Proyecto de laboratorio - Java

Estructura general del proyecto

a) El proyecto debe contener tres carpetas: src, data y doc.
b) La carpeta src debe incluir el código fuente, organizado en paquetes, la carpeta data el fichero de trabajo que más adelante se le proporcionará y, por último, la carpeta doc donde se guardarán la documentación del proyecto.
c) El proyecto debe contener un fichero README.md donde se describirán los datos y los tipos implementados.

Contenido

Entrega 1

1) Diseño moderno[1]

Cree un paquete de nombre fp.clinico y programe los siguientes tipos como records.

Persona

- Propiedades:
  - nombre, de tipo String.
  - apellidos, de tipo String.
  - dni, de tipo String.
  - fecha de nacimiento, de tipo LocalDate.
  - edad, de tipo Integer. (Derivada a partir de la fecha de nacimiento).
- Restricciones:
  - La fecha de nacimiento debe ser anterior a la fecha actual.
  - El dni debe ser una cadena con ocho dígitos y seguidos de una letra.
- Representación como cadena: por defecto asociado al record.
- Criterio de igualdad: por defecto asociado al record.
- Orden natural: por dni.
- Comentarios: añada los siguientes métodos de factoría:
  - Método static of:
    - recibe nombre, apellidos, dni y fecha de nacimiento y devuelve una persona.
  - Método static parse:
    - Recibe una cadena con un formato específico y devuelve una persona. Ejemplo de cadena: "Juan, García Rodríguez, 12755078Z, 20/03/1965".
  - Incluya un método main para comprobar el correcto funcionamiento del método parse. *public static void main(String[] args){ … }*

Paciente

- Propiedades:
  - persona, de tipo Persona.
  - código de ingreso, de tipo String.
  - fecha y hora de ingreso, de tipo LocalDateTime.
  - fecha de ingreso, de tipo LocalDate. (Derivada a partir de la fecha y hora de ingreso)
  - hora de ingreso, de tipo String. (Derivada a partir de la fecha y hora de ingreso). Ejemplo de cadena: "15:03". Es decir, la hora y los minutos tienen que tener dos dígitos. No valdría escribir "15:3".
- Restricciones:
  - La fecha y hora de ingreso debe ser anterior o igual a la fecha actual.
- Representación como cadena: por defecto asociado al record.
- Criterio de igualdad: por defecto asociado al record.
- Comentarios: añada los siguientes métodos de factoría:
  - Método static of:

[1] Siguiendo las pautas de diseño vistas en clase de teoría con Miguel Toro.

        - recibe nombre, apellidos, dni, fecha de nacimiento, código y fecha y hora de ingreso y devuelve un paciente.
    - Método static of:
        - recibe un objeto persona, un código y una fecha y hora de ingreso y devuelve un paciente.

PacienteEstudio

- Propiedades:
    - id, de tipo String.
    - genero, de tipo String.
    - edad, de tipo Double.
    - hipertensión, de tipo Boolean.
    - enfermedad del corazón, de tipo Boolean.
    - tipo de residencia, enumerado TipoResidencia, cuyos valores son rural o urbana.
    - nivel medio de glucosa, de tipo Double.
    - factor de riesgo, de tipo Boolean. (Derivada, si tiene hipertensión y más de 40 años se considerará que tiene factor de riesgo).
- Restricciones:
    - La edad tiene que ser mayor o igual que cero y menor o igual que 130.
    - El nivel medio de glucosa tiene que ser mayor o igual que cero.
- Representación como cadena: informa del id y la edad del paciente.
- Criterio de igualdad: por defecto asociado al record.
- Criterio de orden: según la edad y el id.
- Comentarios: añada los siguientes métodos de factoría:
    - Método static of:
        - recibe valores para cada propiedad básica y devuelve un objeto del tipo.
    - Método static parse:
        - recibe una cadena con un formato especificado y y devuelve un objeto del tipo. Ejemplo de cadena: “6306;Male;80;false;false;URBANA;83.84”
    - Incluya un método main para comprobar el correcto funcionamiento del método parse. *public static void main(String[] args){ … }*

Cree un paquete de nombre fp.vacunas y programe el siguiente tipo como un record.

Vacunacion

- Propiedades:
    - fecha, de tipo LocalDate.
    - comunidad, de tipo String.
    - pfizer, de tipo Integer.
    - moderna, de tipo Integer.
    - astrazeneca, de tipo Integer.
    - janssen de tipo Integer.
    - número de personas, de tipo Integer.

- número total, de tipo Integer. (Derivada, siendo la suma de dosis de Pfizer, moderna, astrazeneca y janssen).

- Restricciones:
  - La fecha de debe ser posterior al 01/02/2021.
- Representación como cadena: por defecto asociado al record.
- Criterio de igualdad: por defecto asociado al record.
- Orden natural: por comunidad y en caso de igualdad por fecha.
- Comentarios: añada los siguientes métodos de factoría:
  - Método static of:
    - recibe valores para cada propiedad básica y devuelve un objeto del tipo.
  - Método static parse:
    - recibe una cadena con un formato específico y devuelve un objeto del tipo. Ejemplo de cadena: "04/01/2021;Andalucía;140295;0;0;0;0".
  - Incluya un método main para comprobar el correcto funcionamiento del método parse. *public static void main(String[] args){ … }*

2) Diseño clásico[2]

Cree un paquete de nombre fp.farmaceutico y programe el siguiente tipo como una clase.

Medicamento

- Propiedades:
  - nombre del medicamento, de tipo String, observable.
  - tipo de medicamento, enumerado de tipo TipoMedicamento, observable. Los valores del enumerado son anatómico, químico y terapéutico.
  - código de la enfermedad, de tipo String, observable.
  - farmacéutica, de tipo String, observable.
  - puntación, de tipo Double, observable.
  - índice somático, de tipo Integer, observable.
  - fecha de catálogo, de tipo LocalDate, **observable y modificable**.
  - tratar enfermedad, de tipo Boolean. (Derivada, siendo cierta si el código de la enfermedad coincide con un parámetro de tipo cadena que reciben como argumento la propiedad).
- Restricciones:
  - La puntación tiene que ser mayor estricta que cero.
  - El índice somático tiene que ser mayor o igual que 1000.
  - La fecha de catálogo tiene que ser posterior al 01/01/2015.
- Representación como cadena: según el nombre del medicamento y de la farmacéutica.
- Criterio de igualdad: por nombre del medicamento y farmacéutica.
- Orden natural: por nombre del medicamento y en caso de igualdad por la farmacéutica.
- Comentarios:

[2] Los records fueron introducidos en la versión 14 de Java – el JDK14 – que salió a la luz el 17 de marzo de 2020. Llamaremos *diseño clásico* al diseño que se solía hacer hasta ese momento. En general, no se insistía en el uso de tipos inmutables para los tipos simples – que teniendo records se programan con mucha comodidad – ni, por otro lado, se utilizaban una serie de pautas o patrones software a la hora de programar los tipos.

- Programe una **clase de nombre FactoriaMedicamentos** que incluya, de momento, un método static de nombre **parseaMedicamente**, que recibe una cadena con un formato específico y devuelve un objeto de tipo Medicamento.
  Ejemplo:
  "efavirenz,Anatomico,Y212XXA,Actavis Mid Atlantic LLC,90.0,1848,04/12/2019". (Siendo la información del nombre del medicamento, el tipo de medicamento, el codigo de la enfermedad, la farmaceutica, la puntuacion, el índice somatico, y la fecha de catalogo)
- Implemente una clase de nombre TestFactoriaMedicamentos en un paquete de nombre fp.farmaceutico.test y compruebe el correcto funcionamiento del método anterior.

Entrega 2

Se le suministran los siguientes ficheros que deberá guardar en la carpeta data:

- *estudio_clinico.csv*: contiene la información sobre objetos del tipo PacienteEstudio.
- *medicamentos.csv*: contiene la información sobre objetos del tipo Medicamento.
- *ccaa_vacunas_3.csv*: contiene la información sobre objetos del tipo Vacunacion.

Así mismo, **se le proporciona la siguiente interfaz** que debe incorporar en el paquete fp.clinico. Para ello, cree una nueva interfaz con el nombre indicado − **EstudioClinico** − y copie los métodos que aparecen en el enunciado. Esta interfaz va a tener dos implementaciones: una de ellas de tipo imperativa, que llamaremos **EstudioClinicoBucles**, y otra de tipo funcional, que llamaremos **EstudioClinicoStream**.

```
public interface EstudioClinico {
	// Propiedades de lista
	Integer numeroPacientes();
	void incluyePaciente(PacienteEstudio paciente);
	void incluyePacientes(Collection<PacienteEstudio> pacientes);
	void eliminaPaciente(PacienteEstudio paciente);
	Boolean estaPaciente(PacienteEstudio paciente);
	void borraEstudio();
	//
	// Método de factoría
	EstudioClinico of(String nombreFichero);
	List<PacienteEstudio> leeFichero(String nombreFichero);
	// Tratamientos secuenciales: implementaciónn funcional vs. imperativa
	//existe, paraTodo
	Boolean todosPacienteSonDelTipo(TipoResidencia tipo);
	Boolean existeAlgunPacienteDelTipo(TipoResidencia tipo);
	//contador, suma, media
	Integer numeroPacientesFactorRiesgo();
	Double edadMediaPacientesConFactorRiesgo();
	//filtrado
	List<PacienteEstudio> filtraPacientesPorEdad(Double edad);
	//devuelve Map que agrupa
	Map<String,List<PacienteEstudio>> agruparPacientesEdadMayorQuePorGenero(Double edad);
	//devuelve Map que realiza un cálculo
	Map<String,Long> numeroPacientesPorGenero();
	Map<String,Double> edadMediaPacientesPorPorGenero();
}
```

Detalles: una vez incorporada la interfaz en el paquete fp.clinico, construya las EstudioClinicoBucles y EstudioClinicoStream.

Al hacerlo, new/class/ se pone el nombre de la clase y, en el recuadro de "Interfaces" se añade el nombre de la interfaz mediante el botón Add. De esta manera se crea la estructura de la clase con todos los métodos de la interfaz con TODOs. Al ir programando se van borrando poco a poco las marcas azules de los TODOs.

- Al realizar la clase TestEstudioClinicoBucles, o la clase TestEstudioClinicoStream – que creará en la Entrega 3 –, cuando necesite crear un objeto. Bastará utilizar el constructor de la clase de la siguiente manera:

  ```
  EstudioClinico est = new EstudioClinicoBucles( --- );
  ```
- Observe que el tipo debe tener un atributo que sea un objeto del tipo List<PacienteEstudio>
- Los métodos son todos propiedades derivadas u operaciones.

- El método leeFichero es un método auxiliar que carga el fichero correspondiente en una lista de objetos del tipo. El método of realiza una operación parecida (de hecho, al programar hará una llamada al método leeFichero), pero construye un objeto del tipo EstudioClinico.
- Los nombres de los métodos son lo suficientemente explicativos para poder comprender qué debe hacer cada uno.
- Observe que los métodos of y leeFichero son métodos normales, no son static. Por lo tanto, para poder llamarlos tiene que construir antes un objeto EstudioClinico. Por este motivo, se debe hacer un constructor sin parámetros que construya un objeto sin contenido y cuya misión es poder utilizar estos métodos.

Se debe programar:

1) Factorías

Hay dos maneras de hacer las factorías. En ambos casos la programación es similar, aunque el diseño – es decir, la manera de estructurar el código – son diferentes.

a) **Como un método más dentro de un tipo que contiene otros tratamientos secuenciales.** Este diseño es más adecuado para proyectos grandes que serán mantenido a lo largo del tiempo. Por ejemplo, es la manera de trabajar en la API de Java.

Debe programar:

- Método **of** y método **leeFichero** de la clase **EstudioClinicoBucles** que implementa a la interfaz EstudioClinico.
  Para ello tiene que haber creado previamente dicha clase y haberle incorporado una lista de objetos del tipo PacienteEstudio, así como los constructores:
  - Constructor vacío, que construya la lista sin ningún elemento.
  - Constructor que reciba una lista de objetos del tipo PacienteEstudio como parámetro.
  - Nota: deberá utilizar el método parse que, dada una línea del fichero, construya un objeto del tipo PacienteEstudio y que programó en la entrega 1.

- Realice una comprobación de que la programación funciona bien en una clase de nombre TestEstudioClinicoBucles que testee la lectura del fichero.

b) **Como una clase independiente cuya una misión es construir listas de objetos.**
Este diseño es más adecuado para proyectos pequeños. Por ejemplo, para un examen que solo dura unas horas.

Debe programar:

- Una clase de nombre **FactoriaVacunaciones** que contenga un método de nombre leeFichero que, dada una cadena con el nombre del fichero, devuelva una lista de objetos Vacunacion. Deberá programar un método auxiliar que parsee cada línea del fichero.
  - Realice una comprobación en una clase de testeo que lo programado funciona correctamente.
- Una clase de nombre **FactoriaMedicamentos** que contenga un método de nombre leeFichero que, dada una cadena con el nombre del fichero, devuelva una

lista de objetos Medicamento. Deberá programar un método auxiliar que parsee cada línea del fichero.

- Realice una comprobación en una clase de testeo que lo programado funciona correctamente.

2) Métodos de la clase EstudioClinicoBucles

Programe **todos los métodos de la clase EstudioClincoBucles**, teniendo en cuenta que son métodos:

- Métodos con propiedades de listas.
- Métodos de tratamientos secuenciales:
  - Operaciones de existencia, si todos los objetos verifican una propiedad, contadores, cálculos de una media.
  - Operación de filtrado en una lista de Pacientes
- Métodos para la construcción de diccionarios o mapas:
  - Construcción de diccionarios que agrupan.
  - Construcción de diccionarios que sirven para hacer un cálculo múltiplo. (Por ejemplo. contador múltiple).

Programe tan solo todos los métodos de las propiedades de lista de la clase **EstudioClincoStream** y deje el resto con los TODOs, que haremos en la siguiente entrega.

Entrega 3

En esta entrega se programará todo lo relativo a la **programación en *streaming*.**

1) Implementación funcional del tipo EstudioClinico: métodos de la clase EstudioClinicoStream

Programe **todos los métodos de la clase EstudioClinicoStream**, teniendo en cuenta que los métodos iniciales son exactamente iguales que en la clase EstudioClinicoBucles – métodos de propiedades de lista y métodos de factoría - y que los métodos de tratamientos secuenciales deben programarse utilizando operacions del tipo Stream<T>

- Nota: se recomienda utilizar la opción Collectors.groupingBy(- , -); para los métodos que construyan diccionarios.

Recordatorio de la interfaz

```
public interface EstudioClinico {
	// Propiedades de lista
	Integer numeroPacientes();
	void incluyePaciente(PacienteEstudio paciente);
	void incluyePacientes(Collection<PacienteEstudio> pacientes);
	void eliminaPaciente(PacienteEstudio paciente);
	Boolean estaPaciente(PacienteEstudio paciente);
	void borraEstudio();
	//
	// Método de factoría
	EstudioClinico of(String nombreFichero);
	List<PacienteEstudio> leeFichero(String nombreFichero);
	// Tratamientos secuenciales: implementaciónn funcional vs. imperativa
	//existe, paraTodo
	Boolean todosPacienteSonDelTipo(TipoResidencia tipo);
	Boolean existeAlgunPacienteDelTipo(TipoResidencia tipo);
	//contador, suma, media
	Integer numeroPacientesFactorRiesgo();
	Double edadMediaPacientesConFactorRiesgo();
	//filtrado
	List<PacienteEstudio> filtraPacientesPorEdad(Double edad);
	//devuelve Map que agrupa
	Map<String,List<PacienteEstudio>> agruparPacientesEdadMayorQuePorGenero(Double edad);
	//devuelve Map que realiza un cálculo
	Map<String,Long> numeroPacientesPorGenero();
	Map<String,Double> edadMediaPacientesPorPorGenero();
}
```

2) Diseño clásico o para proyectos pequeños

a) Implemente una clase que se llame Vacunaciones en el paquete fp.vacunas.

Incluya un único constructor que reciba un Stream<Vacunacion> e inicialice el atributo (que es un List<Vacunacion>) con los objetos de dicho Stream. No es necesario implementar representación como cadena ni criterio de igualdad.
Implemente los siguientes métodos:

- ***anyadeVacunacion***: dado un objeto del tipo Vacunacion lo añade al atributo de List<Vacunacion>.

- ***vacunacionesEntreFechas*:** dadas dos fechas como parámetros de entrada, devuelve una lista con aquellas vacunaciones entre dichas fechas.
- ***existeNumPersonasPautaCompletaPorEncimaDe*:** dada una comunidad y un valor entero, indica si existen o no vacunaciones con un número de personas con la pauta completa de vacunación por encima del valor entero dado.
- ***diaMasVacunacionesEn*:** dada una comunidad, devuelve la fecha en la que hubo más personas vacunadas.
- ***vacunacionesPorFecha*:** devuelve un mapa, o diccionario, en el que las claves son las fechas y los valores son listas de vacunaciones asociadas a dichas fechas.
- ***maximoNumTotalVacunasporComunidad*:** devuelve un mapa, o diccionario, en el que las claves son las comunidades y los valores son el máximo para el número total de vacunas puestas para cada comunidad.

b) Implemente una clase que se llame ListadoMedicamentos en el paquete fp.farmaceutico.

Implemente la clase ListadoMedicamentos. Incluya un único constructor que reciba un Stream<Medicamento> e inicialice el atributo con los objetos de dicho Stream. No es necesario implementar representación como cadena ni criterio de igualdad. Implemente en la clase los siguientes métodos:

- **existeMedicamentoSegunTipoAnteriorA**: dado un tipo de medicamento y una fecha, indica si existe un medicamento de dicho tipo posterior a la fecha dada.
- **nombreMedicamentosPuntuacionMayorA**: dada una puntuación, devuelve un conjunto con los nombres de los medicamentos con una puntuación mayor a la dada.
- **nombreMedicamentoMayorIndiceSomaticoSegunTipoMedicamento**: dado un tipo de medicamento, devuelve el nombre del medicamento con mayor índice somático. En caso de no haber ninguno, se eleva una excepción.
- **agrupaTipoMedicamentoSegunPuntuacionMedia**: devuelve un diccionario que asocia a cada tipo de medicamento su puntuación media.
- **fechaCatalogoMasFrecuente**: devuelve la fecha del catálogo más frecuente, es decir, la que aparece más veces.

Métodos adicionales

Realizamos la ampliación del tipo EstudioClinico con vistas a tener más métodos que trabajen con diccionarios a mapa. Para ello, en lugar de ampliar la interfaz con más métodos – lo que implicaría tener que modificar no solo la clase EstudioClinicoStream, sino también la clase EstudioClinicoBucles –, lo que hacemos es utilizar herencia para definir un subtipo.

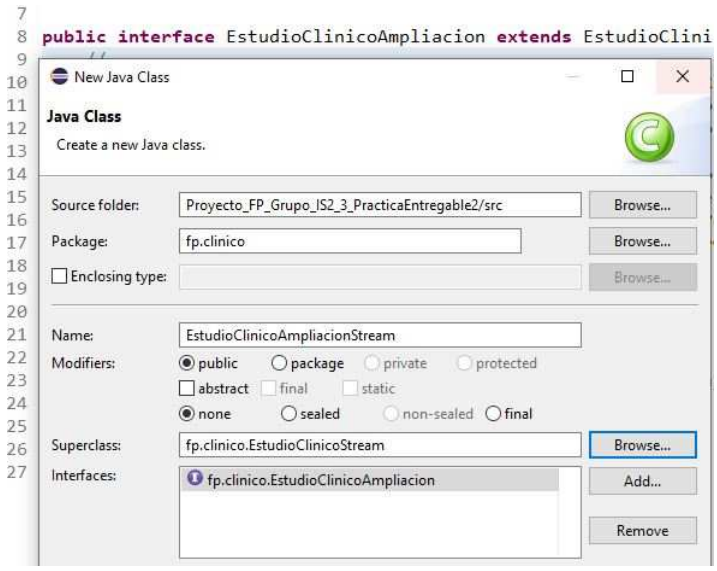
- La herencia no constituye una parte central de la evaluación de la asignatura, por lo que se proporciona el código inicial para poder programar.
- Una vez copiado y pegado el código que se proporciona, el único cambio es que al llamar al atributo se pondrá **super.pacientes** en lugar de **this.pacientes**
- Cambio en la clase padre del atributo de ***private*** a ***protected***.

Interfaz **EstudioClinicoAmpliacion**

(el enunciado detallado de cada método se proporciona al final de este apartado)

```
public interface EstudioClinicoAmpliacion extends EstudioClinico{
	//
	Map<TipoResidencia,Integer> agruparNumeroPacientesPorTipoResidencia();
	Map<TipoResidencia,Double> agruparNivelMedioGlucosaMedioPorTipoResidencia();
	Map<TipoResidencia,PacienteEstudio>
agruparNivelMedioGlucosaMaximoPorTipoResidencia();
	Map<String,List<PacienteEstudio>> agrupaPacientesPorGenero();
	Map<String,Set<PacienteEstudio>> agrupaPacientesPorPorGeneroEnConjunto();
	Map<String,SortedSet<PacienteEstudio>>
agrupaPacientesPorPorGeneroEnConjuntoOrdenado();
	Map<String,PacienteEstudio> pacienteEdadMaximaPacientesPorGenero();
	Map<String,List<Double>> listaEdadesPorGenero();
	Map<String,Double> edadMaximaPacientesPorGenero();
	String generoEdadMaximaPacientesPorGenero();
}
```

Al construir la clase, puede hacerlo de la siguiente manera:



En cualquier caso, el código de la clase **EstudioClinicoAmpliacionStream** con el primer método sería de la siguiente manera:

```
public class EstudioClinicoAmpliacionStream extends EstudioClinicoStream implements EstudioClinicoAmpliacion {
    // Atributos
    //protected List<PacienteEstudio> pacientes; //CAMBIO EN LA CLASE PADRE
    //Constructores
    public EstudioClinicoAmpliacionStream(){
        super();
    }
    public EstudioClinicoAmpliacionStream(List<PacienteEstudio> lista){
        super(lista);
    }
    public EstudioClinicoAmpliacionStream(Stream<PacienteEstudio> st){
        super(st);
    }
    //Métodos
    @Override
    public Map<TipoResidencia, Integer> agruparNumeroPacientesPorTipoResidencia() {
        //
        return super.pacientes.stream().
                collect(Collectors.groupingBy(
                        PacienteEstudio::tipoResidencia,
                        Collectors.collectingAndThen(Collectors.counting(), Long::intValue)
                        ));
    }
    @Override
    public Map<TipoResidencia, Double> agruparNivelMedioGlucosaMedioPorTipoResidencia() {
        // TODO Auto-generated method stub
```

Nota: en la clase EstudioClinicoStream debe cambiar el atributo pacientes de **private** a **protected**. De esta forma es visible en la clase hija – la clase EstudioClinicoAmpliacionStream – y puede llamarlo a través de la cláusula super, como podemos ver en el código de arriba.

Enunciado detallado de los métodos:

`Map<TipoResidencia,Integer> agruparNumeroPacientesPorTipoResidencia();`

- Un método que devuelva un diccionario cuyas claves son el tipo de residencia y cuyos valores asociados son el número de pacientes de cada tipo. Tenga en cuenta que los valores son de tipo Integer.

`Map<TipoResidencia,Double> agruparNivelMedioGlucosaMedioPorTipoResidencia();`

- Un método que devuelva un diccionario cuyas claves son el tipo de residencia y cuyo valor asociado es la media del valor medio de glucosa de los pacientes.

`Map<TipoResidencia,PacienteEstudio> agruparNivelMedioGlucosaMaximoPorTipoResidencia();`

- Un método que devuelva un diccionario cuyas claves son el tipo de residencia y cuyo valor asociado es el paciente con mayor nivel medio de glucosa. Tenga en cuenta que los valores son de tipo PacienteEstudio y no del tipo Optional<PacienteEstudio>.

`Map<String,List<PacienteEstudio>> agrupaPacientesPorGenero();`

- Un método que devuelva un diccionario cuyas claves son el género del paciente y cuyos valores son las listas de pacientes asociados a cada género.

```
Map<String,Set<PacienteEstudio>> agrupaPacientesPorPorGeneroEnConjunto();
```

- Un método que devuelva un diccionario cuyas claves son el género del paciente y cuyos valores son los conjuntos de pacientes asociados a cada género.

```
Map<String,SortedSet<PacienteEstudio>> agrupaPacientesPorPorGeneroEnConjuntoOrdenado();
```

- Un método que devuelva un diccionario cuyas claves son el género del paciente y cuyos valores son los conjuntos de pacientes asociados a cada género.

```
Map<String,PacienteEstudio> pacienteEdadMaximaPacientesPorGenero();
```

- Un método que devuelva un diccionario cuyas claves son el género del paciente y cuyo valor es el paciente con mayor edad. Tenga en cuenta que los valores son de tipo PacienteEstudio y no del tipo Optional<PacienteEstudio>.

```
Map<String,List<Double>> listaEdadesPorGenero();
```

- Un método que devuelva un diccionario cuyas claves son el género del paciente y cuyos valores son listas de edades de los pacientes de cada género.

```
Map<String,Double> edadMaximaPacientesPorGenero();
```

- Un método que devuelva un diccionario cuyas claves son el género del paciente y cuyos valores son listas de edades de los pacientes de cada género.

```
String generoEdadMaximaPacientesPorGenero();
```

- Un método que devuelva el género del paciente con más edad. Este método puede hacerlo basándose en el diccionario que devuelve el método edadMaximaPacientesPorGenero().

Comentarios sobre diseño

- **Métodos factoría en el tipo EstudioClinico**

  Tanto el método ***of*** del tipo EstudioClinico como el método ***leerFichero*** se han dejado como métodos asociados al tipo en lugar de como "métodos de clase". Es decir, son exactamente igual que el resto de métodos, por lo que al llamarlos se debe hacer como una propiedad ligada a un objeto. Como tenemos un constructor vacío en las clases que implementan al tipo se puede realizar sin ningún problema.

  Es decir, se puede utilizar este método de la siguiente manera:

```
//…
EstudioClinico aux = new EstudioClinicoBucles();
EstudioClinico estudio = aux.of("data/estudio_clinico.csv");
Integer num = estudio.numeroPacientes();
System.out.println("Número de pacientes: "+num);
```

  Sin embargo, estos métodos se podrían haber programado como métodos de clase o métodos estáticos. De esa manera, al llamarlos se podría hacer directamente utilizando el nombre de la interfaz:

```
EstudioClinico estudio = EstudioClinico.of("data/estudio_clinico.csv");
```

  No es lo que pedía el enunciado, pero ¿cómo se debería haber programado la interfaz para que así fuera?

```
public interface EstudioClinico {
    // Propiedades de lista
    Integer numeroPacientes();
    void incluyePaciente(PacienteEstudio paciente);
    void incluyePacientes(Collection<PacienteEstudio> pacientes);
    void eliminaPaciente(PacienteEstudio paciente);
    Boolean estaPaciente(PacienteEstudio paciente);
    void borraEstudio();
    //
    // Métodos de factoría
    public static EstudioClinico of(String nombreFichero) {
        //
        List<PacienteEstudio> lista = leeFichero(nombreFichero);
        EstudioClinico res = new EstudioClinicoStream(lista);
        return res;
    }
    private static List<PacienteEstudio> leeFichero(String nombreFichero){
        //
        List<PacienteEstudio> res = new ArrayList<>();
        try {
            res = Files.lines(Paths.get(nombreFichero)).
                    map(PacienteEstudio::parse).
                    collect(Collectors.toList());
        } catch (IOException e) {
            //
            e.printStackTrace();
        }
        return res;
    }
    //
    // Tratamientos secuenciales: implementación funcional vs. imperativa
    //existe, paraTodo
    Boolean todosPacienteSonDelTipo(TipoResidencia tipo);
    Boolean existeAlgunPacienteDelTipo(TipoResidencia tipo);
```

  Observe que en este ejemplo el método leeFichero está programado en streaming usando Files.lines()

- **Diseño general de la práctica**

  Durante toda la práctica hemos diseñado de dos maneras. De esta forma, hemos seguido las ideas vistas en clase de teoría – donde se ha tenido en mente el diseño de un proyecto grande que tiene que perdurar en el tiempo – y, por otro lado, las de prácticas, donde se conciben los proyectos como proyectos pequeños a los que solo le dedicaremos dos o tres horas de trabajo.

  a) Enunciado en letra normal: diseño de un proyecto grande que perdura en el tiempo. El código debe ser fácil de mantener, reutilizar y actualizar. Se siguen las ideas vistas en clase de teoría.

  b) Enunciado en gris: diseño de un proyecto pequeño al que tan solo se le dedican varias horas de trabajo y no se vuelve una vez finalizado. Se sigue la manera de programar vista en los enunciados de los boletines de prácticas.