, que será devuelto como resultado de la operación. A la hora de cerrar una cuenta existente, después de su supresión las cuentas restantes habrán de ocupar posiciones contiguas en el array de cuentas.

Algunas cuestiones:

- Los constructores de la clase Banco se encargarán de crear el array de cuentas con un tamaño inicial, así como de dar valores iniciales a las variables auxiliares. El segundo argumento del constructor de Banco con dos argumentos indica el tamaño inicial del array de cuentas. En el constructor con un argumento el array se inicializará con un tamaño por defecto (10 definido en la constante `TAM_ARRAY_CTAS`).
- La creación de cuentas no debe fallar porque el array esté lleno; en caso de no quedar espacio en el array se creará un array (de doble capacidad) para habilitar más espacio.
- Un banco no permite que una cuenta quede con saldo negativo. Si se intenta realizar un débito por una cantidad mayor que el saldo, solo se permitirá extraer el saldo. En ese caso, el saldo quedará a 0 y se devolverá el dinero que realmente se ha extraído de la cuenta.
- Para facilitar la implementación de algunos métodos se debe implementar un método llamado `private int posicionCuenta(int)`. Éste es un método auxiliar que devuelve la posición dentro del array de cuentas en la que se encuentra la cuenta con el número de cuenta dado como argumento. Si la cuenta no existe, el método debe lanzar una excepción `RuntimeException` de la siguiente manera:

  ```
  throw new RuntimeException("No existe la cuenta dada");
  ```

- Se debe proporcionar una redefinición del método `toString` para las clases `Cuenta` y `Banco`.

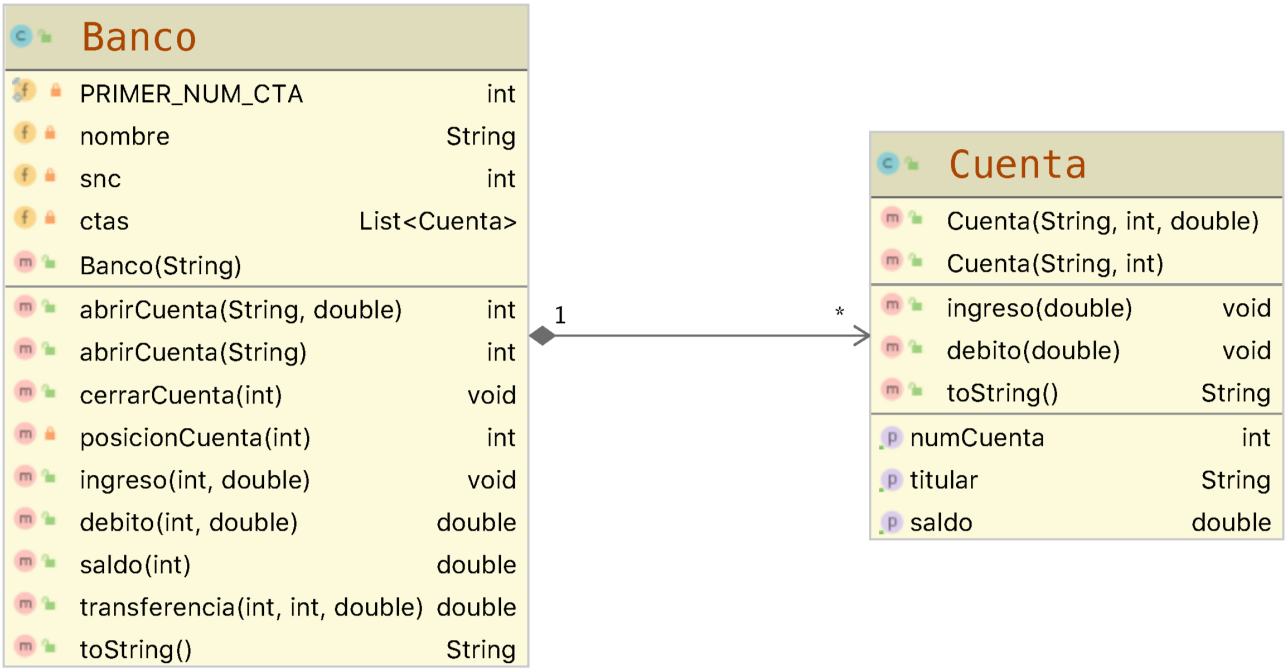
Al ejecutar la clase de prueba `TestBancoA`, el resultado ha de ser el mostrado más abajo.

```
// salida:
// TubbiesBank: [[(Po/1001) -> 500.0] [(Dixy/1002) -> 500.0] [(Tinky Winky/1003) -> 500.0] [(Lala/1004) -> 500.0] [(Nono/1005) -> 0.0] ]
// TubbiesBank: [[(Po/1001) -> 600.0] [(Dixy/1002) -> 400.0] [(Tinky Winky/1003) -> 400.0] [(Lala/1004) -> 600.0] [(Nono/1005) -> 300.0] ]
// TubbiesBank: [[(Dixy/1002) -> 600.0] [(Tinky Winky/1003) -> 400.0] [(Lala/1004) -> 600.0] ]
```

Módulo mdBancoV1L (listas, excepciones)

El objetivo de esta práctica es el de crear clases para representar bancos y cuentas bancarias (en el paquete `banco`).

Defínase una clase `Cuenta` de cuentas bancarias. Toda cuenta bancaria vendrá dada por un número de cuenta (de tipo `int`), un titular de la cuenta bancaria (de tipo `String`) y un saldo (de tipo `double`). Sobre una cuenta bancaria se podrán realizar operaciones de ingreso (añadir dinero a la cuenta) y de débito (extraer una cantidad de dinero de esta, no importando que el saldo quede negativo), además de operaciones de consulta del número de la cuenta, su titular y saldo.



Defínase una clase `Banco` para representar bancos. Un banco tendrá un nombre (dado en el constructor) y dispondrá de una colección de cuentas bancarias (representadas en una `lista` de cuentas). Además, la clase `Banco` contendrá la variable de instancia `snc` que representa el siguiente número de cuenta libre, de forma que una vez asignado el número a una nueva cuenta este contador se debe incrementar. El valor inicial de `snc` será el de la constante `PRIMER_NUM_CTA` (`1001`)

La clase Banco debe proporcionar métodos para llevar a cabo operaciones de ingreso, de débito o de consulta de saldo sobre cualquiera de sus cuentas dado su número de cuenta, así como una operación de transferencia entre dos cuentas bancarias indicando los números de cuenta de la cuenta origen y de la cuenta destino y la cantidad a transferir. Los métodos `public int abrirCuenta(String, double)` y `public cerrarCuenta(int)` se encargarán de crear una nueva cuenta bancaria y de cerrar una existente. Para abrir una cuenta bancaria bastará con proporcionar el nombre del titular de la cuenta (suponemos un único titular por cuenta) y un saldo inicial. Si no se indica saldo inicial la cuenta se creará con saldo cero. El banco se encargará de asignar un número de cuenta a la nueva cuenta, que será devuelto como resultado de la operación.

Algunas cuestiones:

- El constructor de la clase Banco se encargará de crear la lista de cuentas así como de dar valores iniciales al resto de variables

- Un banco no permite que una cuenta quede con saldo negativo. Si se intenta realizar un débito por una cantidad mayor que el saldo, solo se permitirá extraer el saldo. En ese caso, el saldo quedará a 0 y se devolverá el dinero que realmente se ha extraído de la cuenta.
- Para facilitar la implementación de algunos métodos se debe implementar un método llamado `private int posicionCuenta(int)`. Éste es un método auxiliar que devuelve la posición dentro de la lista de cuentas en la que se encuentra la cuenta con el número de cuenta dado como argumento. Si la cuenta no existe, el método debe lanzar una excepción `RuntimeException` de la siguiente manera:

  ```
  throw new RuntimeException("No existe la cuenta dada");
  ```

- Se debe proporcionar una redefinición del método `toString` para las clases `Cuenta` y `Banco`.

Al ejecutar la clase de prueba `TestBancoA`, el resultado ha de ser el mostrado más abajo.

```
// salida:
// TubbiesBank: [[(Po/1001) -> 500.0] [(Dixy/1002) -> 500.0] [(Tinky Winky/1003) -> 500.0] [(Lala/1004) -> 500.0] [(Nono/1005) -> 0.0] ]
// TubbiesBank: [[(Po/1001) -> 600.0] [(Dixy/1002) -> 400.0] [(Tinky Winky/1003) -> 400.0] [(Lala/1004) -> 600.0] [(Nono/1005) -> 300.0] ]
// TubbiesBank: [[(Dixy/1002) -> 600.0] [(Tinky Winky/1003) -> 400.0] [(Lala/1004) -> 600.0] ]
```

Módulo mdTren (listas)(*advanced*)

Un tren de vapor necesita incorporar algunos vagones que transporten el carbón que necesita para su combustión. Vamos a definir dos clases, `Tren` y `Vagon`.

La clase `Vagon` recibe como parámetro del constructor un entero que representa la capacidad de carbón en toneladas que puede almacenar. Un vagón tiene almacenada además de la capacidad, una cantidad de carbón en una variable privada llamada `carga` (`int`) que indica la cantidad de carbón que transporta. Al construir el vagón, la `carga` es 0. Define esta clase y:

- El método `int carga(int ton)` que intenta cargar en el vagón las toneladas que se le pasan como argumento. Si `ton` es mayor que la capacidad actual del vagón, se carga hasta su capacidad. El método devuelve las toneladas que no se han podido cargar o 0 si se han podido cargar todas.
- El método `int descarga(int ton)` que descarga del vagón las toneladas que se pasan como argumento. Devuelve las toneladas que no se han podido descargar porque el vagón no tiene suficiente carga o 0 si se han podido descargar todas.
- El método `int getCarga()` que devuelve la carga que actualmente tiene el vagón y el método `int getCapacidad()` que indica cuanto carbón cabe aún en el vagón.
- Sobreescribe el método `String toString()` para que un vagón con capacidad de almacenaje de 20 toneladas y con carga actual de 5 se convierta en la cadena "`V(5/20)`" .

La clase `Tren` recibe como parámetros del constructor el número de vagones que llevará y la capacidad común que tendrá cada vagón. El constructor deberá crear una lista con el número de vagones indicado, cada uno de la capacidad especificada. Esta lista se almacenará en un variable llamada `vagones`. Define esta clase y:

- El método `void carga(int ton)` que carga en los vagones las toneladas que recibe como argumento. Para ello, va añadiendo carga a cada uno de los vagones, hasta llegar a cargar en el tren un total de `ton` toneladas. Si un vagón se llena, se pasa al siguiente para cargar el resto de toneladas y, si no hay mas vagones disponibles, se creará un nuevo vagón, con la capacidad especificada en el constructor de la clase `Tren`, y se añadirá al final de la lista para continuar con el proceso de carga. De esta forma, puede que sea necesario añadir más de un vagón al tren para completar la carga.
- El método `void gasta(int ton)` que simula el gasto de toneladas que consume el tren en un trayecto. Para ello, se irá descargando de cada vagón las toneladas que sean necesarias, desde el primero en adelante, hasta descargar un total de `ton` toneladas. Si se llega al final del tren y no se ha podido descargar todas las toneladas, se debe lanzar una `IllegalArgumentException` indicando el motivo.
- El método `void optimiza()` que elimina de la lista de vagones aquellos que están vacíos.
- Sobreescribe el método `String toString()` para que un tren con tres vagones de capacidad total 20, el primero de ellos con una carga de 10 toneladas, el segundo con 3 y el tercero vacío se represente con la cadena "`Tren[V(10/20), V(3/20), V(0/20)]`".

Comprueba el funcionamiento de tus clases con la siguiente aplicación:

```
class Main {
  public static void main(String [] args) {
        Tren tren = new Tren(5,20)
        System.out.println("Se crea tren con 5 vagones de 20 ton." + tren)
        tren.carga(71)
        System.out.println("Se cargan 71 ton: " + tren)
        tren.gasta(63)
        System.out.println("Se gastan 63 ton: "+ tren)
         tren.carga(38)
        System.out.println("Se cargan 38 ton: " + tren)
        tren.gasta(21)
        System.out.println("Se gastan 21 ton: " + tren)
        tren.optimiza()
        System.out.println("Se optimiza el número de vagones: " + tren)
        tren.carga(21)
        Sytems.out.println("Se cargan 21 ton: "+ tren)
  }
```

}

La ejecución debería producir el siguiente resultado:

```
Se crea tren con 5 vagones de 20 ton: Tren[V(0/20), V(0/20), V(0/20), V(0/20), V(0/20)]
Se cargan 71 ton: Tren[V(20/20), V(20/20), V(20/20), V(11/20), V(0/20)]
Se gastan 63 ton: Tren[V(0/20), V(0/20), V(0/20), V(8/20), V(0/20)]
Se cargan 38 ton: Tren[V(20/20), V(18/20), V(0/20), V(8/20), V(0/20)]
Se gastan 21 ton: Tren[V(0/20), V(17/20), V(0/20), V(8/20), V(0/20)]
Se optimiza el número de vagones: Tren[V(17/20), V(8/20)]
Se cargan 21 ton: Tren[V(20/20), V(20/20), V(6/20)]
```

Módulo mdUrna (enum, random, excepciones)(*mandatory*)

El objetivo de este ejercicio es crear una clase `Urna` cuyos objetos pueden contener bolas blancas y bolas negras, y que nos permita realizar unas operaciones básicas sobre la misma.

- La clase tendrá un par de variables de instancia, `negras` y `blancas`, en las que se almacenará el número de bolas de cada color.
- La clase `Urna` dispondrá de un constructor que permita crear instancias de la clase con el número inicial de bolas blancas y negras pasados como parámetros. El constructor deberá garantizar que no se crean urnas con un número negativo de bolas. En caso de que alguna sea negativa deberá lanzar la excepción `IllegalArgumentException`.
- Además, incluirá métodos para:
  - Consultar el número total de bolas que tiene (`public int totalBolas()`).
  - Extraer una bola aleatoriamente y saber su color (`public ColorBola extraerBola()`). El color vendrá dado por un enumerado `ColorBola` con dos elementos: `Blanca` y `Negra`. El tipo enumerado `ColorBola` debe definirse como una clase interna (anidada) estática a la clase `Urna`. Para extraer una bola aleatoriamente se ha de sumar el número de bolas blancas y negras y tomar un número aleatorio entre 1 y dicha suma. Si ese número es menor o igual que el número de bolas blancas supondremos que la bola que sale es blanca; en otro caso, que es negra. Utilizad la clase `java.util.Random` para la generación de números aleatorios. Consultad la documentación de la API para disponer de información sobre el uso de dicha clase. Si no hay bolas deberá lanzarse una excepción `NoSuchElementException`.
  - Introducir una bola de un color determinado (`public void ponerBlanca()`, `public void ponerNegra()`).

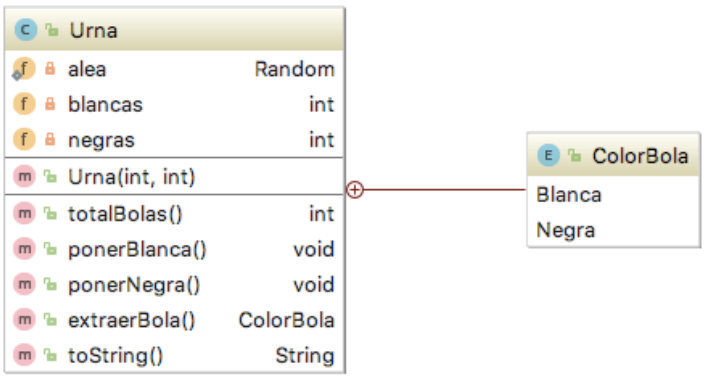
Cread una aplicación que cree una urna, tomando el número de bolas blancas y negras como argumentos al ejecutar la aplicación, y realice con ella el siguiente experimento:

- Mientras quede más de una bola en la urna, sacar dos bolas.
- Si ambas son del mismo color introducir una bola blanca en la urna; si son de distinto color introducir una bola negra (se supone que disponemos de suficientes bolas de ambos colores fuera de la urna).
- Por último, cuando quede sólo una bola, sacarla y mostrar su color.

Utilícese el método `parseInt` de la clase `Integer` en `java.lang` para convertir un `String` en un valor de tipo `int`.

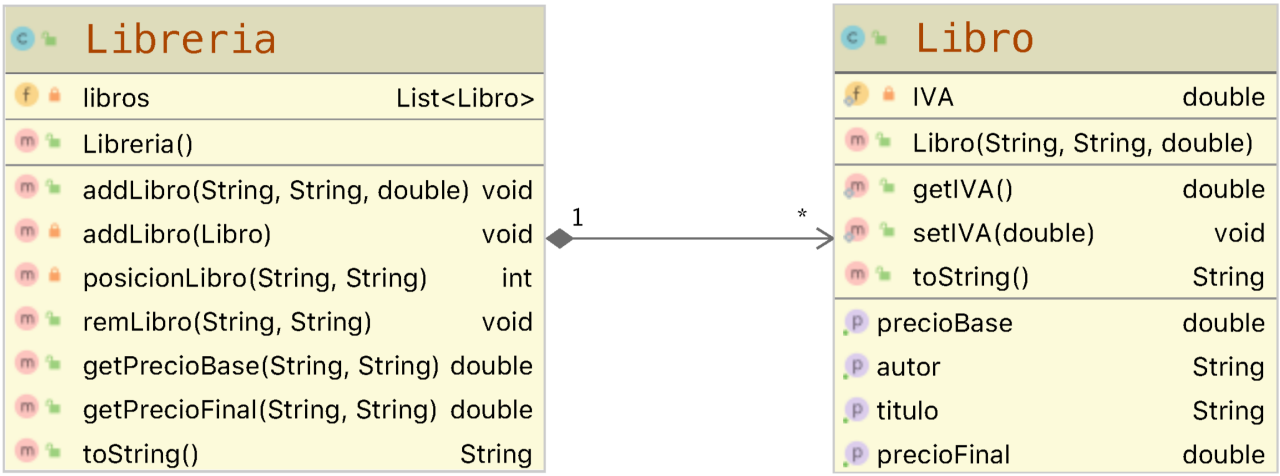
Infórmese al usuario por la salida estándar de cualquier error que se pueda producir.

Analizad los resultados obtenidos sobre el color de la bola final dependiendo del número de bolas iniciales y de su color.



Módulo mdLibreriaV1L (composición, static, listas, paquetes)

Se pretende crear clases que mantengan información sobre libros. Para ello, se crearán las clases `Libro` (del paquete `libreria`) y la clase `Libreria` (en el mismo paquete).



Nota: se pueden añadir a las siguientes clases los métodos **privados** que se consideren necesarios.

La clase `Libro`

La clase `Libro` (del paquete `libreria`) contiene información sobre un determinado libro, tal como el nombre del autor, el título, y el precio base. Además, también posee información sobre el porcentaje de IVA que se aplica para calcular su precio final. Nótese que el porcentaje de IVA a aplicar es el mismo y es compartido por todos los libros, siendo su valor inicial el 10 %.

▸ `Libro(String,String,double)`

Construye un objeto `Libro`. Recibe como parámetros, en el siguiente orden, el nombre del autor, el título, y el precio base del libro.

▸ `getAutor():String`

▸ `getTitulo():String`

▸ `getPrecioBase():double`

Devuelven los valores correspondientes almacenados en el objeto.

▸ `getPrecioFinal():double`

Devuelve el precio final del libro, incluyendo el IVA, según la siguiente ecuación.

$$PF = PB + PB \cdot IVA / 100$$

▸ `toString(): String` *// [@Redefinición]*

Devuelve la representación textual del objeto, según el formato del siguiente ejemplo:

```
(Isaac Asimov; La Fundación; 7.30; 10%; 8.03)
```

▸ `getIVA():double` *// [@MétodoDeClase]*

Devuelve el porcentaje del IVA asociado a la clase `Libro`.

▸ `setIVA(double):void` *// [@MétodoDeClase]*

Actualiza el valor del porcentaje del IVA asociado a la clase `Libro` al valor recibido como parámetro.

La clase `Libreria`

La clase `Libreria` (del paquete `libreria`) almacena múltiples instancias de la clase `Libro` en una lista.

Nota 1: las comparaciones que se realicen tanto del nombre del autor como del título del libro se deberán realizar sin considerar el caso de las letras que lo componen.

Nota 2: se recomienda la definición de métodos privados que simplifiquen y permitan modularizar la solución de métodos complejos.

▸ `Libreria()`

Construye un objeto `Librería` vacío (sin libros) y crea la lista de libros.

▸ `addLibro(String,String,double): void`

Crea un nuevo objeto `Libro` con el nombre del autor, el título, y el precio base recibidos como parámetros. Si ya existe un libro de ese mismo autor, con el mismo título, entonces se reemplaza el libro anterior por el nuevo. En otro caso, añade el nuevo libro a la librería.

▸ `remLibro(String,String): void`

Si existe el libro correspondiente al autor y título recibidos como parámetros, entonces elimina el libro de la librería.

▸ `getPrecioBase(String,String): double`

Devuelve el precio base del libro correspondiente al autor y título recibidos como parámetros. Si el libro no existe en la librería, entonces devuelve cero.

▸ `getPrecioFinal(String,String): double`

Devuelve el precio final del libro correspondiente al autor y título especificados. Si el libro no existe en la librería, entonces devuelve cero.

▸ `toString(): String // [@Redefinición]`

Devuelve la representación textual del objeto, según el formato del siguiente ejemplo: (sin considerar los saltos de línea)

```
[(George Orwell; 1984; 6.20; 10%; 6.82),
 (Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
 (Isaac Asimov; Fundación e Imperio; 9.40; 10%; 10.34),
 (Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
 (Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
 (Isaac Asimov; La Fundación; 7.30; 10%; 8.03),
 (William Gibson; Neuromante; 8.30; 10%; 9.13),
 (Isaac Asimov; Segunda Fundación; 8.10; 10%; 9.81),
 (Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

Desarrolle una aplicación `PruebaLibreria` (en el paquete anónimo) que permita realizar una prueba de las clases anteriores. Así, deberá añadir a la librería los siguientes libros:

```
("george orwell", "1984", 8.20)
("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?", 3.50)
("Isaac Asimov", "Fundación e Imperio", 9.40)
("Ray Bradbury","Fahrenheit451",7.40)
("Alex Huxley", "Un Mundo Feliz",6.50)
("Isaac Asimov", "La Fundación",7.30),
("William Gibson", "Neuromante", 8.30)
("Isaac Asimov","SegundaFundación",8.10)
("Isaac Newton", "arithmetica universalis", 7.50)
("George Orwell", "1984", 6.20)
("Isaac Newton", "Arithmetica Universalis", 10.50)
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
[(George Orwell; 1984; 6.20; 10%; 6.82),
(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
(Isaac Asimov; Fundación e Imperio; 9.40; 10%; 10.34),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
(Isaac Asimov; La Fundación; 7.30; 10%; 8.03),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 10%; 9.81),
(Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

A continuación, se eliminarán los siguientes libros:

```
("George Orwell", "1984")
("Alex Huxley", "Un Mundo Feliz")
("Isaac Newton", "Arithmetica Universalis")
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
[(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%;3.85),
(Isaac Asimov; Fundación e Imperio; 9.40; 10%; 10.34),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Isaac Asimov; La Fundación; 7.30; 10%; 8.03),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 10%; 9.81)]
```

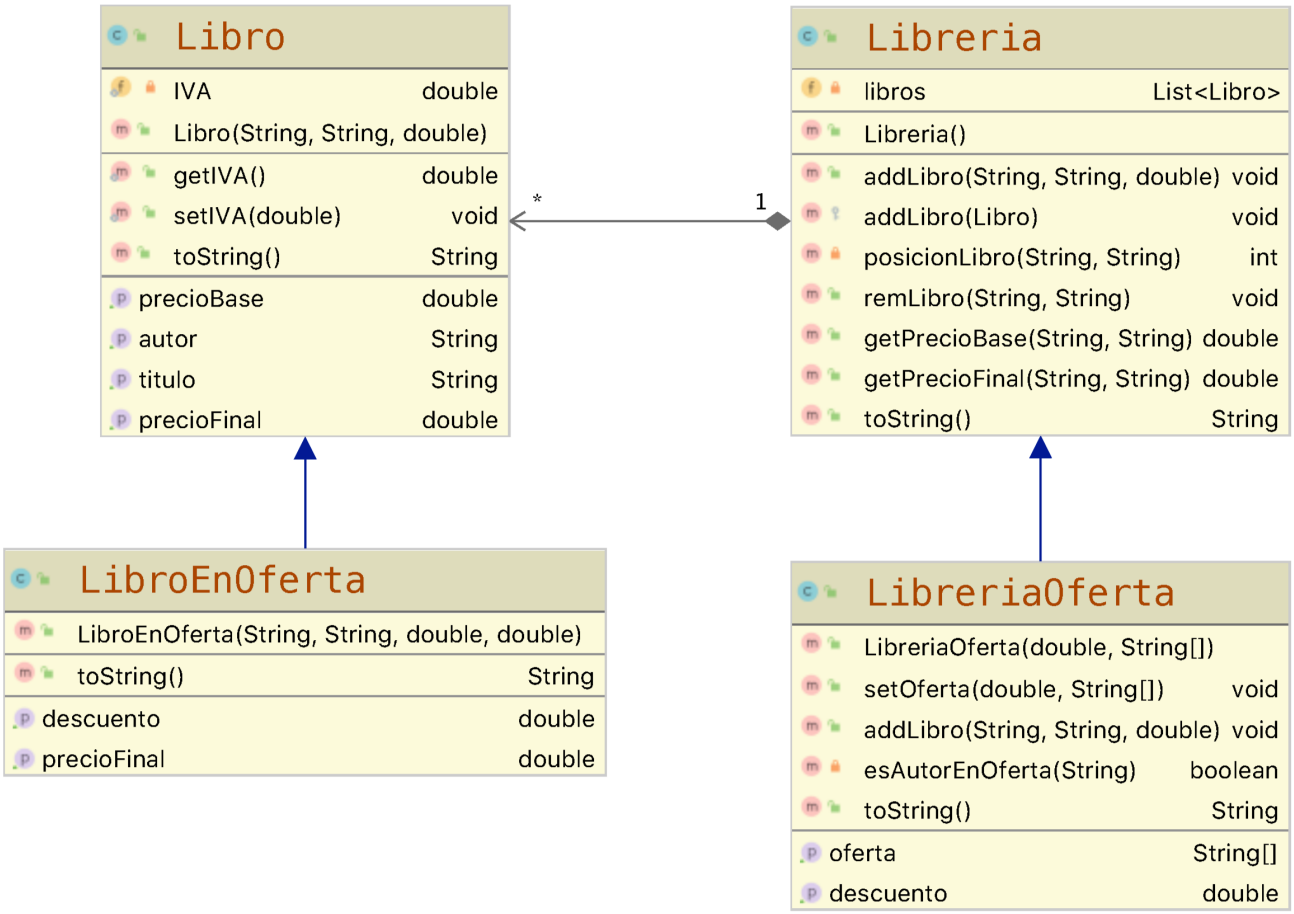
Finalmente se mostrará el precio final de los siguientes libros:

```
getPrecioFinal("George Orwell", "1984"): 0
getPrecioFinal("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?"): 3.85
getPrecioFinal("isaac asimov", "fundación e imperio"): 10.34
getPrecioFinal("Ray Bradbury", "Fahrenheit 451"): 8.14
getPrecioFinal("Alex Huxley", "Un Mundo Feliz"): 0
getPrecioFinal("Isaac Asimov", "La Fundación"): 8.03
getPrecioFinal("william gibson", "neuromante"): 9.13
getPrecioFinal("Isaac Asimov", "Segunda Fundación"): 9.81
getPrecioFinal("Isaac Newton", "Arithmetica Universalis"): 0
```

Módulo mdLibreriaV2L (mdLibreriaV1L, herencia)

Este ejercicio pretende redefinir el comportamiento de las clases `Libro` y `Libreria` del módulo `mdLibreriaV1L` para que permita tener *autores en oferta* en la librería, de tal forma que sus libros tengan precios en oferta. (*hacer una copia de los ficheros del módulo mdLibreriaV1L* para este módulo)

El diagrama de clases UML



Nota: se pueden añadir o cambiar a **`protected`** algunos de los *métodos privados* de las clases de la práctica 2.2.

Nota: se pueden añadir a las siguientes clases los métodos **privados** que se consideren necesarios.

La clase `LibroEnOferta`

La clase `LibroEnOferta` (del paquete `libreria`) deriva de la clase `Libro`, por lo que contiene información sobre un determinado libro, pero además, permite especificar un determinado porcentaje de descuento, que será aplicado al **precio base** al calcular el precio final del libro.

▸ `LibroEnOferta(String,String,double,double)`

Construye un objeto `LibroEnOferta`. Recibe como parámetros, en el siguiente orden, el nombre del autor, el título, el precio base y el porcentaje de descuento del libro.

- `getDescuento():double`

  Devuelve el porcentaje de descuento del libro.

- `getPrecioFinal():double // [@Redefinición]`

  Devuelve el precio final del libro, aplicando el descuento al precio base, e incluyendo el IVA, según las siguientes ecuaciones.

  $$PF = PB + PB \cdot IVA \ /100 \text{ (como en libro)}$$

  $$PFD = PF - PF \cdot Descuento \ / 100$$

- `toString(): String // [@Redefinición]`

  Devuelve la representación textual del objeto, según el formato del siguiente ejemplo:

  ```
  (Isaac Asimov; La Fundación; 7.30; 20%; 5.84; 10%; 6.424)
  ```

La clase `LibreriaOferta`

La clase `LibreriaOferta` (del paquete `libreria`) deriva de la clase `Libreria`, pero permite crear y almacenar libros en oferta. Para ello, contiene además un array con los nombres de los autores en oferta, así como del porcentaje de oferta a aplicar a los libros de estos autores.

Nota 1: las comparaciones que se realicen tanto del nombre del autor como del título del libro se deberán realizar sin considerar el caso de las letras que lo componen.

Nota 2: se recomienda la definición de métodos privados y/o protegidos que simplifiquen y permitan modularizar la solución de métodos complejos.

- `LibreriaOferta(double,String[])`

  Construye un objeto `LibreriaOferta` vacío (sin libros. Además, el porcentaje de descuento se recibe como primer parámetro, y el array con los nombres de los autores en oferta se recibe como segundo parámetro.

- `setOferta(double,String[]):void`

  Actualiza el valor del porcentaje de descuento y el array de autores en oferta con los valores recibidos como parámetros.

- `getOferta():String[]`

  Devuelve el array de autores en oferta.

- `getDescuento():double`

  Devuelve el porcentaje de descuento.

- `addLibro(String,String,double):void // [@Redefinición]`

  Si el nombre del autor recibido como primer parámetro es igual a algún autor del array de autores en oferta, entonces crea un nuevo objeto `LibroEnOferta` con el nombre del autor, el título, y el precio base recibidos como parámetros, y el

descuento según el valor almacenado. En otro caso, crea un nuevo objeto `Libro` con el nombre del autor, el título, y el precio base recibidos como parámetros.

Si ya existe un libro de ese mismo autor, con el mismo título, entonces se reemplaza el libro anterior por el nuevo. En otro caso, añade el nuevo libro a la librería.

▸ `toString(): String // [@Redefinición]`

Devuelve la representación textual del objeto, según el formato del siguiente ejemplo: (sin considerar los saltos de línea)

```
20%[George Orwell; Isaac Asimov]
[(George Orwell; 1984; 6.20; 20%; 4.96; 10%; 5.456),
 (Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
 (Isaac Asimov; Fundación e Imperio; 9.40; 20%; 7.52; 10%; 8.272),
 (Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
 (Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
 (Isaac Asimov; La Fundación; 7.30; 20%; 5.84; 10%; 6.424),
 (William Gibson; Neuromante; 8.30; 10%; 9.13),
 (Isaac Asimov; Segunda Fundación; 8.10; 20%; 6.48; 10%; 7.128),
 (Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

La aplicación `PruebaLibreria`

Desarrolle una aplicación (en el paquete anónimo) que permita realizar una prueba de las clases anteriores. Así, deberá crear un objeto `LibreriaOferta` con un 20 % de descuento para los autores *George Orwell* e *Isaac Asimov*. Además, debe añadir a la librería los siguientes libros:

```
("george orwell", "1984", 8.20)
("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?", 3.50)
("Isaac Asimov", "Fundación e Imperio", 9.40)
("Ray Bradbury", "Fahrenheit 451", 7.40)
("Alex Huxley", "Un Mundo Feliz", 6.50)
("Isaac Asimov", "La Fundación", 7.30)
("William Gibson", "Neuromante", 8.30)
("Isaac Asimov", "Segunda Fundación", 8.10)
("Isaac Newton", "arithmetica universalis", 7.50)
("George Orwell", "1984", 6.20)
("Isaac Newton", "Arithmetica Universalis", 10.50)
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
20%[George Orwell; Isaac Asimov]
[(George Orwell; 1984; 6.20; 20%; 4.96; 10%; 5.456),
(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
(Isaac Asimov; Fundación e Imperio; 9.40; 20%; 7.52; 10%; 8.272),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
(Isaac Asimov; La Fundación; 7.30; 20%; 5.84; 10%; 6.424),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 20%; 6.48; 10%; 7.128),
(Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

A continuación, se eliminarán los siguientes libros:

```
("George Orwell", "1984")
("Alex Huxley", "Un Mundo Feliz")
("Isaac Newton", "Arithmetica Universalis")
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
20%[George Orwell; Isaac Asimov]
[(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
 (Isaac Asimov; Fundación e Imperio; 9.40; 20%; 7.52; 10%; 8.272),
 (Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
 (Isaac Asimov; La Fundación; 7.30; 20%; 5.84; 10%; 6.424),
 (William Gibson; Neuromante; 8.30; 10%; 9.13),
 (Isaac Asimov; Segunda Fundación; 8.10; 20%; 6.48; 10%; 7.128)]
```

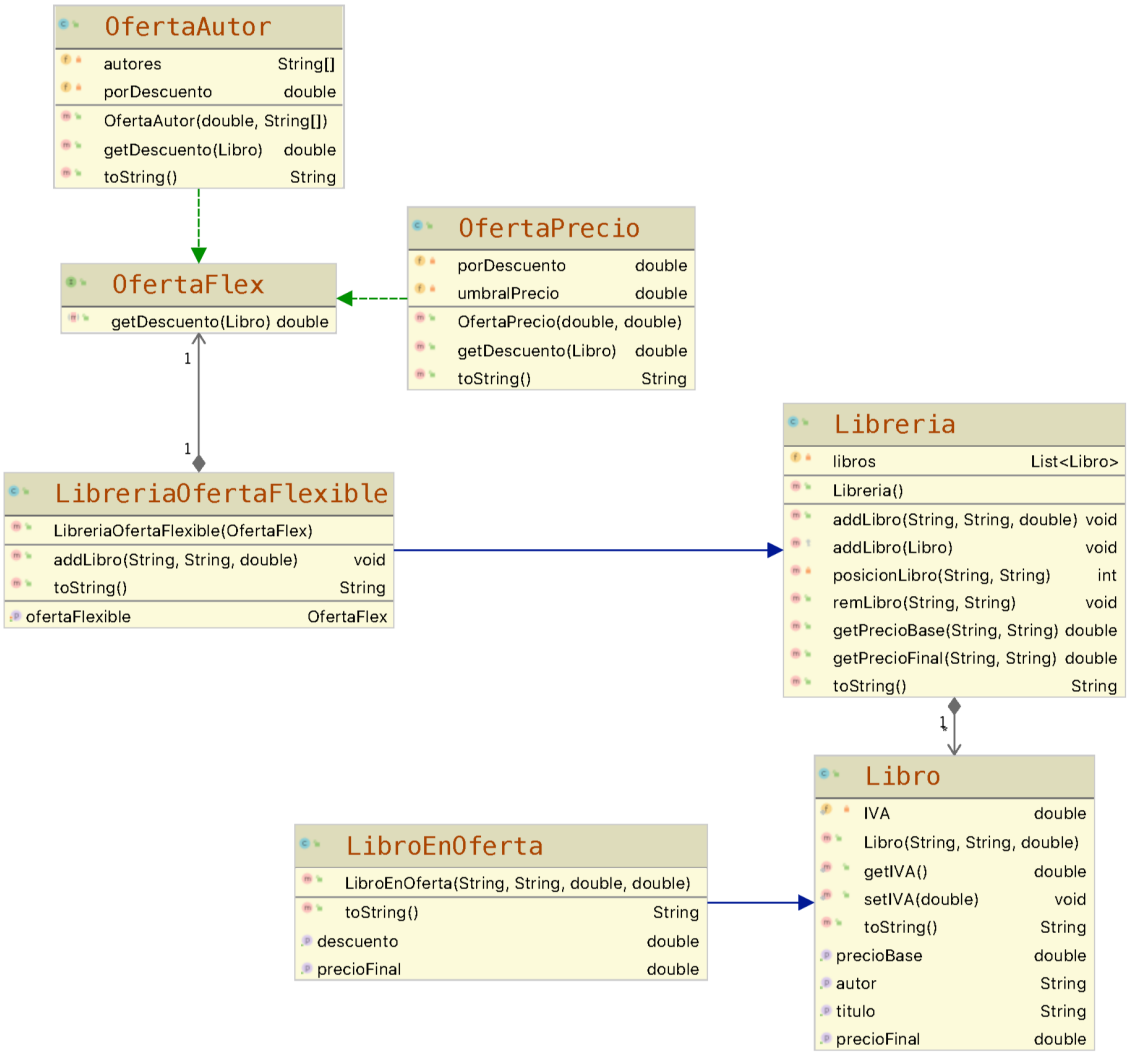
Finalmente se mostrará el precio final de los siguientes libros:

```
getPrecioFinal("George Orwell", "1984"): 0
getPrecioFinal("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?"): 3.85
getPrecioFinal("isaac asimov", "fundación e imperio"): 8.272
getPrecioFinal("Ray Bradbury", "Fahrenheit 451"): 8.14
getPrecioFinal("Alex Huxley", "Un Mundo Feliz"): 0
getPrecioFinal("Isaac Asimov", "La Fundación"): 6.424
getPrecioFinal("william gibson", "neuromante"): 9.13
getPrecioFinal("Isaac Asimov", "Segunda Fundación"): 7.128
getPrecioFinal("Isaac Newton", "Arithmetica Universalis"): 0
```

Módulo mdLibreriaV3L (mdLibreriaV2L, interfaces)

Este ejercicio pretende redefinir el comportamiento de las clases `Libro` y `Libreria` del módulo `mdLibreriaV2L` para que permita especificar diferentes criterios de ofertas en la librería, de tal forma que sus libros tengan precios en oferta. (*Hacer una copia de los ficheros del módulo mdLibreriaV2L en este módulo excepto de la clase LibreriaEnOferta*)

El diagrama de clases UML



Nota: se pueden añadir o cambiar a **`protected`** algunos de los *métodos privados* de las clases de la práctica 2.2.

Nota: se pueden añadir a las siguientes clases los métodos **privados** que se consideren necesarios.

La *Interfaz* `OfertaFlex`

La *interfaz* `OfertaFlex` (del paquete `libreria`) especifica los métodos necesarios para calcular el porcentaje de descuento que se debe aplicar a un determinado libro.

- `getDescuento(Libro): double`

Calcula y devuelve el porcentaje de descuento que se debe aplicar a un determinado libro recibido como parámetro. En caso de que no se deba aplicar ningún descuento, devolverá el valor cero.

La clase `OfertaPrecio`

La clase `OfertaPrecio` (del paquete `libreria`) implementa la *interfaz* `OfertaFlex` y proporciona un método para calcular el porcentaje de descuento a partir de un determinado umbral en el precio base del libro, ambos especificados en la construcción del objeto. (Si el precio base pasa del umbral, se aplica el descuento)

- `OfertaPrecio(double,double)`

  Construye un objeto con el porcentaje de descuento y el umbral del precio, recibidos como parámetros en ese mismo orden.

- `getDescuento(Libro): double // [@Redefinición]`

  Calcula y devuelve el porcentaje de descuento que se debe aplicar a un determinado libro recibido como parámetro si el precio base del libro es mayor o igual al umbral especificado en la construcción del objeto. En caso de que no se deba aplicar ningún descuento, devolverá el valor cero.

- `toString(): String // [@Redefinición]`

  Devuelve la representación textual del objeto, según el formato del siguiente ejemplo: (20% de descuento para los libros que vale 8 euros o mas)

  ```
  20%(8)
  ```

La clase `OfertaAutor`

La clase `OfertaAutor` (del paquete `libreria`) implementa la *interfaz* `OfertaFlex` y proporciona un método para calcular el porcentaje de descuento a partir de los nombres de autores en oferta, ambos especificados en la construcción del objeto.

Nota: las comparaciones que se realicen tanto del nombre del autor como del título del libro se deberán realizar sin considerar el caso de las letras que lo componen.

- `OfertaAutor(double,String[])`

  Construye un objeto con el porcentaje de descuento y el array con los nombres de los autores en oferta, recibidos como parámetros en ese mismo orden.

- `getDescuento(Libro): double // [@Redefinición]`

  Calcula y devuelve el porcentaje de descuento que se debe aplicar a un determinado libro recibido como parámetro si el nombre del autor se encuentra en el array de autores en oferta especificado en la construcción del objeto. En caso de que no se deba aplicar ningún

descuento, devolverá el valor cero.

- `toString(): String // [@Redefinición]`

  Devuelve la representación textual del objeto, según el formato del siguiente ejemplo:

  ```
  20%[George Orwell; Isaac Asimov]
  ```

La clase `LibreriaOfertaFlexible`

La clase `LibreriaOfertaFlexible` (del paquete `libreria`) deriva de la clase `Libreria`, pero permite crear y almacenar libros en oferta. Para ello, contiene una referencia a un objeto que implemente la intefaz `OfertaFlex`.

Nota 1: las comparaciones que se realicen tanto del nombre del autor como del título del libro se deberán realizar sin considerar el caso de las letras que lo componen.

Nota 2: se recomienda la definición de métodos privados y/o protegidos que simplifiquen y permitan modularizar la solución de métodos complejos.

- `LibreriaOfertaFlexible(OfertaFlex)`

  Construye un objeto `LibreriaOfertaFlexible` vacío (sin libros). Además, almacena la referencia al objeto para calcular las ofertas de los libros, que se recibe como primer parámetro.

- `setOferta(OfertaFlex):void`

  Actualiza el valor del objeto para calcular ofertas de libros al objeto recibido como parámetro.

- `getOferta():OfertaFlex`

  Devuelve el objeto para calcular ofertas de libros.

- `addLibro(String,String,double): void // [@Redefinición]`

  Crea un nuevo objeto `Libro` con el nombre del autor, el título, y el precio base recibidos como parámetros. Si el porcentaje de descuento calculado para este libro es distinto de cero, entonces vuelve a crear un nuevo objeto `LibroEnOferta` con el nombre del autor, el título, y el precio base recibidos como parámetros, y el descuento según el valor calculado por el objeto para el cálculo de las ofertas.

  Si ya existe un libro de ese mismo autor, con el mismo título, entonces se reemplaza el libro anterior por el nuevo. En otro caso, añade el nuevo libro a la librería.

- `toString(): String // [@Redefinición]`

  Devuelve la representación textual del objeto, según el formato del siguiente ejemplo: (sin considerar los saltos de línea)

  ```
  20%[George Orwell; Isaac Asimov] // u otra representación de OfertaFlex
  [(George Orwell; 1984; 6.20; 20%; 4.96; 10%; 5.456),
   (Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
  ```

```
(Isaac Asimov; Fundación e Imperio; 9.40; 20%; 7.52; 10%; 8.272),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
(Isaac Asimov; La Fundación; 7.30; 20%; 5.84; 10%; 6.424),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 20%; 6.48; 10%; 7.128),
(Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

La aplicación `PruebaLibreria`

Desarrolle una aplicación (en el paquete anónimo) que permita realizar una prueba de las clases anteriores. Así, deberá crear un objeto `LibreriaOfertaFlexible` con un objeto `OfertaAutor` que especifica un 20% de descuento para los autores *George Orwell* e *Isaac Asimov*. Además, debe añadir a la librería los siguientes libros:

```
("george orwell", "1984", 8.20)
("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?", 3.50)
("Isaac Asimov", "Fundación e Imperio", 9.40)
("Ray Bradbury", "Fahrenheit 451", 7.40)
("Alex Huxley", "Un Mundo Feliz", 6.50)
("Isaac Asimov", "La Fundación", 7.30),
("William Gibson", "Neuromante", 8.30)
("Isaac Asimov", "Segunda Fundación", 8.10)
("Isaac Newton", "arithmetica universalis", 7.50)
("George Orwell", "1984", 6.20)
("Isaac Newton", "Arithmetica Universalis", 10.50)
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
20%[George Orwell; Isaac Asimov]
[(George Orwell; 1984; 6.20; 20%; 4.96; 10%; 5.456),
(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
(Isaac Asimov; Fundación e Imperio; 9.40; 20%; 7.52; 10%; 8.272),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
(Isaac Asimov; La Fundación; 7.30; 20%; 5.84; 10%; 6.424),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 20%; 6.48; 10%; 7.128),
(Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

A continuación, se eliminarán los siguientes libros:

```
("George Orwell", "1984")
("Alex Huxley", "Un Mundo Feliz")
("Isaac Newton", "Arithmetica Universalis")
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
20%[George Orwell; Isaac Asimov]
[(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
(Isaac Asimov; Fundación e Imperio; 9.40; 20%; 7.52; 10%; 8.272),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Isaac Asimov; La Fundación; 7.30; 20%; 5.84; 10%; 6.424),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 20%; 6.48; 10%; 7.128)]
```

Finalmente se mostrará el precio final de los siguientes libros:

```
getPrecioFinal("George Orwell", "1984"): 0
getPrecioFinal("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?"): 3.85
getPrecioFinal("isaac asimov", "fundación e imperio"): 8.272
```

```
getPrecioFinal("Ray Bradbury", "Fahrenheit 451"): 8.14
getPrecioFinal("Alex Huxley", "Un Mundo Feliz"): 0
getPrecioFinal("Isaac Asimov", "La Fundación"): 6.424
getPrecioFinal("william gibson", "neuromante"): 9.13
getPrecioFinal("Isaac Asimov", "Segunda Fundación"): 7.128
getPrecioFinal("Isaac Newton", "Arithmetica Universalis"): 0
```

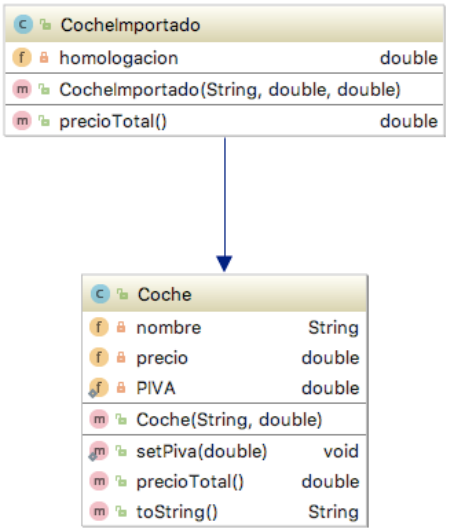
Módulo mdCoche (herencia)(*mandatory*)

Cread la clase `Coche` en el paquete `coches` cuyas instancias mantienen el nombre del coche y su precio (antes de impuestos). El precio total de un coche se calcula aplicando un IVA al precio marcado. Este IVA puede variar (por defecto el IVA es del 16%), pero será siempre el mismo para todos los coches. Proporcionad métodos para cambiar el IVA (`public static void setPiva(double)`, calcular el precio total (`public double precioTotal()`) y mostrar el coche como cadena de caracteres con el formato:

```
<nombre> -> <precio_total>
```

Un coche importado es un coche para el que además del IVA aplicamos un impuesto de homologación. El impuesto de homologación es específico de cada vehículo, y será dado en el momento de su creación. El precio total de un coche importado se calcula como el de cualquier coche pero ahora hay que sumar este nuevo impuesto. Cread la clase `CocheImportado` para mantener información de coches importados.

Probad las clases con la siguiente clase de prueba `EjCoches` proporcionada.



La salida del programa `EjCoches` debe ser:

```
Seat Panda -> 17400.0
Ferrari T-R -> 83400.0
Seat Toledo -> 24360.0
Jaguar XK -> 53560.0
Porche GT3 -> 58040.0
Con IVA de 18%
Seat Panda -> 17700.0
Ferrari T-R -> 84700.0
Seat Toledo -> 24780.0
Jaguar XK -> 54380.0
Porche GT3 -> 58920.0
```

Módulo mdJarrasMezcla (herencia)

Este módulo usará las clases creadas en el módulo mdJarras. Se trata de construir unas jarras capaces de almacenar agua o vino. Una jarra cuyo contenido es de agua diremos que tiene una pureza de 0 mientras que la que contiene todo vino tiene una pureza de 100. Si una jarra que contiene 3 litros con una pureza de 30 le añadimos 2 litros con una pureza de 80, la jarra finalmente contendrá 5 litros con una pureza de (3*30+2*80)/5 = 50.

Crear la clase `JarraMezcla` en el paquete (`jarrasMezcla`), heredera de Jarra que implemente que además de contenido y capacidad, contiene una variable de tipo `double` que almacena la pureza.

Esta jarra tendrá el siguiente comportamiento:

- Un constructor que crea una jarra mezcla de una capacidad dada. La jarra se crea sin contenido y con `pureza = 0`.
- Sobreescribe el método `public void llena()` para que su comportamiento sea que la jarra se llena de agua.
- Un método `public void llenaVino()` que llena la jarra de vino.
- Sobreescribe el método `public void llenaDesde(Jarra)` para que se interprete al argumento como una jarra de agua.
- Un método `public void llenaDesde(JarraMezcla)` que vuelque la jarra argumento sobre la receptora hasta que se llene una o se vacíe la otra.
- Sobreescribe el método `public String toString()` para que además del contenido y la capacidad, muestre la pureza del líquido de la jarra.

Probar la jarra con el siguiente programa `Main` proporcionado.

Módulo mdHospitalV1L (listas, enum, herencia)

El objetivo de este ejercicio es crear una serie de clases que representen una visión simplificada de la estructura organizativa de los hospitales. En esta estructura intervendrán elementos diversos como: pacientes, médicos, departamentos y hospitales, representados en sendas clases: `Paciente`, `Medico`, `Departamento` y `Hospital`. Tanto pacientes como médicos incluyen características (atributos y métodos) comunes, que deberán establecerse en una clase `Persona`. Cada departamento está compuesto por un conjunto de médicos, que organizaremos en una lista. Y un hospital estará formado por una colección de departamentos, que también organizaremos en una lista.

Las clases y sus relaciones se incluyen en el diagrama que se adjunta a continuación. Los atributos y métodos que se deben incluir en cada clase también se especifican en el diagrama, con el significado que se desprende de forma natural de los nombres utilizados, teniendo en cuenta que:

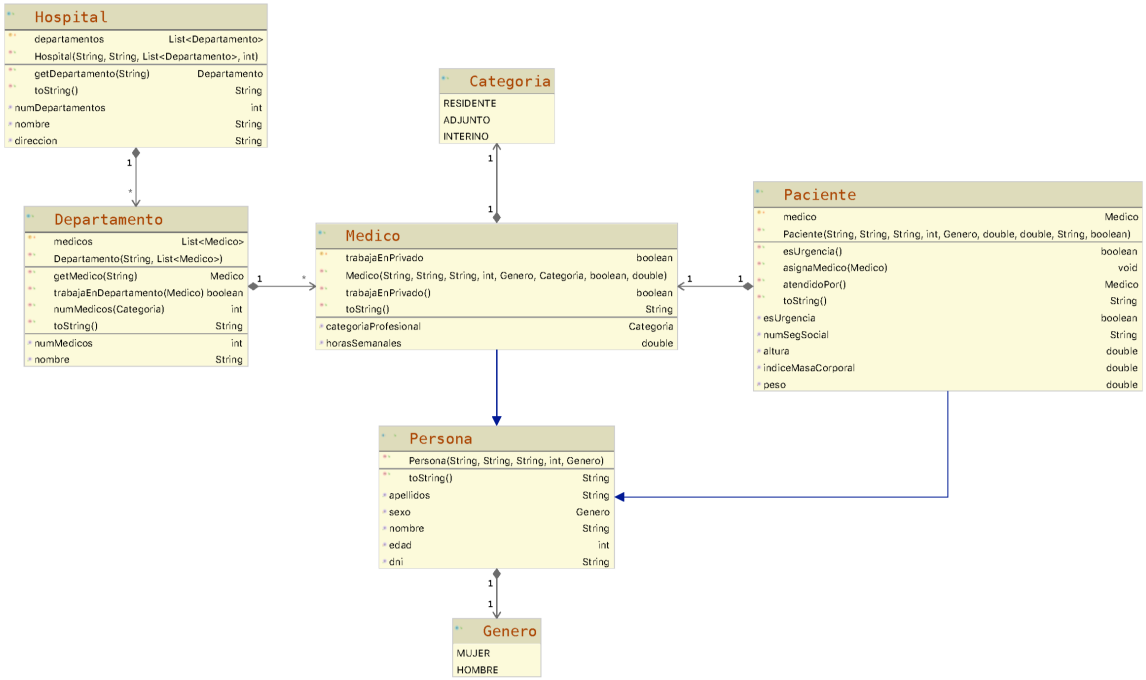
a) El índice de masa corporal en un paciente se calcula dividiendo el peso (en Kg.) por la altura (en metros) al cuadrado.

b) El método `public Medico atendidoPor()` de la clase `Paciente` debe devolver el médico asignado si existe, o `null`, si aún no se ha asignado ningún médico.

c) El método `public int numMedicos(Categoria cat)` de la clase `Departamento` devuelve el número de médicos con la categoría indicada en el argumento.

d) El método `public Medico getMedico(String dni)` de la clase `Departamento` devuelve el objeto Medico con el DNI indicado como argumento.

La práctica consistirá en implementar todas estas clases, cuidando que los identificadores (de clases, variables de instancia y métodos) se correspondan con los que aparecen en el diagrama, y sean acordes a la convención de Java. Obsérvese que la clase `Persona` incluye solo las características propias de una persona física (tal y como se entiende desde un punto de vista jurídico); es decir, lo relevante son los datos jurídicos. Otras características como la altura y el peso son solo relevantes para los pacientes (al menos en el modelo que estamos interesados).



Para probar las clases, utilícese la clase de prueba, `PruebaHospitalV1L`.

Módulo mdHospitalV2L (recorridos, herencia, excepciones)

El objetivo de este ejercicio es completar las clases que representen una visión simplificada de la estructura organizativa de los hospitales y que fue objeto de la práctica `mdHospitalV1L` (Hacer una copia de los ficheros del módulo `mdHospitalV1L` en este módulo). En esta estructura se incorporan nuevos elementos como `Planta, Habitacion` y `Cama` y se modifican `Paciente` y `Hospital.` Así, ahora, un paciente siempre tiene asignada una cama hasta que se dé de alta y un hospital incluye además una colección de plantas. Cada planta incluye una colección de habitaciones y cada habitación incluye una colección de camas. Por su parte, una cama, puede contener a un paciente. Todas las colecciones las organizaremos en listas.

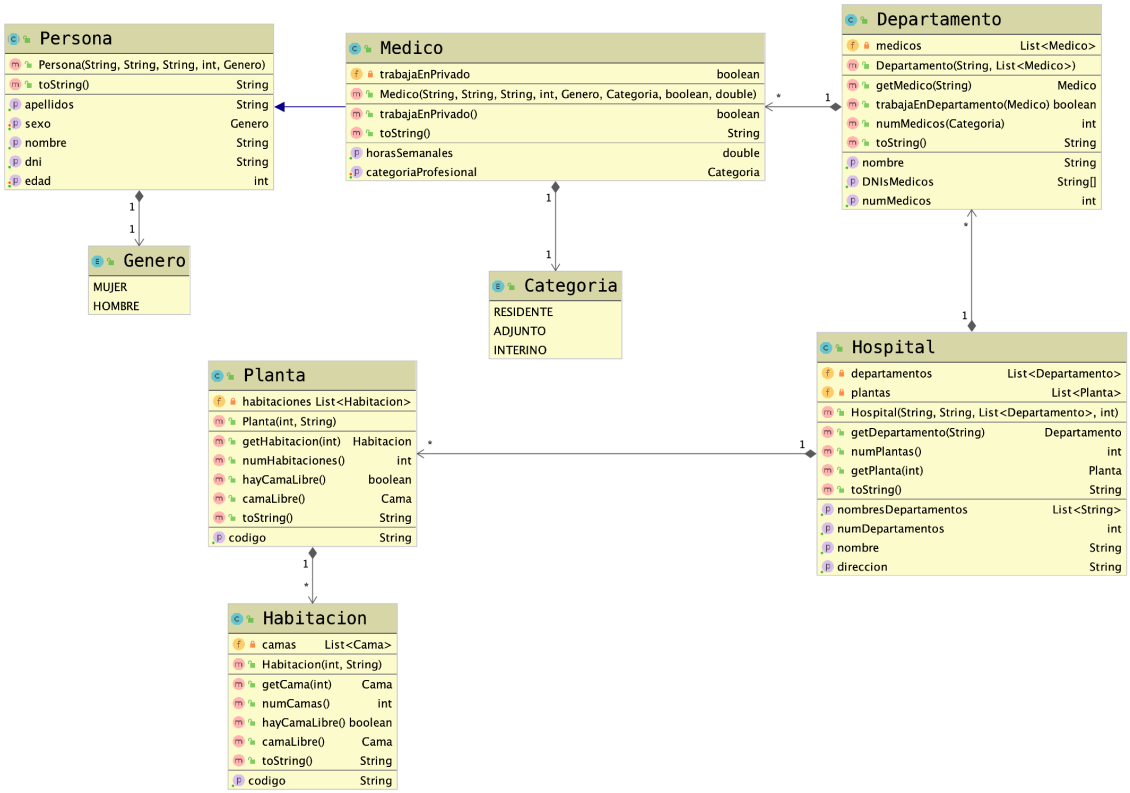
Las clases y sus relaciones se incluyen en el diagrama que se adjunta a continuación. Los atributos y métodos que se deben incluir en cada clase también se especifican en el diagrama, con el significado que se desprende de forma natural de los nombres utilizados, teniendo en cuenta que:

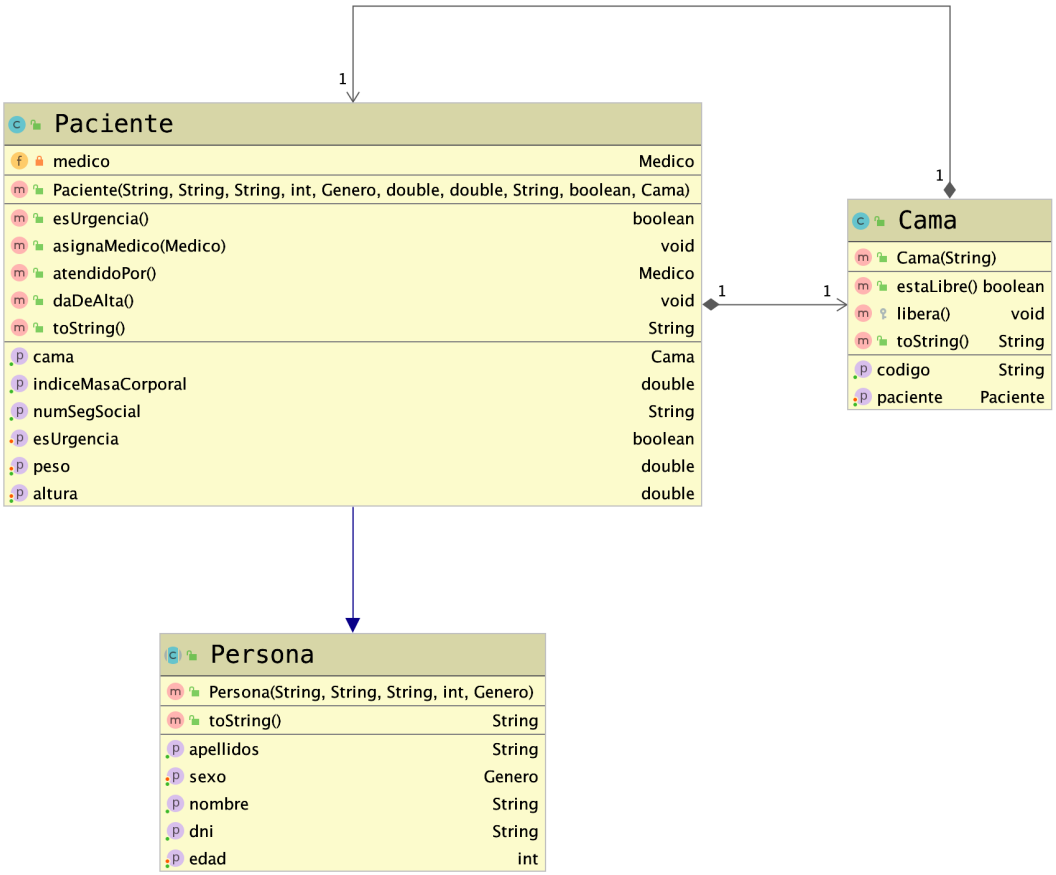
- El constructor de Paciente incluye un parámetro más para proporcionar la cama en la que se instalará el paciente.
- Cada cama puede instalar a lo sumo a un paciente.
- El constructor de hospital incluye un nuevo parámetro que indica el número de plantas que tendrá el hospital. La construcción de un hospital incluye la construcción de sus plantas. La clase de prueba que se proporciona incluye la creación de un hospital con 5 plantas numeradas del 0 al 4 y con códigos "P0", "P1", … "P4".
- La construcción de una planta incluirá la construcción de las habitaciones que contiene. Supondremos que cada planta contiene 8 habitaciones numeradas de 0 a 7 y con códigos dependientes de la planta. Por ejemplo, las habitaciones de la planta 3 tendrán por código "P3H0", "P3H1", …, "P3H7".
- La construcción de una habitación incluye la creación de sus camas. Supondremos 4 camas por habitación numeradas de 0 a 3 y sus códigos dependen de la planta y habitación en la que se encuentren. Por ejemplo, las camas de la habitación 4 de la planta 3 serán la "P3H4C0"; "P3H4C1", "P3H4C2" y "P3H4C3".
- Los métodos `public Planta getPlanta(int i)` de la clase `Hospital` y `public Habitacion getHabitacion(int i)` de la clase `Planta` lanzarán una `RuntimeException` si el argumento no está dentro de los rangos válidos.
- El método `public Cama camaLibre()` definido en las clases `Planta` y `Habitacion` proporcionará una cama cualquiera que esté libre. Lanzará una `RuntimeException` si no hay camas libres en donde se solicita.
- El método `setPaciente(Paciente)` de la clase `Cama` debe asegurarse de que la cama está libre (si no lanzar una `RuntimeException`) y debe mantener consistencia con la clase `Paciente` (asignar al paciente la cama).
- El método `daAlta()` en la clase `Paciente` debe liberar la cama.

La práctica consistirá en modificar las clases `Paciente` y `Hospital` e incluir las nuevas clases según el diagrama adjunto.

Para probar las clases, utilícese la clase de prueba, `PruebaHospitalV2L`. Además, complétese esta clase con nuevos métodos de clase que resuelvan los siguientes problemas:

- Mostrar todos los pacientes que están tratados en una planta dada.
- Mostrar los médicos que tienen asignado paciente.
- Mostrar todas las camas vacías del hospital.
- Mostrar todas las habitaciones vacías del hospital
- Mostrar todas las habitaciones en las que hay alguna cama libre.
- ¿Cuántos pacientes se están tratando en una planta dada?
- ¿Cuántas camas hay libres en una planta dada?
- ¿Están todas las habitaciones del hospital ocupadas por al menos un paciente?
- ¿Tiene un médico dado asignado algún paciente en una planta dada?
- Mostrar todos los departamentos cuyos médicos tienen pacientes en una planta dada.


Persona
Persona(String, String, String, int, Genero)
toString() String
apellidos String
sexo Genero
nombre String
dni String
edad int
Medico
trabajaEnPrivado boolean
Medico(String, String, String, int, Genero, Categoria, boolean, double)
trabajaEnPrivado() boolean
toString() String
horasSemanales double
categoriaProfesional Categoria
Departamento
medicos List<Medico>
Departamento(String, List<Medico>)
getMedico(String) Medico
trabajaEnDepartamento(Medico) boolean
numMedicos(Categoria) int
toString() String
nombre String
DNIsMedicos String[]
numMedicos int
Genero
MUJER
HOMBRE
Categoria
RESIDENTE
ADJUNTO
INTERINO
Hospital
departamentos List<Departamento>
plantas List<Planta>
Hospital(String, String, List<Departamento>, int)
getDepartamento(String) Departamento
numPlantas() int
getPlanta(int) Planta
toString() String
nombresDepartamentos List<String>
numDepartamentos int
nombre String
direccion String
Planta
habitaciones List<Habitacion>
Planta(int, String)
getHabitacion(int) Habitacion
numHabitaciones() int
hayCamaLibre() boolean
camaLibre() Cama
toString() String
codigo String
Habitacion
camas List<Cama>
Habitacion(int, String)
getCama(int) Cama
numCamas() int
hayCamaLibre() boolean
camaLibre() Cama
toString() String
codigo String


Paciente
medico Medico
Paciente(String, String, String, int, Genero, double, double, String, boolean, Cama)
esUrgencia() boolean
asignaMedico(Medico) void
atendidoPor() Medico
daDeAlta() void
toString() String
cama Cama
indiceMasaCorporal double
numSegSocial String
esUrgencia boolean
peso double
altura double
Cama
Cama(String)
estaLibre() boolean
libera() void
toString() String
codigo String
paciente Paciente
Persona
Persona(String, String, String, int, Genero)
toString() String
apellidos String
sexo Genero
nombre String
dni String
edad int

Módulo mdFiltroImagen (librerías, interfaces)

Este módulo pretende manipular imágenes para aplicarles diferentes filtros. Las imágenes serán leídas desde un fichero, se manipularán y se guardarán modificadas en otro. Para realizar este módulo se deben manejar imágenes por lo que previamente comentaremos algo sobre las clases que las manejan. Todas las clases se deben crear en el paquete `filtraImagen`.

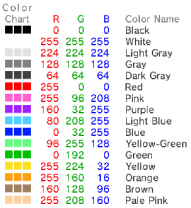
Leer una imagen desde fichero. - La clase `javax.imageio.ImageIO` dispone del método estático `read(File fichero)` (para generar un File usa path.toFile()); si su argumento es un fichero que contiene una imagen devuelve un objeto `BufferedImage` del paquete (`java.awt.image`) que contiene la información de la imagen leída del fichero. Los formatos admitidos de imagen son los habituales (extensiones `gif`, `jpg`, `jpeg`, `tif`, `tiff`, `png`, etc.).

Este método lanza la excepción comprobada `IOException` si hay algún error en la lectura. Para construir un `File` desde el camino un `String` hay que usar el constructor `File(String)` o crea un `Path` y aplicale el método `toFile()`.

Guardar una imagen. - Nuevamente, la clase `javax.imageio.ImageIO` dispone del método estático `write(BufferedImage image, String ext, File fichero)`. Este método tiene tres argumentos: el primero es el objeto `BufferedImage` con el contenido de la imagen a guardar; el segundo una cadena de caracteres que describe la extensión que tendrá el nombre del fichero a guardar, y el tercero un objeto `File` que identifica al fichero donde se guardará (incluida la extensión). Este método lanza la excepción comprobada `IOException` si hay algún error en la escritura.

Manipular una imagen. - Para realizar los procesos de modificación de una imagen la clase `BufferedImage` proporciona métodos que permiten conocer la anchura de la imagen (`int getWidth()`), su altura (`int getHeight()`) y el color de cada píxel (`int getRGB(int x, int y)`). De igual forma, permite modificar el color de un píxel con el método `void setRGB(int x, int y, int color)`. El color se representa como un entero, pero es posible construir un objeto `Color` para separar sus componentes (rojo, verde, azul):

```
Color color = new Color(image.getRGB(x, y));
int rojo  = color.getRed(); // entre 0 y 255
int verde = color.getGreen(); // entre 0 y 255
int azul  = color.getBlue(); // entre 0 y 255
```



También se puede crear un objeto color a partir de los valores rojo verde y azul y luego convertirlo a un entero para colocarlo en un píxel de la imagen.

```
Color color = new Color(15,0,255); //rojo, verde, azul
image.setRGB(x, y, color.getRGB());
```

Filtros

Una vez que disponemos de una imagen (un objeto `BufferedImage`) vamos a proporcionar la facilidad de modificarla por medio de la aplicación de filtros. Para disponer de un comportamiento común para todos los filtros se crea una interfaz `FiltroImagen`, que implementarán todos los filtros:

```
import java.awt.image.BufferedImage;
public interface FiltroImagen {
            void filtra(BufferedImage im);
}
```

Es decir, un filtro debe responder al mensaje `filtra` que toma como argumento un objeto `BufferedImage`. Ese `BufferedImage` contiene toda la información de la imagen que queremos filtrar y el filtrado se hace sobre el propio objeto.

Por ejemplo, un filtro que sólo deja pasar el color azul puede definirse como:

```
import java.awt.Color;
import java.awt.image.BufferedImage;

public class FiltroAzul implements FiltroImagen {

         public void filtra(BufferedImage imagen) {
            int fWidth  = imagen.getWidth();
            int fHeight = imagen.getHeight();

            for (int x = 0; x < fWidth ; x++) {
               for (int y = 0; y < fHeight; y++) {
                  int azul = new Color(imagen.getRGB(x, y)).getBlue();
                  imagen.setRGB(x, y, new Color(0, 0, azul).getRGB());
               }
            }
         }
}
```

***Ejercicio*:** Crear otro filtro que intercambie los componentes azul y rojo de cada píxel, `FiltroAzulPorRojo`, y otro que sustituya cada componente de cada píxel por la media de los tres componentes que ese píxel tiene, `FiltroMedia`. Definir el filtro `FiltroQuitaManchas` que sustituya por el color blanco (255, 255, 255) cualquier pixel cuyo color tenga de componentes azul, rojo y verde un valor mayor de 200.

Probar los filtros con la aplicación `MainFiltroImagen`.

Filtros Java para `BufferedImage`

Java dispone de clases (predefinidas) que permiten aplicar un tipo particular de filtros a una imagen a través de un objeto `BufferedImage`.

El filtro consistirá en utilizar para modificar cada píxel la información de su color y el de sus vecinos. Para ello, se construye una matriz de pesos como por ejemplo la siguiente de tres por tres:

0.25f	0.0f	0.25f
0.0f	0.0f	0.0f
0.25f	0.0f	0.25f

La matriz representa el peso que tendrá el color del píxel a modificar en el resultado en función de su color y el de sus vecinos. Este ejemplo indica que cada píxel tomará el color como la media de los 4 colores de los píxeles vecinos que están en las esquinas. Si no queremos modificar la luminosidad total de la imagen, la suma de los valores de la matriz debe ser 1. Un valor mayor que uno aumentará el brillo de la imagen mientras que un valor menor que 1 la oscurecerá.

Aquí se muestra un ejemplo de cómo crear un filtro que utilice esa matriz:

```
import java.awt.image.BufferedImage;
import java.awt.image.BufferedImageOp;
import java.awt.image.ConvolveOp;
import java.awt.image.Kernel;

public class FiltroSuavizado implements FiltroImagen {
    private float[] mascara =
        {1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f,
         1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f,
         1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f};
    private int dimension = 3;
    public void filtra(BufferedImage image) {
          Kernel kernel = new Kernel(dimension, dimension,mascara);
          BufferedImageOp bright = new ConvolveOp(kernel);
          BufferedImage convolvedImage =bright.filter(image, null);
          image.getGraphics().drawImage(convolvedImage, 0, 0, null);
    }
}
```

Básicamente, se crea un objeto `Kernel` que contiene a la matriz dada indicando su dimensión; luego se genera un objeto capaz de ejecutar el filtro con el `kernel` apropiado:

```
          BufferedImageOp bright = new ConvolveOp(kernel);
```

y se realiza el filtrado creando un nuevo `BufferedImage`:

```
          BufferedImage convolvedImage = bright.filter(image, null);
```

Por último, quedaría copiar el resultado en la propia imagen ya que los filtros que nuestro programa trata así lo requieren:

```
          image.getGraphics().drawImage(convolvedImage, 0, 0, null);
```

que copia el contenido de `convolvedImage` en `image` a partir de la esquina superior izquierda.

En lugar de implementar la clase `FiltroSuavizado`, vamos a crear una clase que valga para cualquier filtro de matriz que se nos ocurra. Sea `FiltroMatriz`:

```
import java.awt.image.BufferedImage;
import java.awt.image.BufferedImageOp;
import java.awt.image.ConvolveOp;
import java.awt.image.Kernel;

public class FiltroMatriz implements FiltroImagen {
    private int dimension;
    private float[] mascara;

    public FiltroMatriz(int d, float[] mas) {
        dimension = d;
        mascara = mas;
    }

    public void filtra(BufferedImage image) {
```

```
        Kernel kernel = new Kernel(dimension, dimension, mascara);
        BufferedImageOp bright = new ConvolveOp(kernel);
        BufferedImage convolvedImage = bright.filter(image, null);
        image.getGraphics().drawImage(convolvedImage, 0, 0, null);
    }

    public static FiltroMatriz creaFiltroMedia() {
        float[] mascara =
                {1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f,
                 1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f,
                 1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f};
        return new FiltroMatriz(3, mascara);
    }

}
```

Ahora solo hay que crear un objeto de esta clase con el constructor pasándole la dimensión de la matriz y la propia matriz.

Para facilitar la creación vamos a crear métodos factoría. Son métodos estáticos que crean objetos de la clase. Así `creaFiltroMedia` crea un objeto `FiltroMatriz` proporcionándole la dimensión y la matriz que asocia a cada pixel la media de los valores de los pixeles vecinos incluyendo al propio pixel.

***Ejercicio*:** Crear dos métodos factoría, una para un filtro de bordes (método `creaFiltroBordes`), que crea un objeto `FiltroMatriz` proporcionando la matriz de resalte de bordes de dimensión 3 siguiente:

```
{-1.0f, -1.0f, -1.0f,
 -1.0f,  9.0f, -1.0f,
 -1.0f, -1.0f, -1.0f};
```

El otro método factoría (método `creaFiltroBrillo`) crea un objeto `FiltroMatriz` con la matriz de dimensión 1,

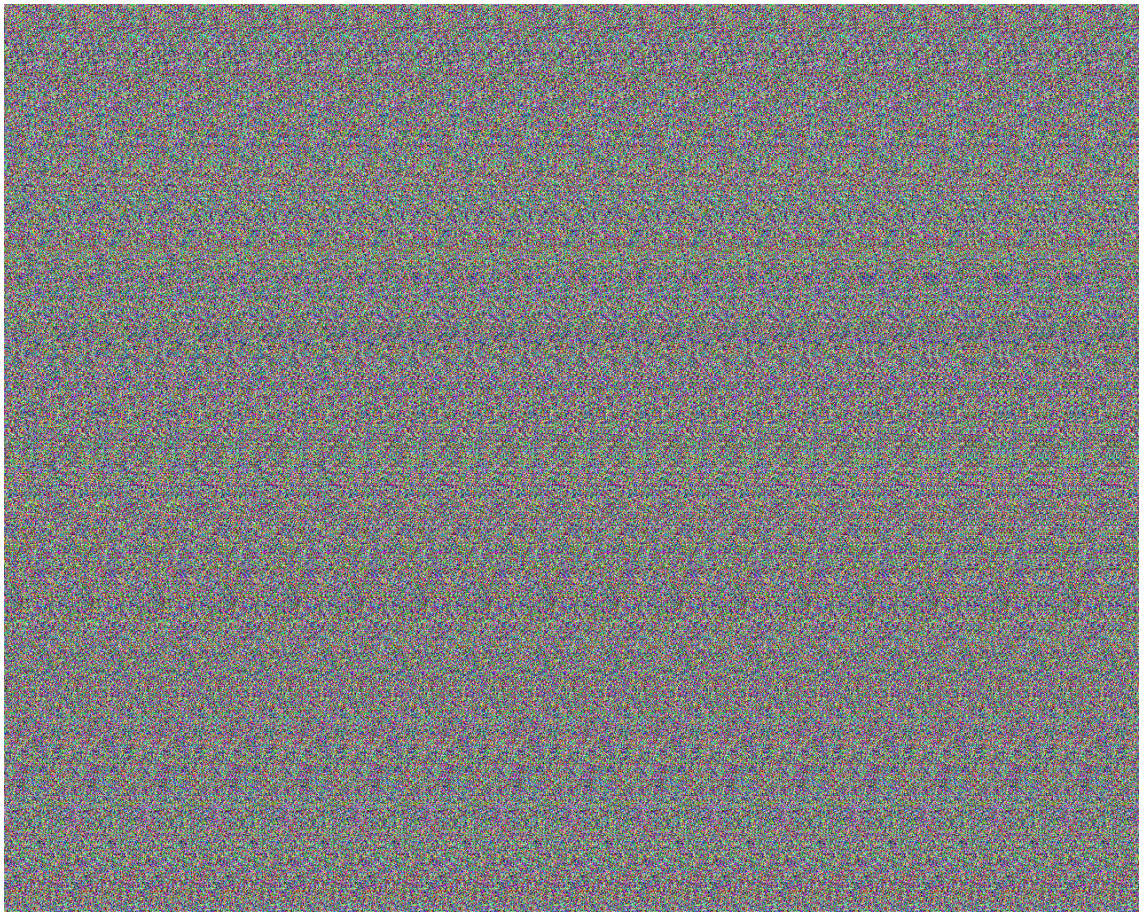
```
{1.2f};
```

En este último caso, al sumar los elementos de la matriz mas de 1, la luminosidad de la imagen aumenta.

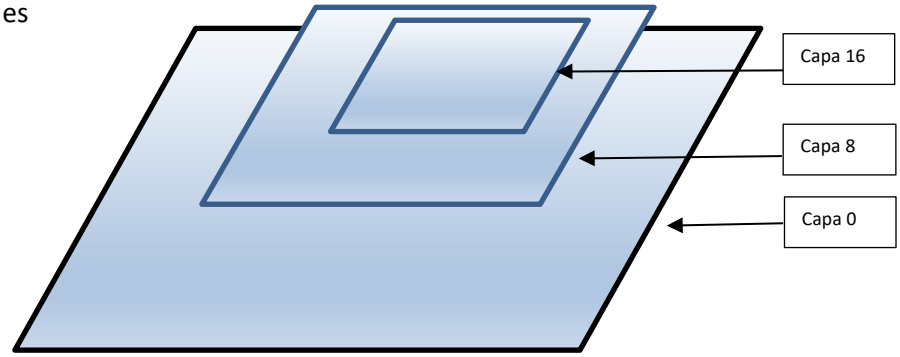
De esta forma podemos construir cualquier filtro que se nos ocurra.

Estereograma

Un estereograma es una imagen en la que aparentemente cada pixel muestra un color aleatorio, pero en realidad hay una imagen distribuida por capas. Si suponemos que una imagen dispone de 16 capas, cada capa representa una profundidad de visión del píxel en la imagen. La capa 0 es la mas lejana a nuestros ojos (el fondo de la pantalla) y la capa 16 sería la mas cercana. El siguiente gráfico es un estereograma donde se ven dos rectángulos, el mayor sobre la capa 8 en el menor sobre la capa 16.



De lado su forma es



A continuación, se muestra cómo se genera un estereograma:

- Sobre una imagen se selecciona la banda izquierda con una anchura de aproximadamente 80 píxeles (sea este valor `tam`). Esa franja se colorea con colores aleatorios para cada píxel.
- Para el resto de los píxeles de la imagen se realiza lo siguiente:
    - Si un píxel debe verse en la capa `n`, entonces asígnale el color del píxel que se encuentra en la misma fila y `tam - n` píxeles a la izquierda.

Por ejemplo, el siguiente método crea un estereograma de tamaño 1200 por 800 píxeles y donde todos los píxeles están en la capa 0 excepto los que se encuentran en el rectángulo de esquina superior izquierda (200,300) y de esquina inferior derecha (800,600) que están a la capa 8:

```
public static BufferedImage creaStereogramaEjemplo() {
    BufferedImage stereograma = new BufferedImage(1200, 800, BufferedImage.TYPE_3BYTE_BGR);
    int fWidth = stereograma.getWidth();
    int fHeight = stereograma.getHeight();
    int tamTrama = 60;

    // Creamos una trama inicial a la izquierda
    for (int x = 0; x < tamTrama; x++) {
        for (int y = 0; y < fHeight; y++) {
            Color color = new Color(   aleatorio.nextInt(256),
                    aleatorio.nextInt(256),
                    aleatorio.nextInt(256));
            stereograma.setRGB(x, y, color.getRGB());
        }
    }
    // Completamos siguiendo el algoritmo
    for (int x = tamTrama; x < fWidth; x++) {
        for (int y = 0; y < fHeight; y++) {
            int capa = 0;
            if (200 < x && x < 800 && 300 < y && y < 600) {
                capa = 8;
            }
            int nx = x - tamTrama + capa;
            stereograma.setRGB(x, y, estereograma.getRGB(nx, y ));
        }
    }
    return stereograma;
}
```

Para no tener que calcular qué píxeles deben ir en cada capa podemos tomar una imagen de referencia que tenga colores. Se lee esa imagen de referencia y se calcula para cada píxel del estereograma su capa de la siguiente manera:

- Se genera la banda izquierda, es decir, las primeras `80` columnas (o lo que indique la variable `tam`) con colores aleatorios.
- Para el resto, se toma el pixel correspondiente de la imagen de referencia y se obtiene su color azul (para tener un valor de `0` a `255`) y lo restamos a `255`. Así, el blanco será `0` y el negro `255`. Ahora, valores del `0` al `15` irán a la capa `0`, del `16` al `31` a la capa `1`, etc. (suponiendo que hay `16` capas). Es decir:

```
int capa = (255 - azul) / (255/numCapas)
```

Nota: La imagen de referencia debe representar los colores de tal forma que lo que irá al fondo debe tener un color mas claro. Cuanto mas alta sea la capa a la que va un píxel, mas oscuro será el color de ese píxel (principalmente en la gama de azules).

***Ejercicio*:** Crea la clase `FiltroStereograma` para crear estereogramas. La clase implementará la interfaz `FiltroImagen` y dispondrá de dos constantes,

```
private static final int TAM_TRAMA = 80;
private static final int NUM_CAPAS = 16;
```

y una variable de clase para guardar una variable aleatoria

```
private static final Random aleatorio = new Random();
```

Además, tendrá dos variables de instancia:

```
private int tamTrama;  // Tamaño de la banda izquierda
private int numCapas;  // número de capas
```

Y los siguientes constructores y métodos:

```
public FiltroStereograma(int tamTrama, int numCapas)
```

Crea un objeto para crear estereogramas con un tamaño de trama y número de capas dados.

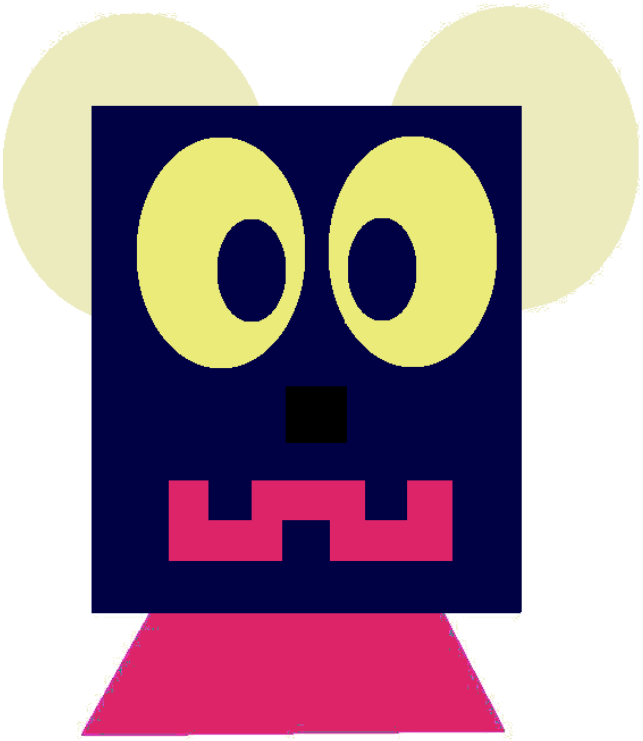
```
public FiltroStereograma()
```

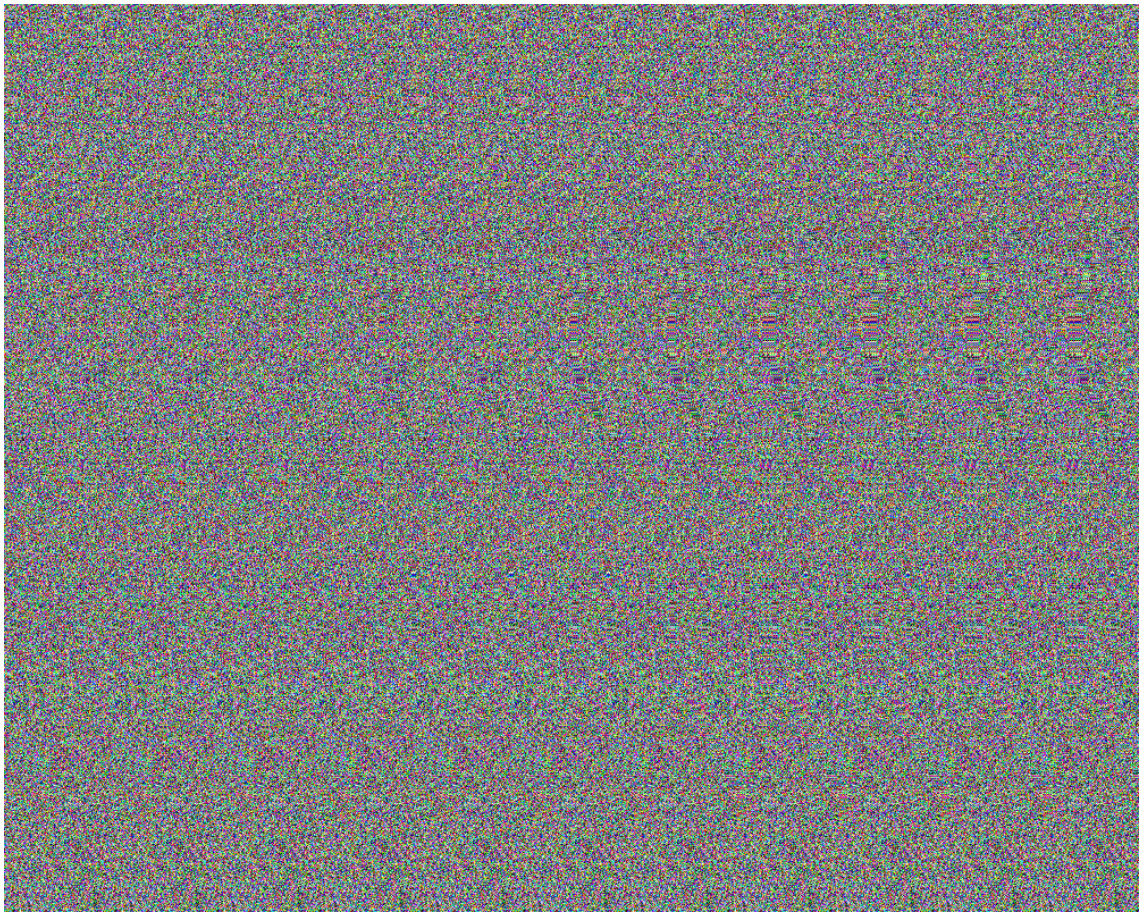
Crea un estereograma con los valores de tamaño de trama y número de capas por defecto.

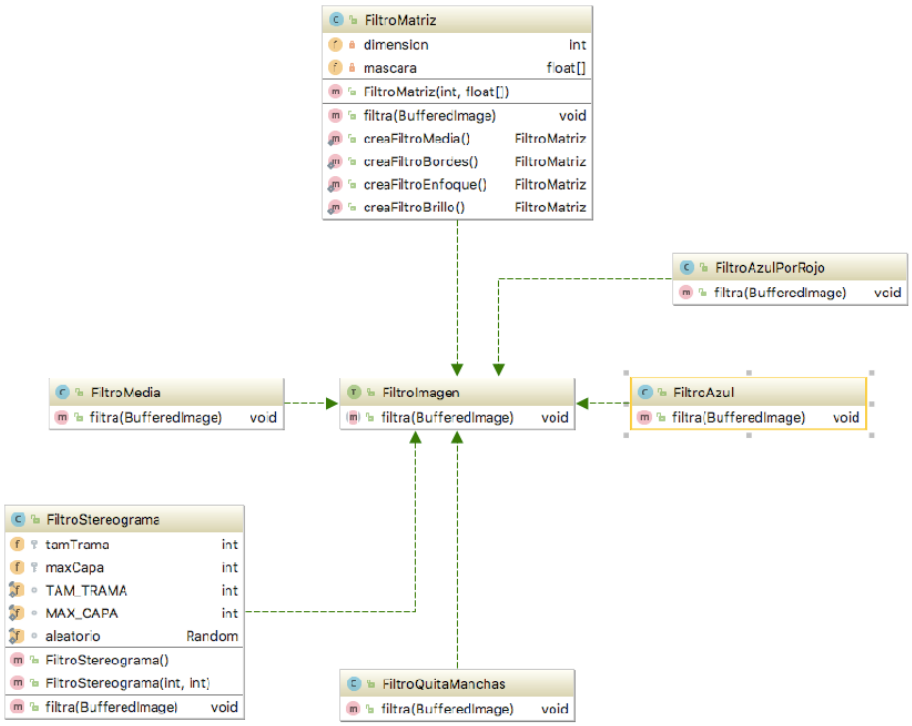
```
public void filtra(BufferedImage imagen)
```

Crea el estereograma como se ha descrito anteriormente.

Aquí se presenta una imagen de referencia y después el estereograma calculado con los valores por defecto de anchura de banda y capas.






FiltroMatriz
dimension int
mascara float[]
FiltroMatriz(int, float[])
filtra(BufferedImage) void
creaFiltroMedia() FiltroMatriz
creaFiltroBordes() FiltroMatriz
creaFiltroEnfoque() FiltroMatriz
creaFiltroBrillo() FiltroMatriz
FiltroAzulPorRojo
filtra(BufferedImage) void
FiltroMedia
filtra(BufferedImage) void
FiltroImagen
filtra(BufferedImage) void
FiltroAzul
filtra(BufferedImage) void
FiltroStereograma
tamTrama int
maxCapa int
TAM_TRAMA int
MAX_CAPA int
aleatorio Random
FiltroStereograma()
FiltroStereograma(int, int)
filtra(BufferedImage) void
FiltroQuitaManchas
filtra(BufferedImage) void

Módulo mdGenetico (herencia, interfaces)(*mandatory*)

Los algoritmos genéticos se utilizan para resolver problemas de optimización de forma aproximada cuando no es posible hacerlo mediante algoritmos exactos; están inspirados en la teoría de la evolución y trabajan manipulando un conjunto de soluciones tentativas para un problema. A dichas soluciones se les denomina *individuos* y al conjunto de soluciones *población*. En el caso más simple, que será el que nosotros trataremos, un individuo está formado por una cadena binaria (secuencia de ceros y unos) y tiene asociado un valor real que representa, de algún modo, su aproximación a una solución óptima, conocido como su *aptitud* (*fitness*) para el problema. A cada dígito de la secuencia binaria se le llama *gen* y a la secuencia completa *cromosoma*. La única información que necesitan estos algoritmos sobre el problema que han de resolver es la forma de calcular la aptitud de una solución, lo que se conoce como la función de fitness.

Un algoritmo genético procede del siguiente modo. En primer lugar, todos los individuos de la población son inicializados con secuencias aleatorias de ceros y unos. A continuación, el algoritmo entra en un bucle cuyo cuerpo se repite un número determinado de veces (*generaciones*). Dentro de este bucle, dos individuos (padres) son elegidos aleatoriamente de la población, se *recombinan* sus cromosomas para dar a luz a un nuevo cromosoma que, seguidamente, es *mutado*. Con el cromosoma resultante se forma un nuevo individuo que se compara con el peor individuo de la población y, si su fitness es más alto, lo sustituye; de lo contrario, se desecha.

En los casos en que se presente alguna situación excepcional se deberá lanzar la excepción del tipo `RuntimeException` (o de alguna clase heredera de ella que se adecue mejor a la situación en cuestión).

Se construirán las siguientes clases e interfaces en el paquete `genetico`:

1) La clase `Cromosoma` con los métodos que aparecen en el diagrama adjunto, teniendo en cuenta lo siguiente:

Para almacenar los datos de un cromosoma se debe usar un array de enteros, cada una de cuyas posiciones representará un gen. Dicho array debe ser visible en subclases y clases dentro del paquete.

El constructor de la clase `Cromosoma` tomará como parámetros la longitud del mismo (número de genes que contiene) y un valor booleano que indica si debe asignar de forma aleatoria un valor cero o uno a cada gen (caso true), o inicializarlos con el valor por defecto 0 (caso false).

Los métodos `getter` y `setter` con nombre `gen` permitirán consultar y establecer el valor de cada gen del cromosoma, y `getLongitud()` permitirá consultar el número de genes.

El método `mutar(double)` debe recorrer todos los genes del cromosoma que se le pasa como parámetro invirtiendo su valor (de 0 a 1 ó de 1 a 0) de acuerdo con la probabilidad de mutación que recibe como parámetro, y que deberá aplicar a cada gen por separado.

El método `copia()` deberá construir y devolver una copia del cromosoma.

2) La interfaz `Problema` con un único método, `evalua(Cromosoma)`, que debe devolver

el valor de fitness asociado al cromosoma que recibe como argumento en el problema que representa.

3) La clase `Individuo`, formada por un `Cromosoma` y su valor de `fitness`, teniendo en cuenta lo siguiente:

a. Esta clase deberá tener dos constructores: el primero con dos parámetros, la longitud del cromosoma y el problema a resolver; y el segundo, también con dos parámetros, que ahora son el cromosoma que debe asociarse al individuo y el problema. El primero debe crear un cromosoma de forma aleatoria, y el segundo debe generar una copia del cromosoma que recibe como parámetro antes de asociarlo al objeto que está creando. En ambos constructores el objeto de clase `Problema` se usará únicamente para evaluar el cromosoma y asociarle el correspondiente valor de `fitness` (en dicho problema).

b. El método `Cromosoma getCromosoma()` debe devolver una copia del objeto de la clase `Cromosoma` asociado al objeto de clase `Individuo`.

c. El valor de `fitness` de un objeto `Individuo` puede consultarse con el método `getFitness()`.

4) La clase `OneMax` que implementa la interfaz `Problema` sabiendo que, en este problema, el `fitness` de un individuo viene dado por el número de cifras "1" que contiene su cromosoma y el `CeroMax` es igual pero con los "0".

5) La clase `Poblacion`, teniendo en cuenta que una población está constituida por un número fijo de individuos y se implementará usando un array de `Individuo`; además:

a. El constructor recibirá como parámetros: el tamaño de la población (número de individuos), la longitud de los individuos (número de genes de sus cromosomas) y el problema a resolver (un objeto que implementa la interfaz `Problema`) y, con estos datos, deberá crear una población de individuos generados de forma aleatoria.

b. El método `getNumIndividuos()` debe devolver el número de individuos de la población (tamaño de la población).

c. El método `getIndividuo(int)` devolverá el *i*-ésimo individuo de la población.

d. El método `mejorIndividuo()` devuelve el individuo con mayor valor de fitness de la población.

e. El método `reemplaza(Individuo)` sustituirá el peor individuo de la población (el de menor valor de fitness) por el individuo que se pasa como parámetro; pero sólo en el caso de que este último sea mejor (mayor fitness).

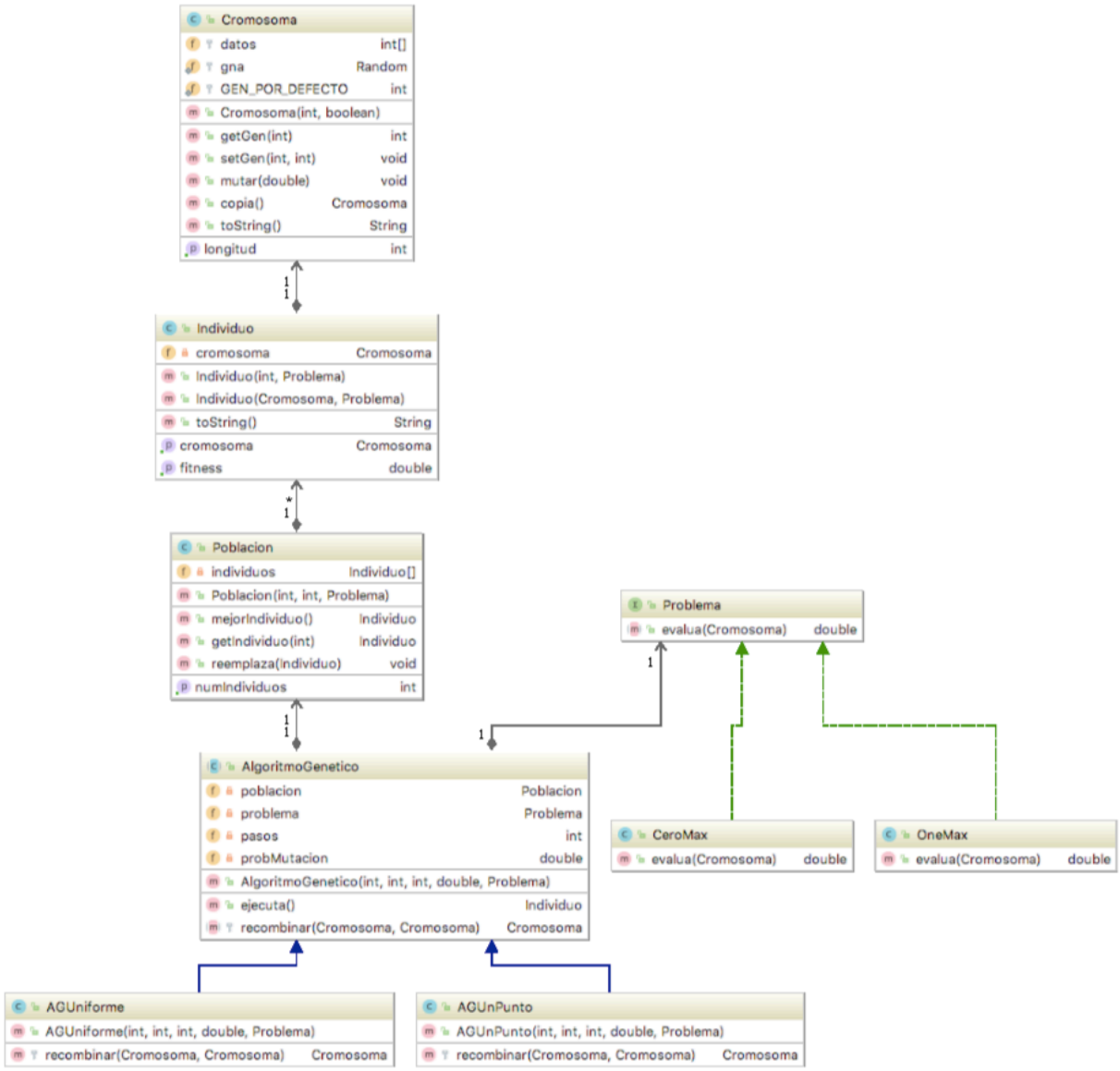
6) La clase abstracta `AlgoritmoGenetico`, sabiendo que cada algoritmo genético almacenará información necesaria sobre un problema, una población de soluciones tentativas, la probabilidad de mutación de los genes de los individuos y el número de pasos que debe realizar. Además, debe tenerse en cuenta lo siguiente:

a. El constructor de esta clase debe recibir el tamaño de la población que va a utilizar, la longitud de los individuos de dicha población, el número de pasos del algoritmo (generaciones), la probabilidad de mutar un gen en el cromosoma y el problema que se debe resolver. El constructor deberá crear la población de individuos.

b. El método `ejecuta()` deberá ejecutar, tantas veces como indique el número de pasos, la secuencia siguiente: seleccionar dos individuos de la población aleatoriamente, tomar sus cromosomas y recombinarlos usando el método abstracto `recombinar(Cromosoma, Cromosoma)`, mutar el resultado con la probabilidad indicada y, por último, crear un individuo con el cromosoma resultante que se insertará en la población reemplazando al peor individuo siempre y cuando sea mejor que éste. Finalmente, devolverá el mejor individuo de la población después de la terminación del bucle.

7) Existen muchos operadores de recombinación que pueden utilizarse para recombinar individuos. Dos de los más conocidos son la *recombinación de un punto* y la *recombinación uniforme*. En el primero se genera un número aleatorio *z* comprendido entre cero y la longitud del cromosoma. Los primeros *z* genes del individuo resultante se toman del primer padre y el resto del segundo. En la recombinación uniforme el valor de cada gen del individuo resultante se elige de forma aleatoria de uno de los padres. Dicho lo anterior, se deberá construir las subclases `AGUnPunto` y `AGUniforme` de `AlgoritmoGenetico` de forma que implementen la recombinación de un punto y la recombinación uniforme, respectivamente, en el método `recombinar`. El constructor de dichas clases debe tener la misma signatura que el de la clase `AlgoritmoGenetico`.

Para probar el funcionamiento de los dos tipos de algoritmos genéticos implementados se puede usar la clase `TestGenetico`.


Cromosoma
datos int[]
gna Random
GEN_POR_DEFECTO int
Cromosoma(int, boolean)
getGen(int) int
setGen(int, int) void
mutar(double) void
copia() Cromosoma
toString() String
longitud int
1
1
Individuo
cromosoma Cromosoma
Individuo(int, Problema)
Individuo(Cromosoma, Problema)
toString() String
cromosoma Cromosoma
fitness double
*
1
Poblacion
individuos Individuo[]
Poblacion(int, int, Problema)
mejorIndividuo() Individuo
getIndividuo(int) Individuo
reemplaza(individuo) void
numIndividuos int
1
1
Problema
evalua(Cromosoma) double
1
1
AlgoritmoGenetico
poblacion Poblacion
problema Problema
pasos int
probMutacion double
AlgoritmoGenetico(int, int, int, double, Problema)
ejecuta() Individuo
recombinar(Cromosoma, Cromosoma) Cromosoma
CeroMax
evalua(Cromosoma) double
OneMax
evalua(Cromosoma) double
AGUniforme
AGUniforme(int, int, int, double, Problema)
recombinar(Cromosoma, Cromosoma) Cromosoma
AGUnPunto
AGUnPunto(int, int, int, double, Problema)
recombinar(Cromosoma, Cromosoma) Cromosoma