# Diseño de clases.

1. (Paquete: **ejercicios.gestionempleados**) Una empresa quiere hacer una gestión informatizada básica de sus empleados. Para ello, de cada empleado le interesa:
   - Nombre (String)
   - DNI (String)
   - Año de ingreso (número entero)
   - Sueldo bruto anual (número real)

a) Diseñar una clase Java Empleado, que contenga los atributos (privados) que caracterizan a un empleado e implemente los métodos adecuados para:

- Consultar el valor de cada uno de sus atributos. (Consultores)
- Métodos modificadores de los atributos. El año de ingreso y el sueldo bruto no deben poder poder establecerse a un valor negativo. Se lanzará IllegalArgumentException para evitarlo si se intentase hacer.
- Crear objetos de la clase: Constructor que reciba todos los datos del empleado a crear. El constructor también debe tener en cuena que el año de ingreso y el sueldo bruto no pueden ser negativos (usar los correspondientes setter en el constructor)
- *public int antigüedad (). D*evuelve el número de años transcurridos desde el ingreso del empleado en la empresa. Si el año de ingreso fuera posterior al de la fecha actual, devolverá 0. Para obtener el año actual puedes usar:

```
int añoActual = Calendar.getInstance().get(Calendar.YEAR);
```

- *public void incrementarSueldo(double porcentaje).* Incrementa el sueldo del empleado en un porcentaje dado.
- *public String toString().* Devuelve un String con los datos del empleado, de la siguiente forma:

```
Nombre: Juan González
Dni: 545646556K
Año de ingreso: 1998
Sueldo bruto anual: 20000 €
```

- *public boolean equals(Object o).* Método para comprobar si dos empleados son iguales. Dos empleados se consideran iguales si tienen el mismo dni.
- Indicar que la clase empleado i*mplements Comparable<Empleado>* y añadir el método *public int compareTo(Empleado e).* Un empleado se considera menor que otro si su dni es menor, teniendo en cuenta la comparación de Strings.
- **Metodo estático** public static double calcularIRPF(double salarioMensual). Determina el % de IRPF que corresponde a un salario **mensual** determinado, según la siguiente tabla:

| Desde salario (incluido) | Hasta salario (no incluido) | % IRPF |
|---|---|---|
| | 800 | 3 |
| 800 | 1000 | 10 |
| 1000 | 1500 | 15 |
| 1500 | 2100 | 20 |
| 2100 | | 30 |

b) Diseñar una clase Java **TestEmpleado** que permita probar la clase Empleado y sus métodos. Para ello se desarrollará el método main en el que:

- Se crearán dos empleados utilizando los datos que introduzca el usuario,
- Se incrementará el sueldo un 20 % al empleado que menos cobre.
- Se incrementará el sueldo un 10% al empleando más antiguo.
- Muestra el irpf que correspondería a cada empleado. Ten en cuenta que el salario de un empleado es anual, mientras que calcularIRPF debe recibir un salario mensual.

- Para comprobar que las operaciones se realizan correctamente, muestra los datos de los empleados tras cada operación.

c) Diseñar una clase **Empresa**, que permita almacenar el nombre de la empresa y la información de los empleados de la misma. Para ello se definirán dos atributos privados:

- nombre, de tipo String (nombre de la empresa)
- plantilla, de tipo ArrayList<Empleado> (que contendrá la relación de empleados de la empresa)

En esta clase, se deben implementar los métodos:

- *public Empresa (String nombre)*. Constructor de la clase. Crea la empresa con el nombre indicado y sin empleados.
- public int getNumeroEmpleados(): Devuelve el número de empleados de la empresa.
- Método getter y setter del atributo nombre.
- Método getter de la plantilla
- *public void contratar (Empleado e) throws IllegalArgumentException.* Añade el empleado indicado a la plantilla de la empresa. Si el empleado ya estaba en la plantilla se lanza IllegalArgumentExeption con el mensaje "No se puede contratar a alguien que ya está contratado"
- *public void despedir (Empleado e) throws NoSuchElementException*. Elimina el empleado indicado de la plantilla. Si el empleado no existe en la empresa, se lanza la excepción *NoSuchElementException* con el mensaje "No se puede despedir a alguien que no está contratado"
- *public void despedir (String dni) throws NoSuchElementException*. Elimina de la plantilla el empleado cuyo dni se indica como parámetro. Si el empleado no existe en la empresa, se lanza la excepción *NoSuchElementException* con el mensaje "No se puede despedir a alguien que no está contratado"
- *public void subirTrienio (double porcentaje)* Subir el sueldo, en el porcentaje indicado, a todos los empleados que cumplen un trienio, es decir, cuya antigüedad sea múltiplo de 3.
- *public String toString().* Devuelve un String con el nombre de la empresa y la información de todos los empleados. La información de los distintos empleados debe estar separada por saltos de línea.

d) Diseñar una clase **TestEmpresa** que permita probar la clase Empresa y sus métodos. Para ello, desarrolla el método main y en él …:

- Crea una empresa, de nombre "MatisseDAM".
- Contrata a varios empleados (con el nombre, dni, etc. que quieras).
- Usa el método subirTrienio para subir un 10% el salario de los empleados que cumplen un trienio en el año actual.
- Despide a alguno de los empleados.
- Trata de despedir a algún empleado que no exista en la empresa.
- Muestra los datos de la empresa siempre que sea necesario para comprobar que las operaciones se realizan de forma correcta.

2. (paquete: **ejercicios.reservaslibreria**) Una librería quiere proporcionar a sus clientes el siguiente servicio: Cuando un cliente pide un libro y la librería no lo tiene, el cliente puede hacer una reserva de manera que cuando lo reciban en la librería le avisen por teléfono.

   De cada reserva se almacena:

   - Nif del cliente (String)
   - Nombre del cliente (String)
   - Teléfono del cliente (String)
   - Código del libro reservado. (Entero)
   - Número de ejemplares (entero)

a) Diseñar la clase **Reserva**, de manera que contemple la información descrita e implementar:

   - public Reserva(String nif,String nombre, Strint tel, int codigo, int ejemplares) Constructor que recibe todos los datos de la reserva.
   - *public Reserva(String nif,String nombre, Strint tel, int codigo)* Constructor que recibe los datos del cliente y el código del libro. Establece el número de ejemplares a uno.
   - Consultores de todos los atributos.
   - *public void setEjemplares(int ejemplares)* Modificador del número de ejemplares. Establece el número de ejemplares al valor indicado como parámetro.
   - *public String toString()* que devuelva un String con los datos de la reserva
   - *public boolean equals(Object o)*. Dos reservas son iguales si son del mismo cliente y reservan el mismo libro.
   - Indicar que Reserva implements Comparable <Reserva> y añadir el método *public int compareTo(Reserva r)*. Es menor la reserva cuyo código de libro es menor. A igual libro es menor aquella cuyo nif de cliente es menor (criterio alfabético).

b) Diseñar una clase Java **TestReservas** que permita probar la clase Reserva y sus métodos. Para ello se desarrollará el método main en el que:

   - Se creen dos reservas con los datos que introduce el usuario. Las reservas no pueden ser iguales (equals). Si la segunda reserva es igual a la primera se pedirá de nuevo los datos de la segunda al usuario.
   - Se incremente en uno el número de ejemplares de ambas reservas.
   - Se muestren las dos reservas por pantalla en orden, según el criterio del método compareTo.

c) Diseñar una clase **ListaReservas** que implemente una lista de reservas. Se utilizará un ArrayList de Reservas. Implementar los siguientes métodos:

   - Constructor, que cree la lista de Reservas vacía, sin reservas.
   - public void reservar(String nif, String nombre, String telefono, int libro, int ejemplares) throws IllegalArgumentException: Crea una reserva y la añade a la lista. Lanza IllegalArgumentException si la reserva ya estaba en la lista.
   - *public void cancelar(String nif, int libro) throws NoSuchElementException.* Dado un nombre de cliente y un código de libro, cancela (elimina) la reserva correspondiente. Lanzar *NoSuchElementException* si la reserva no existe.
   - *public String toString():* Devuelve un String con los datos de todas las reservas de la lista.
   - *public int numEjemplaresReservadosLibro(int codigo):* Devuelve el número de ejemplares que hay reservados en total de un libro determinado.
   - *public void reservasLibro(int codigo):* Dado un código de libro, muestra el nombre y el teléfono de todos los clientes que han reservado el libro indicado.

d) Realizar un programa **GestionReservas** que, utilizando un menú, permita:

   - Realizar reserva. Permite al usuario realizar una reserva.
   - Anular reserva: Se anula la reserva que indique el usuario (Nif de cliente y código de libro).

- Pedido: El usuario introduce un código de libro y el programa muestra el nº de reservas que se han hecho del libro. Esta opción de menú le resultará útil al usuario para poder hacer el pedido de un libro determinado.
- Recepción: Cuando el usuario recibe un libro quiere llamar por teléfono a los clientes que lo reservaron. Solicitar al usuario un código de libro y mostrar los datos (nombre y teléfono) de los clientes que lo tienen reservado.

3. (paquete **ejercicios.coches**)

a) Escribe una clase **Coche** que simula el comportamiento con la velocidad y revoluciones de un coche.

Los atributos serán:

- Si está o no en marcha (enMarcha)
- La velocidad máxima. (velocidadMaxima)
- La velocidad actual. (velocidadActual)
- Las revoluciones máximas.(rpmMaxima)
- Las revoluciones actuales (rpmActual)
- La marcha

Implementar los métodos getter.

Implementar los métodos setter, teniendo en cuenta que:

- La velocidad actual no deberá sobrepasar la máxima ni bajar de cero. Si se intenta bajar de cero se pondrá a cero. Si se intenta subir del máximo se pondrá al máximo.
- Las rpm actual no deberá sobrepasar la máxima ni bajar de cero. Si se intenta bajar de cero se pondrá a cero. Si se intenta subir del máximo se pondrá al máximo.
- La marcha se deberá mantener entre 0 y 5. Si se intenta bajar de cero se pondrá a cero. Si se intenta subir de 5 se pondrá a 5..

Constructores: Implementar un constructor sin parámetros que ponga la velocidad máxima a 180 y las rpm máximas a 65000. Implementar un segundo constructor que reciba la velocidad máxima y rpm máxima. El resto de valores se inicializarán a cero.

Implementa además los siguientes métodos:

- public String toString(): Que devuelva un String que refleje el estado del coche.
- public void arrancar(): Pone el coche en marcha (si no lo estaba ya).
- public void apagar(): Si el coche está en marcha, se apaga. La velocidad actual, rmp actual y marcha pasará a ser 0.
- public void acelerar (double v): incrementa en v la velocidad actual, y en v x 70 las rpm actuales. Solo tendrá efecto si el coche está en marcha.
- public void frenar (double v): decrementa en v la velocidad actual, y en v x 70 las rpm actuales. Si la velocidad llega a cero, se bajarán a cero también las rpm y la marcha. Solo tendrá efecto si el coche está en marcha.
- public void subirMarcha (): incrementa la marcha en 1 (maximo 5), y decrementa las rpm actuales un 30%. Solo tendrá efecto si el coche está en marcha.
- public void bajarMarcha (): decrementa la marcha en 1 (mínimo 0), e incrementa las rpm actuales un 30%. Solo tendrá efecto si el coche está en marcha.
- public double consumo (): devuelve los litros de combustible consumidos a los 100 km., calculados como las rpm actuales dividido entre la marcha, todo ello dividido por 1000. Si el coche está en la marcha 0 el consumo será 0.
- public double tiempoParaLlegar (double km): devuelve cuantas horas son necesarias para recorrer la distancia *km* a la velocidad actual.

b) Escribe un programa TestCoche que cree algún objeto Coche y probar los métodos

4. (paquete **gestorDeVuelos**) Se desea realizar una aplicación GestorVuelos para gestionar la reserva y cancelación de vuelos en una agencia de viajes. Dicha agencia trabaja únicamente con 1a compañía aérea Iberia, que ofrece vuelos desde/hacia varias ciudades de Europa. Se deben definir las clases que siguen, teniendo en cuenta que sus atributos serán privados y sus métodos sólo los que se indican en cada clase.

a) Implementación de la clase Pasajero:

- Atributos: dni y nombre.
- Métodos. getter, setter, constructor, toString (los dos datos en la misma línea, separados por un guión) y equals (son iguales si tienen el mismo dni).

b) Implementación de la clase **Vuelo**, que permite representar un vuelo mediante los atributos:

- identificador (String),
- origen (String),
- destino (String),
- horaSalida (Tiempo),
- horaLlegada (Tiempo).
- capacidad (int), que es el número de pasajeros que caben en el avión,
- numReservas (int), es el número de asientos que están reservados.
- Pasajero[] pasaje: Un array de pasajeros con la información de los pasajeros que viajarán en el vuelo. El tamaño del array será la capacidad + 1, ya que el asiento 0 (la posición 0 del array no se usará). Cada elemento del array (pasaje[i]) contendrá un objeto Pasajero si el asiento i está ocupado y contendrá el valor null si el asiento está libre. En el array pasaje, las posiciones impares corresponden a asientos de ventanilla y las posiciones pares, a asientos de pasillo

En esta clase, se deben implementar los siguientes métodos:

- public Vuelo (String id, String orig, String dest, Tiempo hsal, Tiempo hlleg, int capacidad): **Constructor** que crea un vuelo con la información indicada en los parámetros. El número de reservas será 0 y no tendrá ningún pasajero
- public boolean hayLibres(). Devuelve true si quedan asientos libres y false si no quedan.
- public boolean equals(Object o). Dos vuelos son iguales si tienen el mismo identificador.
- public int reservarAsiento(String nombrePasajero, String nifPasajero): Reserva un asiento al pasajaro cuyos datos se indican como parámetro. Al pasajero se le asignará un asiento libre de forma aleatoria. El método devolverá el número de asiento que se le ha reservado.
    - Si el vuelo ya está completo, el método devuelve 0.
    - Si en el vuelo ya hay un pasajero igual al que se quiere reservar, se lanzará IllegalArgumentException(“pasajero duplicado”) y no se realizará la reserva.
- public int reservarAsiento(String nombrePasajero, String nifPasajero, char preferencia): Reserva un asiento al pasajaro cuyos datos se indican como parámetro:
    - El parámetro *preferencia* contendrá ‘P’ o ‘V’, indicando si se quiere reservar un asiento de pasillo o de ventanilla.
    - Si el vuelo ya está completo, el método devuelve 0.
    - Si en el vuelo ya hay un pasajero igual al que se quiere reservar, se lanzará IllegalArgumentException(“pasajero duplicado”) y no se realizará la reserva.
    - Se reserva el primer asiento que esté libre de la preferencia que se solicita (pasillo o ventanilla).. En caso de que no quede ningún asiento libre en la preferencia indicada (pref), se reservará el primer asiento libre de la otra preferencia. El método devolverá el número de asiento que se le ha reservado.
- public void cancelarReserva(int numasiento). Se cancela la reserva del asiento indicado. Si el asiento no estaba reservado, lanzará la excepción IllegalArgumentException
- public String toString(). Devuelve una String con los datos del vuelo y de los pasajeros de los pasajeros, de los asientos ocupados:

```
IB101 Valencia París 19:05:00 21:00:00
Pasajeros:
Asiento 1: 20174738M Sonia Dominguez
…
Asiento 23: 573638837K Fernando Romero
```

c) Diseñar una clase Java TestVuelo que permita probar la clase Vuelo y sus métodos. Para ello se desarrollará el método main en el que:

- Se cree el vuelo IB101 de Valencia a París, que sale a las 19:05 y llega a las 21:00
- Reservar varios asientos tanto indicando ventana o pasillo, como sin indicarlo. Prueba a reservar cuando el avión está lleno. Prueba a duplicar una reserva.
- Mostrar el vuelo por pantalla
- Cancelar la reserva del asiento que indique el usuario. Prueba a cancelar una reserva inexistente.
- Mostrar el vuelo por pantalla

5. (paquete: ejercicios.gestionhospital) Se desea realizar una aplicación para gestionar el ingreso y el alta de pacientes de un hospital. Una de las clases que participará en la aplicación será la clase Paciente, que se detalla a continuación :

a. La clase Paciente permite representar un paciente mediante los atributos: nombre (String), edad (entero), gravedad (entero entre 0 y 5, donde 0 se interpreta como que está curado y 5, que está muy grave), y con las siguientes operaciones:
   - public Paciente(String n, int e). Constructor de un objeto Paciente de nombre n, de e años y cuya gravedad es un valor **aleatorio** entre 1 y 5.
   - public int getEdad(). Consultor que devuelve edad.
   - public int getEstado(). Consultor que devuelve estado.
   - public void mejorar(). Modificador que mejora en uno la gravedad del paciente (mejora al paciente)
   - public void empeorar(). Modificador que empeora en uno la gravedad del paciente (empeora al paciente)
   - public String toString(). Transforma el paciente en un String. Los estados se mostrarán como "curado", "muy leve", "leve", "enfermo", "grave" o "muy grave". Por ejemplo,

     ```
     Pepe Pérez - 46 años - muy grave
     ```

   - public int compareTo(Object o). Indicar que la clase *implements Comparable<Paciente>* y añadir el método compareTo, teniendo en cuenta que se considera menor el paciente más leve. A igual gravedad, se considera menor el paciente más joven.

b. Diseñar una clase Java TestPaciente que permita probar la clase Paciente y sus métodos. Para ello se desarrollará el método main en el que:
   - Se crearán dos pacientes: "Juan" de 20 años y "Miguel" de 30 años.
   - Mostrar los datos del que se considere menor (según el criterio de compareTo de la clase Paciente).
   - Aplicar "mejoras" al paciente más grave hasta que los dos pacientes tengan el mismo estado.

c. La clase Hospital contiene la información de las camas de un hospital, así como de los pacientes que las ocupan. Un Hospital tiene un número máximo de camas MAXC = 200 y para representarlas se utilizará un array de tamaño MAXC + 1 (llamado listaCamas) de objetos de tipo Paciente junto con un atributo (numLibres) que indique el número de camas libres del hospital en un momento dado. El número de cada cama coincide con su posición en el array de pacientes (la posición 0 no se utiliza), de manera que listaCamas[i] es el Paciente que ocupa la cama i o es null si la cama está libre. Las operaciones de esta clase son:
   - *public Hospital().* Constructor de un hospital. Cuando se crea un hospital, todas las camas están libres.
   - *public int getNumLibres().* Consultor del número de camas libres.
   - *public boolean hayLibres().* Devuelve true si en el hospital hay camas libres y devuelve false en caso contrario.
   - *public int primeraLibre().* Devuelve el número de la primera cama libre del array listaCamas si hay camas libres o devuelve un 0 si no las hay.
   - *public void ingresarPaciente(String n, int e) throws IllegalOperationException* Si hay camas libres, la primera de ellas (la de número menor) pasa a estar ocupada por el paciente de nombre n y edad e. Si no hay camas libres, lanza una excepción.
   - private void darAltaPaciente(int i). La cama i del hospital pasa a estar libre. (Afectará al número de camas libres)
   - public void darAltas(). Se mejora el estado (método *mejorar() de Paciente)* de cada uno de los pacientes del hospital y a aquellos pacientes sanos (cuyo estado es 6) se les da el alta médica (invocando al método darAltaPaciente).
   - public String toString(). Devuelve un String con la información de las camas del hospital. Por ejemplo,

     ```
     1 María Medina – 30 años - leve
     ```

```
2 Pepe Pérez -  46 años - grave
3 libre
4 Juan López - 50 años – muy grave
5 libre
...
200 Andrés Sánchez - 29 años – muy leve
```

d. En la clase GestorHospital se probará el comportamiento de las clases anteriores. El programa deberá:
   - Crear un hospital.
   - Ingresar a cinco pacientes con los datos simulados introducidos directamente en el programa.
   - Realizar el proceso de darAltas mientras que el número de habitaciones libres del hospital no llegue a una cantidad (por ejemplo 198).
   - Mostrar los datos del hospital cuando se considere oportuno para comprobar la corrección de las operaciones que se hacen.

6. **(paquete: ejercicios.contrarreloj)** Se quiere realizar una aplicación para registrar las posiciones y tiempos de llegada en una carrera ciclista contrarreloj.

a) La clase **Corredor** representa a un participante en la carrera. Sus atributos son el dorsal (entero), el nombre (string), el instante de tiempo en que ha salido y el instante de tiempo en que ha llegado. Los métodos con los que cuenta son:

- public Corredor(int d, String n, Tiempo salida); Constructor a partir del dorsal, el nombre y el instante de salida.
- *Métodos getter y setter de todos los atributos.* El método setter del instante de llegada tendrá en cuenta que el instante de llegada no debe ser anterior al de salida. Si se produjera esta circunstancia, el método lanzará la excepción IllegalArgumentException.
- public int getDuracion(): Devuelve el tiempo tardado por el corredor. Si el corredor todavía no ha llegado, el método lanzará una excepción de tipo UnsupportedOperationException, que se pertenece al paquete java.lang
- public String toString(): Devuelve un String con los datos del corredor, como una de las dos siguientes (según sea el caso)

```
(234) -  Juan Ramirez – Salida: 10:20:30
```

```
(234) -  Juan Ramirez – Salida: 10:20:30 - Llegada: 10:30:30 - Duración: 600 segundos
```

- public boolean equals(Object o): Devuelve true si los corredores tienen el mismo dorsal y false en caso contrario.
- public int compareTo (Object o): Un corredor es menor que otro si tiene menor dorsal.
- public static int generarDorsal(): Devuelve un número de dorsal generado secuencialmente. Para ello la clase hará uso de un atributo *static int siguienteDorsal* que incrementará cada vez que se genere un nuevo dorsal.

b) Diseñar una clase Java **TestCorredor** que permita probar la clase Corredor y sus métodos. Para ello se desarrollará el método main en el que:

- Se crearán dos corredores: El nombre lo indicará el usuario mientras que el dorsal se generará utilizando el método generarDorsal() de la clase.
- Se mostrarán los datos de ambos corredores (toString)
- Se establecerá el instante de llegada de ambos corredores, en uno de ellos con valores correctos y en otro con valores incorrectos (instante de llegada anterior al de salida).
- Se mostrarán de nuevo los datos de ambos corredores.

c) La clase **ListaCorredores** permite representar a un conjunto de corredores. Para ello se utilizará un ArrayList, llamado lista.

Métodos:

- public ListaCorredores (). Construtor. Crea la lista de corredores, incicialmente vacía.
- public void añadir(Corredor c) throws *IllegalArgumentException*. Añade un corredor al final de la lista de corredores, siempre y cuando el corredor no esté ya en la lista, en cuyo caso se lanzará *IllegalArgumentException*
- public void insertarOrdenado(Corredor c). Inserta un corredor en la posición adecuada de la lista de manera que esta se mantenga ordenada crecientemente por duración del recorrido. Para poder realizar la inserción debe averiguarse la posición que debe ocupar el nuevo elemento e insertarlo en esa posición de la lista.
- public Corredor quitar(int dorsal) throws NoSuchElementException Quita de la lista al corredor cuyo dorsal se indica. El método devuelve el Corredor quitado de la lista. Si no se encuentra se lanza NoSuchElementException.
- public String toString() Devuelve un String con la información de la lista de corredores. Por ejemplo:

```
Posición: 1
```

```
(34) -   Juan Ramirez – Salida: 10:20:30 - Llegada: 10:30:30 -
Duración: 600 segundos Posición: 2
(56) -   Miguel Pérez – Salida: 10:30:00 - Llegada: 10:40:30 -
Duración: 630 segundos
…
…
```

d) (Clase **Contrarreloj**) Realizar un programa que simule una contrarreloj. Para llevar el control de una carrera contrarreloj se mantienen dos listas de corredores (dos objetos de tipo ListaCorredores):

- (hanSalido) Una con los que han salido, que tiene a los corredores por orden de salida. El atributo tiempo de estos corredores será 0. Para que los corredores se mantengan por orden de salida, se añadiran a la lista utilizando el método añadir.
- (hanLlegado) Otra con los corredores que hay llegado a la meta. A medida que los corredores llegan a la meta se les extrae de la primera lista, se les asigna un tiempo y se les inserta ordenadamente en esta segunda lista.
- En el método main realizar un programa que muestre un **menú** con las siguientes opciones
    - 1.- Salida: Para registrar que una corredor ha comenzado la contrarreloj y sale de la línea de salida. Solicita al usuario el nombre de un corredor, su dorsal y el instante de salida, y lo añade a la lista de corredores que han salido.
    - 2.- Llegada: Para registrar que un corredor ha llegado a la meta. Solicita al usuario el dorsal de un corredor y el instante de llegada de llegada. Quita al corredor de la lista de corredores que hanSalido, le asigna el instante de llegada y lo inserta (ordenadamente) en la lista de corredores que hanLlegado
    - 3.- Clasificación: Muestra la lista de corredores que hanLlegado. Dado que esta lista está ordenada por tiempo, mostrarla por pantalla nos da la clasificación.
    - 4.- Salir: Sale del programa

7. (paquete **ejercicios.gestorCorreoElectronico**) Queremos realizar la parte de un programa de correo electrónico que gestiona la organización de los mensajes en distintas carpetas. Para ello desarrollaremos:

a) La clase **Mensaje.** De un mensaje conocemos:

- Codigo (int) Número que permite identificar a los mensajes.
- Emisor (String): email del emisor
- Destinatario (String): email del destinatario.
- Asunto (String)
- Texto (String)

Desarrollar los siguientes métodos:

- Constructor que reciba todos los datos, excepto el código, que se generará automáticamente (nº consecutivo. Ayuda: utiliza una variable de clase (static) )
- Consultores de todos los atributos.
- *public boolean equals(Object o)*. Dos mensajes son iguales si tienen el mismo código.
- *public static boolean validarEMail(String email):* Método estático que devuelve true o false indicando si la dirección de correo indicada es válida o no. Una dirección es válida si tiene la forma *direccion@subdominio.dominio*
- public String toString()

b) Con la clase **TestCorreo** probaremos las clases y métodos desarrollados.

- Crea varios mensajes con los datos que introduzca el usuario y muéstralos por pantalla.
- Prueba el método *validarEMail* de la clase Mensaje con las direcciones siguientes (solo la primera es correcta):
    - tuCorreo@gmail.com
    - tuCorreogmail.com
    - tuCorreo@gmail
    - tuCorreo.com@gmail

c) La clase **Carpeta**, Cada carpeta tiene un nombre y una lista de Mensajes. Para ello usaremos un ArrayList. Además se implementarán los siguientes métodos:

- *public Carpeta(String nombre):* Constructor. Dado un nombre, crea la carpeta sin mensajes.
- *public void añadir(Mensaje m):* Añade a la carpeta el mensaje indicado.
- *public void borrar(Mensaje m) throws NoSuchElementException:* Borra de la carpeta el mensaje indicado. Lanza la excepción si el mensaje no existe.
- *public Mensaje buscar(int codigo):* Busca el mensaje cuyo código se indica. Si lo encuentra devuelve el mensaje, en caso contrario devuelve null.
- *public String toString()* que devuelva un String con el nombre de la carpeta y sus mensajes
- *public **static** void moverMensaje(Carpeta origen, Carpeta destino, int codigo):* Método estático. Recibe dos Carpetas de correo y un código de mensaje y mueve el mensaje indicado de una carpeta a otra. Para ello buscará el mensaje en la carpeta origen. Si existe lo eliminará y lo añadirá a la carpeta de destino y devolverá true. Si el mensaje indicado no está en la carpeta de origen devolverá false.

d) Con la clase **TestCarpetas** probaremos las clases y métodos desarrollados:

- Crea dos carpetas de correo de nombre Mensajes recibidos y Mensajes eliminados respectivamente.
- Crea varios mensajes y añádelos Mensajes recibidos.
- Mueve el mensaje de código 1 desde la Mensajes recibidos a Mensajes eliminados.
- Muestra el contenido de las carpetas antes y después de cada operación (añadir, mover,...)