**Asignatura /módulo: Programación en Java**

**Curso 2020/21**

## COLECCIÓN DE EJERCICIOS NÚMERO 10

**Interfaces y Colecciones de elementos**

*Todos los ejercicios se realizarán escribiendo los comentarios que se consideren adecuados.*

*ES muy recomendable dibujar los diagramas de clase correspondientes antes de resolver los ejercicios.*

### Ejercicio 1:

Escribir el código de una interfaz genérica **Operable<T>** , que declarará 4 operaciones, las cuales recibirán como argumento un objeto de la clase genérica T y devolverán otro objeto de la misma clase T. Estas 4 operaciones serán:

- *suma (T elemento) : T*
- *resta (T elemento): T*
- *producto (T elemento): T*
- *cociente (T elemento): T*

Cada uno de estos métodos realiza una operación entre el objeto que recibe el método y el objeto que se pasa como argumento (ambos pertenecen lógicamente a la clase T).

A continuación, escribir el código de la clase **Fraccion**, que implementará la interfaz *Operable<T>* y dispondrá de las siguientes propiedades:

- ➔ *numerador* : entero
- ➔ *denonimador*: entero y distinto de 0

Tendrá un solo constructor *Fraccion (int num, int denom)* que inicializará las propiedades. Lanzará una excepción si se intenta crear un objeto con denominador 0.

Implementará las operaciones definidas en la interfaz Operable, de acuerdo con las reglas habituales de las operaciones con fracciones:

Suma: $\frac{a}{b}+\frac{c}{d}=\frac{a.d+b.c}{b.d}$

Resta: $\frac{a}{b}-\frac{c}{d}=\frac{a.d-b.c}{b.d}$

Producto: $\frac{a}{b}.\frac{c}{d}=\frac{a.c}{b.d}$

Cociente $(\frac{a}{b})\div(\frac{c}{d})=\frac{a.d}{b.c}$

En cada método recibido de la interfaz, creará y devolverá un nuevo objeto de la clase Fraccion con el resultado de la operación.

Además, incorporará un método *toString()* que devolverá la cadena "numerador / denominador".

**Ejercicio 2:**

Comenzaremos creando una clase a la que se llamará "MensajeCorto" y que recibirá las propiedades siguientes:

- mensaje (texto con un máximo de 160 caracteres)
- telefono (constará de un mínimo de 9 dígitos, sin límite máximo)
- correo ( obedecerá al patrón " nombre@subdominio.dominio," con al menos 2 niveles de dominio -es decir, no es válido un correo de la forma "pepito@es " o "pepito@gmail" con un solo nivel de dominio-

Todas las propiedades son de tipo cadena

La clase tendrá un constructor único para validar e inicializar todos los parámetros

En la clase "MensajeCorto" definiremos los métodos getXxX() y setXxX() de todas las propiedades.
Los métodos setXxX() validarán sus respectivas propiedades (usando expresiones regulares, en caso necesario). Si falla la validación, crearán una excepción de tipo IllegalArgumentException.

A continuación, definiremos una interface llamada "Facturable" , con un método facturar() que devolverá un valor decimal y en su implementación por defecto (default) devolverá 0.0

La clase "MensajeCorto" implementará la interfaz anterior pero no desarrollará el método facturar(). Por tanto, debe convertirse en clase abstracta.

De la clase "MensajeCorto" heredarán las dos clases siguientes:

- "MensajeSMS"
- "MensajeEmail"

“MensajeSMS” desarrollará su propia versión del método facturar() que devolverá el siguiente importe:  0,15 euros si el teléfono es nacional y  0,45 si es extranjero.  Se tratan como extranjeros todos los números telefónicos que comiencen por “0”.

“MensajeEmail” usará la versión de facturar() definida en la interfaz.

Se creará por último un método llamado  “enviarMensaje()” cuyo comportamiento será el siguiente:

Para la clase MensajeEmail:

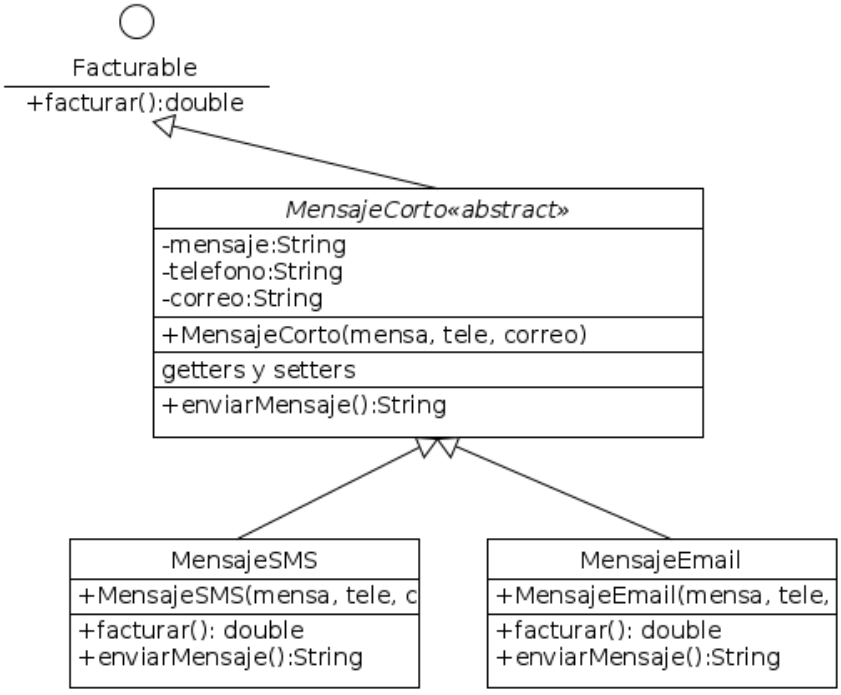
- Si el campo “mensaje” está vacío, mostrará la advertencia “El texto del mensaje está vacío”.
- En caso contrario, mostrará un mensaje de confirmación del tipo “Mensaje enviado a la dirección electrónica <correo>” (sustituyendo <correo> por su valor)

Para la clase MensajeSMS:

- Si el campo “mensaje” está vacío, mostrará la advertencia “El texto del mensaje está vacío”.
- En caso contrario, mostrará un mensaje de confirmación del tipo “Mensaje enviado al número <telefono> con un coste de NN euros” (sustituyendo <telefono> por su valor).  El coste será el devuelto por el método facturar()

*Sugerencia*: Definir el método enviarMensaje como abstracto en la clase padre.

Diagrama de clases para el Ejercicio:




**Ejercicio 3:**

Confeccionar una aplicación Java para gestionar las publicaciones de una biblioteca. Estas publicaciones pueden ser de dos tipos: libros o revistas.

- ➔ Tanto para libros como para revistas se guardan como atributos comunes: el código, título y año de publicación. Estos atributos se inicializan en el constructor del correspondiente objeto.
- ➔ Comprobaremos que el año no es anterior a 1500 ni posterior al año correspondiente a la fecha actual
- ➔ Los libros tienen además un atributo booleano “prestado” y un atributo “fechaDevolucion” (que pertenecerá a la clase LocalDate). Un nuevo objeto de Libro tomá el valor “false” para el atributo “prestado”.
- ➔ Tanto libros como revistas tendrán un método toString() que devuelve su estado (código, título y año de publicación).
  Para los libros, el método toString() devuelve también la fecha de devolución (si el libro está prestado), o el mensaje “No Prestado” (si no lo estuviera)

Por último, los libros (no las revistas) implementarán una interfaz llamada **Prestable**, la cual constará de los siguientes métodos:

- prestar (): devuelve un objeto de la clase LocalDate

- devolver ():  sin retorno
- prestado (): devuelve un objeto de la clase LocalDate

Al implementar los métodos de la  interfaz, se tendrá en cuenta lo siguiente:

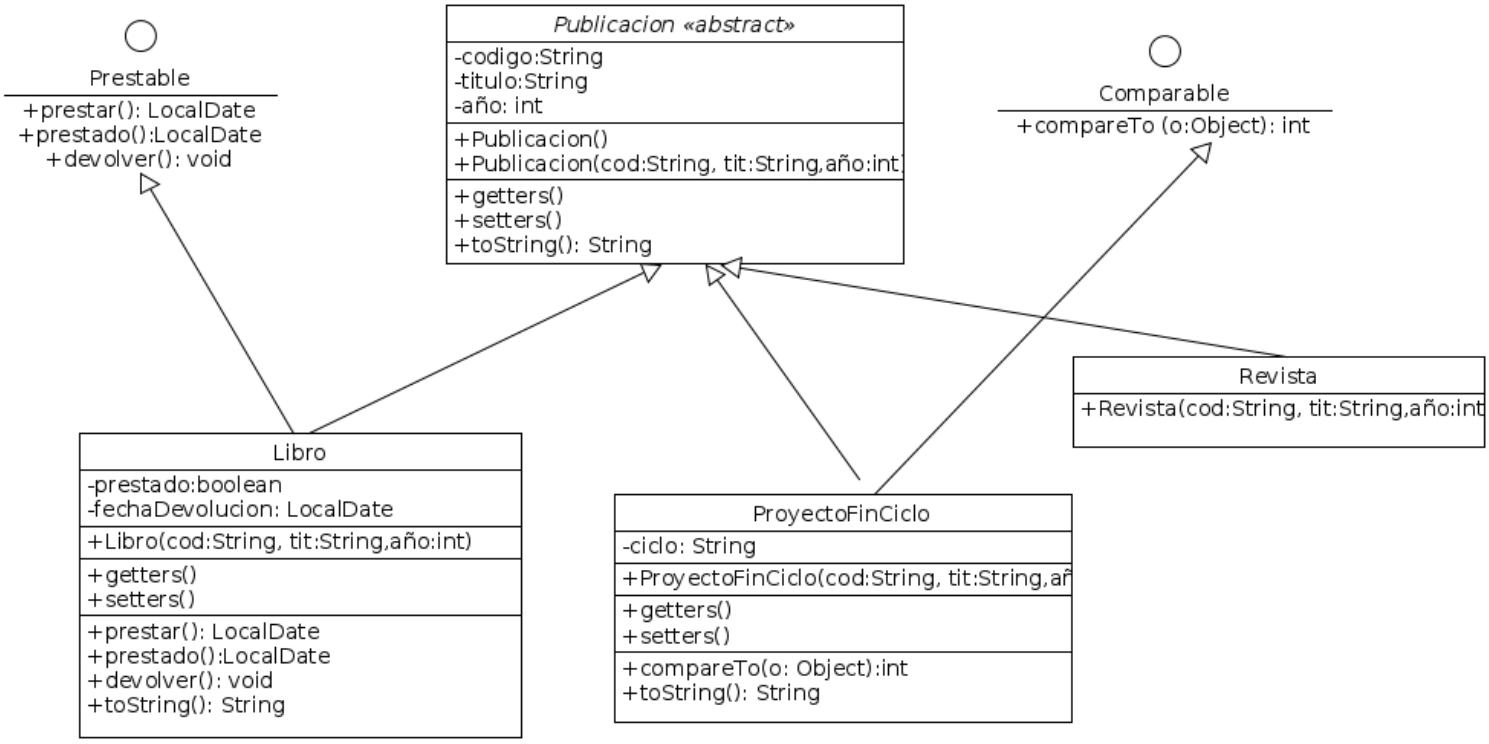
- prestar(): En caso de no estar prestado, lo marca como “prestado” y devuelve la fecha de devolución del libro (teniendo en cuenta que los libros se prestan por 1 mes). Si estuviese ya prestado, devolverá null.
- devolver(): Efectúa la devolución del libro (lo desmarca como “prestado” y restablece la fecha de devolución a null)
- prestado(): En caso de estar prestado, devuelve la fecha prevista de devolución del libro; en caso, contrario, devuelve *null*.

**LocalDate** es una clase para trabajar con fechas en el huso horario local. Se encuentra en el paquete *java.time.*

Para este ejercicio, son de especial interés los siguientes métodos de LocalDate:

- *LocalDate.now()*  : estático, devuelve un objeto LocalDate con la fecha y hora del sistema
- *LocalDate.of (int año, int mes, int dia):* estático, devuelve un objeto LocalDate con la fecha cuyo año, mes y día se pasen como argumentos
- *plusMonths( long meses):*  a partir de un objeto LocalDate inicializado, devuelve una nueva fecha cuyo número de meses se incrementa en la cantidad pasada.
- *getYear(), getMontValue(), getDayOfMonth()* nos devolverían para un objeto LocalDate los enteros que representan el año, el número de mes y el número de días (respectivamente)

Diagrama de clases para este ejercicio y los siguientes:




**Ejercicio 4:**

La biblioteca ha decidido incorporar entre sus fondos los discos/CD de música Escribir una clase **DiscoPrestable** que herede de la clase *Disco*, desarrollada en la Colección 9, ejercicio 4, e implemente la interfaz *Prestable*.

DiscoPrestable deberá poseer los atributos :

- prestado , de tipo booleano
- fechaDevolucion, de tipo LocalDate

Para desarrollar los métodos definidos en Prestable, tener en cuenta que el tiempo de préstamo de los discos es de una semana. Podemos utilizar el método *plusWeeks (long cantidad)* de la clase LocalDate.

Nota: Hemos de traer o importar de la Colección 9 las clases Multimedia y Disco.

**Ejercicio 5:**

Escribir el código de una clase llamada **ProyectoFinCiclo** para gestionar los proyectos de fin de ciclo de alumnos de Grado Superior.

Esta clase hereda de **Publicacion** y reutiliza todos sus métodos y atributos. Agrega como atributo propio la denominación del ciclo al que corresponde el proyecto.

Además, debe implementar la interfaz **Comparable** a fin de que los proyectos puedan ordenarse. El criterio de ordenación será el siguiente:

1º) Por la  denominación del ciclo (alfabético), sin diferenciar entre mayúsculas y minúsculas.
2º) Para un  mismo ciclo, por su código(numérico) .

Por último, los proyectos sobreescribirán el método toString() para mostrar su código, título y ciclo al que corresponden.

**Ejercicio 6:**

Diseñar la clase **Futbolista** con los siguientes atributos:

- dni (cadena, que conste de 8 dígitos y una letra mayúscula)
- nombre (cadena, entre 5 y 100 caracteres)
- edad (entero, entre 16 y 99 años )
- goles (entero, no negativo)

Los valores de los atributos se validarán usando expresiones regulares.

Desarrollar en la clase Futbolista.

- Un constructor que inicialice todos los atributos.
- El método toString().
- El método equals(), basado en el atributo “dni”
- La implementación de la interfaz Coparable, con la ordenación basada igualmente en el atributo “dni”
- Dos métodos de comparación diferentes, que implementen la interfaz Comparator:
  - Un primer comparador basado en el nombre (sin diferenciar mayúsculas de minúsculas)
  - Un segundo método de comparación basado en la edad

Probar la clase creando una lista con al menos 5 futbolistas, y devolverlos ordenados por dni, nombre y edad.

**Ejercicio 7:**

Desarrollar una aplicación para gestionar la información de los Empleados y Clientes de un banco, teniendo en cuenta que una misma persona puede ser a la vez empleado y cliente. Para ello:

1.Se desarrollará la interfaz **Cliente** que tendrá:

- Un atributo con el saldo de la cuenta (decimal), inicialmente a 0.
- Los métodos para recuperar el saldo e incrementarlo en una cantidad dada.

2.Se desarrollará la interfaz **Empleado**, que tendrá:
- Un atributo con el número de horas trabajadas (entero), inicialmente a 0.
- Los métodos para consultar las horas trabajadas e incrementarlas en una cantidad dada.

3. SE desarrollará la clase **Persona** que implementará las dos interfaces anteriores y contará con las propiedades siguientes:
- dni (cadena de 9 dígitos y letra mayúscula. No se puede modificar una vez asignada en el constructor)
- nombre (cadena)
- dos atributos booleanos para indicar si es cliente y /o empleado, que tampoco pueden modificarse una vez asignados

La clase Persona contará con un constructor de la forma *Persona(String dni, String nombre, boolean esCliente, boolean esEmpleado)*. En este constructor:
- Se asignan las propiedades dni y nombre
- Se asignan las propiedades booleanas, con la condición de que al menos una de ellas debe ser verdadera

Igualmente, tendrá un método *toString()* el cual:
- Mostrará las propiedades dni y nombre.
- Indicará si es Empleado y/o Cliente. Dependiendo de esta circunstancia, mostrará el saldo de su cuenta, las horas trabajadas, o ambas propiedades

Probar la clase creando una lista de la clase Persona que contenga al menos:
- Un objeto que sea empleado, pero no cliente
- Un objeto que sea cliente, pero no empleado
- Un objeto que sea a la vez cliente y empleado, para el que aumentaremos tanto su saldo como sus horas trabajada.

**Ejercicio 8:**

Las cartas de la baraja española están formadas por un palo (Sota, Caballo, Rey o Bastos) y un número ( As, Dos, Tres, Cuatro, Cinco, Seis, Siete, Ocho, Nueve, Sota, Caballo y  Rey, que representan respectivamente los números 1 al 12 de su palo).
Crear la clase **Carta**, que tendrá 2 propiedades definidas sobre tipos enumerados:
- palo
- número

Desarrollar dos métodos para la ordenación de cartas:
- El primer método las ordena por el número.

- El segundo método las ordena por el palo y, para el mismo palo, por el número
- 
- Desarrollar por último un método estático en la clase Carta que construya y devuelva una carta donde el palo y número hayan sido elegidos al azar.