Sesión L07- Interfaces

Para realizar esta práctica vamos a crear el directorio *L07-Interfaces*, dentro de nuestro directorio de trabajo (Practicas). Por lo tanto tendremos que hacer:

```
> cd Practicas
> mkdir L07-Interfaces
> cd L07-Interfaces
```

Este será el subdirectorio en el que vamos a trabajar durante el resto de la sesión. Por lo tanto cualquier fichero o directorio que creemos en esta sesión estará dentro de *L07-Interfaces.*

Copia la carpeta que os hemos proporcionado: ***sorting-L07*** en tu zona de trabajo *L07-Interfaces*.

01 Ordenando cualquier cosa

Vamos a comenzar a trabajar con el proyecto ***sorting-L07***.

En primer lugar crea la run configuration ***sorting-L07-run***, en la *carpeta eclipse-launch-files*, con las *goals*:

```
clean compile exec:java -Dexec.mainClass=pa.Demo1
```

El proyecto maven proporcionado está formado por tres clases: ***Sort***, ***Grade*** y ***Country***.. Abre el editor de código para la clase ***Sort***. Esta clase tiene implementados dos métodos estáticos. Recuerda que un método o campo estático tiene que invocarse directamente desde la clase, no desde los objetos instanciados de esa clase:

- método ***selectionSort(int [ ] a)*** es **público**, y ordena el array de enteros que se pasa como parámetro (en orden ascendente); para ello utiliza un algoritmo de ordenación conocido como el *método de selección*). Es un método **estático**, por lo que se invoca a partir de la clase.
- método ***minimumPosition()*** es **privado** y **estático**, y sólo puede accederse desde el método *selectionSort()* (un método estático no puede invocar a ningún método que no sea estático, por eso el método *minimumPosition()* tiene que ser también estático).

Recuerda que las interfaces nos permiten reducir la duplicación de código, compartiendo comportamientos entre clases dispares (entre las que no podemos establecer una relación "es un"). En este caso, tenemos un algoritmo para ordenar *arrays* de enteros (método *selectionSort(),* https://www.youtube.com/watch?v=xWBP4lzkoyM. Fíjate que hay muchos tipos de objetos que pueden ordenarse, por ejemplo Personas, Paises (ordenados por ejemplo por nombre), Notas (de un examen), Cartas (de una baraja), etc. El algoritmo para ordenar es idéntico en todos los casos, sólo cambia el tipo de datos. Y es evidente que un Pais, no tiene nada que ver con una Carta de una baraja, o una Persona (no podemos utilizar herencia para compartir el comportamiento entre ellos).

Utilizaremos interfaces para **no tener que duplicar el código de ordenación** para todas las clases que puedan ser *ordenadas.* Si en lugar de un array de enteros, queremos ordenar otro tipo de datos, tendríamos que crearnos los métodos *sortPaises(), sortPersonas(), sortCartas()*,... En lugar de eso, podremos reutilizar el código de la clase *Sort* si en el método *selectionSort()* cambiamos el tipo del parámetro por un tipo interfaz. De esta forma podremos ordenar cualquier tipo de objeto que implemente dicha interfaz.

- Lo primero que haremos será crearnos la interfaz ***pa.interfaces.IOrdenable,*** (recuerda que como convención, las interfaces las definiremos precedidas con una "`I`" para identificarlas fácilmente). Para crear la interfaz, lo haremos desde el menú contextual del directorio de fuentes del proyecto seleccionando la opción ***New→ Interface***. La interfaz *IOrdenable* define los comportamientos que queremos compartir entre las diferentes clases.

  Si te fijas en el código de *selectionSort()*, ¿qué operación/es se utilizan para realizar la ordenación?, y ¿cuáles de ellas son dependientes del tipo de parámetro concreto?, en otras

palabras ¿qué es lo que diferencia la ordenación de Paises de la de Cartas de una baraja, o las Notas de un examen? No es la operación de asignación, pues esto podemos hacerlo con cualquier tipo de objeto. Es la **comparación entre dos elementos**. La comparación es la que desencadena el intercambio de posición de los elementos a ordenar. Cada clase tiene su forma particular de hacer esa comparación. Por lo tanto, vamos a elegir como comportamiento común el método "***boolean menorQue (IOrdenable a)***". Este método devolverá *true* si el objeto sobre el que se invoca (que también debe ser de tipo *IOrdenable*) es menor que el objeto *a*, y *false* en caso contrario. Este método se implementará de forma diferente en cada clase que implemente la interfaz *IOrdenable* y hará posible la ordenación de cualquier tipo *IOrdenable*, compartiendo el mismo código (el código del método *selectionSort()*)

- Ahora cambiamos el código del método ***Sort.selectionSort()*** para que funcione con arrays de objetos de tipo *IOrdenable*. Necesitas cambiar el tipo array de enteros por el tipo *IOrdenable.* También necesitas cambiar la forma de comparar dos objetos de forma que no se haga con el operador "<", sino utilizando el método *menorQue()* de la interfaz.

- Sólo nos queda hacer que ***Grade*** y ***Country*** implementen la interfaz ***IOrdenable***, proporcionando cada uno de ellos su forma particular de implementar el método *menorQue()*. Dados dos objetos de tipo *IOrdenable*, por ejemplo *objeto1*, y *objeto2*, la invocación: *objeto1.menorQue(objeto2)* debe devolver *true* si *objeto1* es anterior a *objeto2* si los ordenásemos en orden creciente. Devolverá *false* en caso contrario.

  Veamos algunos detalles de cómo sería una posible implementación para la clase *Country*. El parámetro del método *menorQue()* es de tipo *IOrdenable*. Country implementa la interfaz *IOrdenable*, y por lo tanto es de tipo *IOrdenable*, pero también es de tipo *Country*. Para hacer la comparación entre países necesitamos utilizar el tipo *Country*, por lo que podemos "convertir" una variable de tipo *IOrdenable* a tipo *Country* utilizando la sentencia `(Country) objeto;` siendo objeto de tipo *IOrdenable*.

  Por otro lado, la clase *Country* tiene un campo *name* de tipo *String*. Por lo tanto, la operación *menorQue()* tendría que tener en cuenta el orden lexicográfico de ese campo para decidir si un país es menor que otro o no. El tipo *String* es un tipo de la librería de Java, si consultamos el javadoc de este tipo (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html). veremos que podemos utilizar el método `compareTo()`, que precisamente implementa lo que estamos buscando. Este método devuelve cero si las cadenas de caracteres son iguales, un número menor que cero si la primera cadena es menor que la segunda, y un número mayor que cero en otro caso. Para saber qué métodos son aplicables a un objeto podemos usar el autocompletado de Eclipse.

El editor de Eclipse permite autocompletar código. Simplemente a partir de un objeto, después de pulsar el ".", verás que aparece una ventana con una lista de todos los métodos que puedes utilizar para dicho objeto. Seleccionando uno de ellos, verás que se autocompleta la sentencia en el editor.

Volviendo a la implementación de la clase *Country*, y según todo lo expuesto, una posible implementación para el método *Country.menorQue()* podría ser ésta:

```
@Override
public boolean menorQue(IOrdenable o) {
  Country pais = (Country)o;
  return (name.compareTo(pais.name) < 0);
}
```

Implementa este método también para la clase *Grade*.

- Para probar la ordenación elementos de las dos clases (*Grade* y Country), utilizando el código *Sort.selectionSort*() para todos ellos, crea una clase ***pa.Demo1.*** Esta clase debe

contener un único método main en el que implementaremos el código con las entradas que se muestran en la **Figura 1**, y el resultado debe ser también el mostrado en la **Figura 1**.

Nota: debes usar arrays para almacenar los datos de entrada, así como bucles siempre que sea posible.

```
Original (Paises)
=================
  España  Venezuela  Francia  Bélgica  Holanda

Ordenado (Paises)
=================
  Bélgica  España  Francia  Holanda  Venezuela

Original (Notas)
================
  8.5  4.5  0.2  5.0  1.6

Ordenado (Notas)
================
  0.2  1.6  4.5  5.0  8.5
```

Figura 1. Entradas y resultados de ordenación de objetos de tipos diferentes con un único código gracias al uso de interfaces

Sube GitHub el trabajo realizado hasta el momento:

```
> git add .   //desde el directorio que contiene el repositorio git
> git commit -m"Terminado el ejercicio 1 de L07: sorting"
> git push
```

02 Más interfaces ... pero ahora creando un proyecto maven desde cero

En este ejercicio vamos a trabajar con el proyecto Maven ***hospital***, creándolo desde cero.

Crea un nuevo proyecto Maven (aplicación Java) en tu directorio de trabajo (*L07-interfaces*). Ya lo hemos explicado en el ejercicio 4 de L05.

El *Group Id* del proyecto será ***pa***, el *Artifact Id*: ***hospital-L07,*** y el valor de *packaging* será ***jar.*** Recuerda modificar el fichero pom.xml del proyecto para que incluya las etiquetas *<packaging>* y *<properties>*.

Crea las siguientes *Run Configurations* (en la subcarpeta "***eclipse-launch-files***", igual que hemos hecho en prácticas anteriores),

- ***hospital-L07-sinInterfaz-run ,*** con las goals:

  `compile exec:java -Dexec.mainClass=pa.hospital.DemoSinInterfaz`

- ***hospital-L07-conInterfaz-run ,*** con las goals:

  `compile exec:java -Dexec.mainClass=pa.hospital.DemoConInterfaz`

Queremos **gestionar las salas** de un hospital, en concreto estamos interesados en poder **hacer reservas** y **anular reservas** para diferentes **tipos de salas**. Implementa dos soluciones, una sin usar interfaces y otra usando interfaces, en los paquetes ***pa.hospital.sinInterfaz***, y ***pa.hospital.conInterfaz***, respectivamente.

Supón que, de momento, solamente hay dos tipos de salas: **quirófano**, y **sala de video**, cada una con sus correspondientes métodos *reservar()* y *anularReserva().* Entre ambas clases no hay una relación de herencia puesto que no comparten atributos. Un quirófano se identifica con una cadena de caracteres, y una sala de vídeo se identifica con un entero. Debes tener en cuenta que el identificador asociado a cada sala no se puede cambiar, aunque sí se puede consultar. Crea un constructor con un parámetro "nombre" (de tipo String) y genera los correspondientes *getters* y/o

setters que procedan. Puedes generarlos de forma automática desde el editor de texto. Sitúate en la línea en la que quieras añadir el código, y, con botón derecho, usando las opciones: "*Source→Generate Getters and Setters...*", y *Source→Generate Constructor using Fields...*").

A continuación te indicamos una guía para que implementes el proyecto.

VERSIÓN SIN INTERFACES.

Comenzaremos implementando la versión SIN usar interfaces. Por lo tanto, trabajaremos en el paquete ***pa.hospital.sinInterfaz***. Lo primero que tenemos que hacer es determinar qué clases necesitamos para modelar el problema. Necesitamos 3 clases, a las que llamaremos ***GestionSalasHospital***, ***Quirofano*** y ***SalaVideo***.

La clase ***GestionSalasHospital*** tendrá los métodos (cuatro en total) para realizar reservas y anularlas para cada uno de los **tipos de salas** que se pasarán por **parámetro** (puedes usar los nombres de métodos que consideres oportuno). No vamos a hacer una implementación real de los métodos, sino que simplemente imprimiremos por pantalla un mensaje para indicar que se está ejecutando dicho método. Podemos usar, por ejemplo, los mensajes: “Reservando quirófano X”, “Reservando de sala de vídeo X”, “Anulada reserva de quirófano X”, y “Anulada reserva de sala de vídeo X”,en donde X representa el identificador de cada sala.

Una vez que tengas implementadas las clases, usa una cuarta clase, a la que llamaremos ***DemoSinInterfaz*** (en el paquete ***pa.hospital***), con un método ***main()***, desde el que crearemos una instancia de GestionSalasHospital, y haremos varias reservas y anulación de las mismas de los tipos de salas de los que disponemos ahora mismo (por ejemplo dos reservas y dos anulaciones para cada uno de los tipos de salas).

El resultado de la ejecución del método pa.hospital.DemoSinInterfaz.main() debe ser similar al que mostramos a continuación (dependerá del identificador que hayas usado y del número de salas que hayas creado. Para practicar con arrays, puedes crear 7 quirófanos y 10 salas de vídeo.

Nota: ninguno de los mensaje se imprimen desde main(), sino desde los métodos que son invocados desde main().

```
Reservando quirófano Q1
Reservando quirófano Q2
Anulando reserva de quirófano Q1
Anulando reserva de quirófano Q2
Reservando sala de vídeo 1
Reservando sala de vídeo 2
Anulando reserva de sala de vídeo 1
Anulando reserva de sala de vídeo 2
```

VERSIÓN CON INTERFACES.

Ahora implementa una segunda solución pero utilizando una interfaz para evitar duplicar código en la clase ***GestionSalasHospital***. Tendrás que trabajar en el paquete **pa.hospital.conInterfaz.**

En las clases ***Quirofano*** y ***SalaVideo*** vamos a realizar unas modificaciones con respecto a la versión sin interfaces. Queremos que el identificador de cada clase tenga un valor ÚNICO para cada objeto. Es decir que no puede haber dos objetos con el mismo valor para dicho atributo.

La idea es que cuando creemos un nuevo objeto de tipo Quirofano, se le asigne de forma automática el identificador "quirofano-X", siendo X un entero que siempre se irá incremenndo en una unidad. Por ejemplo, si creamos tres objetos Quirofano, se identificarán como "quirofano-0", "quirofano-1", y "quirofano-2" respectivamente.

Lo mismo ocurrirá con los objetos de tipo *SalaVideo*, sólo que en este caso, los identificadores serán 100, 101, 102,....

Tendrás que modificar el constructor (puesto que ya no necesitamos el parámetro). Podremos consultar los identificadores de cada sala, pero no cambiarlos

Cuando reservemos o anulemos un quirófano el mensaje que debe mostrarse por pantalla será: "Reservado el quirófano <id>", y "Anulada reserva del quirófano <id>", respectivamente. Siendo <id> el valor del identificador para ese objeto.

De forma análoga, para las reservas y anulaciones de salas de vídeo mostraremos los mensajes "---> Reservada la sala <id>", y "---> Anulada la reserva de la sala de video <id>", respectivamente. Siendo <id> el valor del identificador para ese objeto.

Finalmente crea una clase **pa.hospital.DemoConInterfaz** con un método main() en el que vamos a crear 5 quirófanos y 10 salas de vídeo. Y a continuación hagamos una reserva de todas ellas, y después anulemos las reservas que hemos realizado previamente.

Nota: debes usar variables final en el ejercicio. Piensa qué variables pueden ser finales. Razona qué ventajas tiene el que las variables sean final o no lo sean.

El resultado de la ejecución debe ser el que mostramos a continuación. Igual que antes, todos los mensajes se imprimen desde los métodos invocados desde main().

```
    Reserva del quirófano: quirofano-0
    Reserva del quirófano: quirofano-1
    Reserva del quirófano: quirofano-2
    Reserva del quirófano: quirofano-3
    Reserva del quirófano: quirofano-4
---> Reserva de la sala de video 100
---> Reserva de la sala de video 101
---> Reserva de la sala de video 102
... //Y asi sucesivamente
---> Reserva de la sala de video 108
---> Reserva de la sala de video 109
    Anulada reserva del quirófano: quirofano-0
    Anulada reserva del quirófano: quirofano-1
    Anulada reserva del quirófano: quirofano-2
    Anulada reserva del quirófano: quirofano-3
    Anulada reserva del quirófano: quirofano-4
---> Anulada reserva de sala de video 101
---> Anulada reserva de sala de video 101
... //y así sucesivamente
---> Anulada reserva de sala de video 101
```

Observa bien las diferencias entre ambas soluciones.

Recuerda que la práctica no está terminada si no te queda claro el uso de las interfaces, variables static y final, que hemos explicado en teoría.

Sube GitHub el trabajo realizado hasta el momento:

```
> git add .  //desde Practicas
> git commit -m"Terminado el ejercicio 2 de L07: hospital"
> git push
```

03 Proyecto ascensor: cambiando la entrada de datos usando interfaces

Usaremos el proyecto maven ***ascensor-L07***, que os hemos proporcionado, y que está parcialmente implementado.

Hemos creado tres ***Run Configurations***, que puedes consultar.

Disponemos de las clases:

- ***pa.Peticion***: representa la petición de un usuario que quiere subir al ascensor. Implementa esta clase siguiendo las indicaciones de los comentarios incluidos en el código.
- ***pa.sinInterfaces.Ascensor***: representa un ascensor, en el que caben 4 personas como máximo, y capaz de recorrer hasta 20 plantas. Implementa esta clase siguiendo las indicaciones de los comentarios incluidos en el código.
- ***pa.sinInterfaces.DemoSinInterfaces***: contiene un método ***main()*** para probar la aplicación.

Una vez que hayas implementado las clases Petición y Ascensor, usa la ***Run Configuration*** que corresponda para ejecutar el método ***main()*** de la clase *DemoSinInterfaces* anterior para probar el funcionamiento de la aplicación.

La aplicación lee las peticiones de los usuarios desde el teclado, usando la clase *Scanner*, que ya conocemos de prácticas anteriores.

A continuación mostramos un ejemplo de ejecución del programa (en rojo mostramos la entrada del usuario desde el terminal. Todas las entradas terminan con el carácter 'A').

```
---------------------------------
Estoy en el piso: 0
Puertas abiertas. Espero peticiones:
3 5 22 -2 19 0 5 7 A
  Entran en el ascensor las personas que van a los pisos: 3, 5, 19, 5,
  AVISOS:
     - El usuario que ha pulsado 22 ha introducido un valor incorrecto
     - El usuario que ha pulsado -2 ha introducido un valor incorrecto
     - El usuario que ha pulsado 0 ya está en esa planta
     - El usuario que ha pulsado 7 ya no cabe
Cerrando puertas. Estamos en el piso: 0
      Subiendo a una persona la planta 3
      Subiendo a una persona la planta 5
      Subiendo a una persona la planta 19
      Bajando a una persona la planta 5

---------------------------------
Estoy en el piso: 5
Puertas abiertas. Espero peticiones:
1 0 3 -2 17 17 A
  Entran en el ascensor las personas que van a los pisos: 1, 0, 3, 17,
  AVISOS:
     - El usuario que ha pulsado -2 ha introducido un valor incorrecto
     - El usuario que ha pulsado 17 ya no cabe
Cerrando puertas. Estamos en el piso: 5
      Bajando a una persona la planta 1
      Bajando a una persona la planta 0
      Subiendo a una persona la planta 3
      Subiendo a una persona la planta 17

---------------------------------
Estoy en el piso: 17
Puertas abiertas. Espero peticiones:
3 3 5 5 3 A
  Entran en el ascensor las personas que van a los pisos: 3, 3, 5, 5,
  AVISOS:
     - El usuario que ha pulsado 3 ya no cabe
Cerrando puertas. Estamos en el piso: 17
      Bajando a una persona la planta 3
            La siguiente persona también puede bajar
      Subiendo a una persona la planta 5
            La siguiente persona también puede bajar

---------------------------------
Estoy en el piso: 5
Puertas abiertas. Espero peticiones:
1 1 1 1 10 12 A
  Entran en el ascensor las personas que van a los pisos: 1, 1, 1, 1,
  AVISOS:
     - El usuario que ha pulsado 10 ya no cabe
     - El usuario que ha pulsado 12 ya no cabe
Cerrando puertas. Estamos en el piso: 5
      Bajando a una persona la planta 1
            La siguiente persona también puede bajar
            La siguiente persona también puede bajar
            La siguiente persona también puede bajar

---------------------------------
Estoy en el piso: 1
Puertas abiertas. Espero peticiones:
5    7 A
  Entran en el ascensor las personas que van a los pisos: 5, 7,
Cerrando puertas. Estamos en el piso: 1
      Subiendo a una persona la planta 5
      Subiendo a una persona la planta 7
```

(continuación)

```
--------------------------------
Estoy en el piso: 7
Puertas abiertas. Espero peticiones:
-8 -5  33  A
  AVISOS:
     - El usuario que ha pulsado -8 ha introducido un valor incorrecto
     - El usuario que ha pulsado -5 ha introducido un valor incorrecto
     - El usuario que ha pulsado 33 ha introducido un valor incorrecto

Ascensor inactivo.
```

VERSIÓN DE LA APLICACIÓN DE ASCENSOR CON INTERFACES

Vamos a implementar una nueva versión de la aplicación usando una interfaz que nos va a permitir obtener las peticiones de diferentes formas reutilizando el código de la clase Ascensor.

Importante: Todo el código de la versión con interfaces lo implementaremos en el paquete ***pa***.

Cambiaremos el diseño de la clase Ascensor, según el diagrama de clases de la **Figura 2** que mostramos a continuación:

En el diagrama sólo mostramos la relación de la clase Ascensor con la interfaz que usaremos para leer las peticiones realizadas (método ***obtenerDatos()***). Cada valor leído se corresponde con el número de piso al que quiere ir cada persona que solicita usar dicho ascensor.

Proporcionamos el código de la clase ***pa.EntradaDesdeFichero***, que tendrás que modificar para poder usarla desde la clase Ascensor.

Tendrás que implementar la clase ***pa.EntradaDesdeTeclado.***

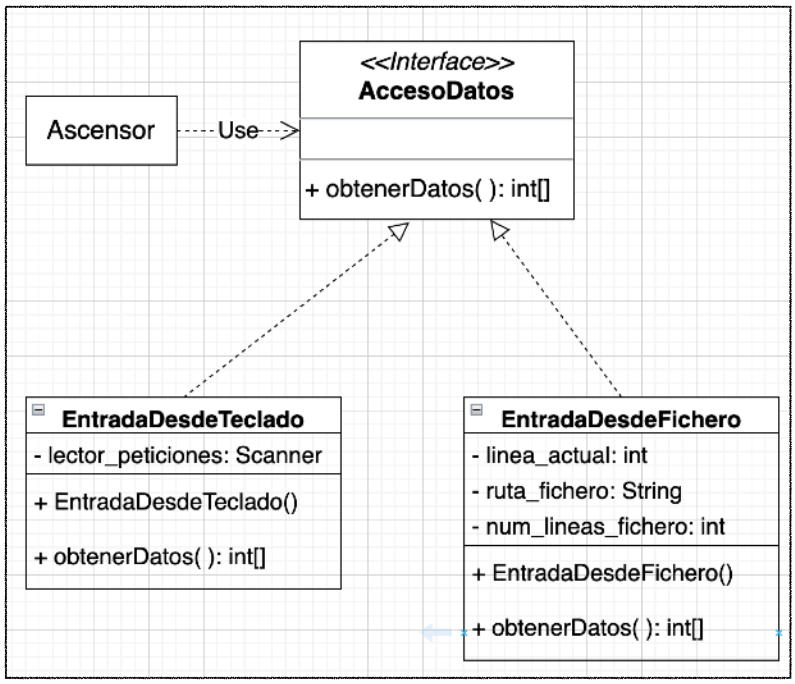


Figura 2.Interfaz AccesoDatos

Copia el fichero Ascensor.java (del paquete pa.sinInterfaces) en el paquete pa. Y modifica convenientemente la clase pa.Ascensor para poder usar la interfaz (que también deberás crear).

El constructor de pa.Ascensor tendrá como parámetro un objeto de tipo AccesoDatos.

Crea la clase ***pa.Demo***, con un método ***main()*** para probar la aplicación. Puedes copiar el código de la clase *pa.DemoSinInterfaces.Demo* y realizar las siguientes modificaciones:

- En esta ocasión vamos a usar el array de cadenas de caracteres que tiene como parámetro el método main().

  Dado que el método main() es invocado desde la máquina virtual de java, indicaremos los valores del array cuando ejecutemos la aplicación.

En nuestro caso, como estamos ejecutando desde maven, necesitamos usar la variable `exec.arguments`, a la que asignaremos los valores del array que queremos pasar como parámetros al método main(). Para indicar a maven que se trata de una variable, anteponemos `-D` al nombre de la variable.

Vamos a usar las cadenas de caracteres "fichero" y "terminal" para indicar a nuestro programa que vamos a leer los datos desde el terminal o desde un fichero.

Si miras las Run configurations que os hemos proporcionado, verás que por ejemplo: que las goals para ***ascensor-run-fichero*** son:

```
clean compile exec:java -Dexec.mainClass=pa.Demo -Dexec.arguments=fichero
```

En este caso, al ejecutar el método main(), de la clase pa.Demo, estaremos pasando como parámetro a dicho método un array con un único elemento cuyo valor es la cadena de caracteres "fichero".

En nuestro *main()* por lo tanto, la sentencia:

`args.length`, tendría como valor 1, ya que el array `args` tiene un único elemento (que ocuparía la posición 0).

Y si queremos consultar el valor pasado por parámetro lo haríamos con `args[0]`

Por lo tanto, vamos a modificar el código de pa.Demo, de forma que si el valor del argumento pasado por parámetro es "fichero", entonces estamos indicando que nuestro programa leerá los datos de un fichero (y por lo tanto, usará el método obtenerDatos() de la clase EntradaDesdeFichero, y si el valor del argumento pasado por parámetro es "terminal", entonces el programa leerá los datos desde el teclado.

- El bucle do..while es similar al de pa.sinInterfaces.Demo, salvo que en esta versión, después de seleccionar las peticiones válidas mostraremos el mensaje "Número total de peticiones: X, Peticiones rechazadas: Y", siendo X e Y el número de total de peticiones y el número de peticiones no válidas, respectivamente.

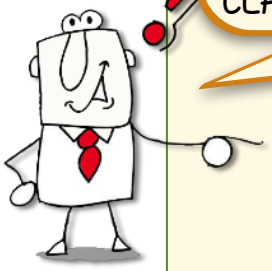
A continuación mostramos un ejemplo de ejecución desde el terminal:

```
--------------------------------
Estoy en el piso: 0
Puertas abiertas. Espero peticiones:
5 5 4 4 A
  Entran en el ascensor las personas que van a los pisos: 5, 5, 4, 4,
   Número total de peticiones: 4, Peticiones rechazadas: 0

Cerrando puertas. Estamos en el piso: 0
      Subiendo a una persona la planta 5
            La siguiente persona también puede bajar
      Bajando a una persona la planta 4
            La siguiente persona también puede bajar

--------------------------------
Estoy en el piso: 4
Puertas abiertas. Espero peticiones:
A
   Número total de peticiones: 0, Peticiones rechazadas: 0

Ascensor inactivo.
```

La *run configuration* ***ascensor-L07-run-fichero***, nos permite cambiar la entrada y usar los datos amacenados en un fichero de texto. En la carpeta resources puedes ver el fichero con nombre ***peticiones-fichero.txt*** con los datos de prueba. Puedes ver el contenido de dicho fichero desde el editor de Eclipse. También tienes otro fichero ***peticiones-teclado.txt*** con los mismos datos de prueba. En este caso, si ejecutamos el programa introduciendo las entradas desde el terminal podemos ir copiando cada línea y pegarla durante la ejecución del programa, cuando nos pida que introduzcamos los datos.

Sube GitHub el trabajo realizado hasta el momento:

```
> git add .  //desde Practicas
> git commit –m"Terminado el ejercicio 3 de L07: ascensor"
> git push
```



¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

- El tipo interfaz define un conjunto de servicios (signaturas de métodos) que pueden se compartidos (usados) por objetos de tipos diferentes.
- El tipo interfaz permite reducir la duplicación de código cuando no se puede usar herencia (no existe una relación "es un" entre los tipos de objetos
- El tipo interfaz puede declarar constantes (requieren del modificador final). Dichas constantes serán compartidas por todos los objetos de ese tipo, por lo que requerirán del modificador static.
- El modificador static puede aplicarse tanto a atributos como a métodos de una clase. Un método static será invocado a partir de la clase (no de las instancias de esa clase.
- Una clase pueden implementar CUALQUIER número de interfaces
- A una variable de tipo interfaz, podemos asignarle cualquier tipo que IMPLEMENTE todos los métodos especificados en dicha interfaz. Para ejecutar el método correspondiente se hace uso de polimorfismo.