



Sesión L04- Agrupando objetos 1

01. [Preparamos el directorio de trabajo](#)
02. [Organizamos un proyecto en paquetes de clases Java](#)
03. [Documentamos nuestro proyecto](#)
04. [Trabajando con Arrays \(de tipos primitivos\)](#)
05. [Detalles de las clases: implementación en Java](#)
06. [Empaquetando nuestro proyecto Maven](#)

[Recuerda!](#)

**01**

Preparamos el directorio de trabajo



Para realizar esta práctica vamos a crear el directorio *L04-Colecciones1*, dentro de nuestro directorio de trabajo (Practicas). Nos situaremos en dicho directorio, y crearemos el subdirectorio correspondiente a esta práctica:

```
> cd Practicas
> mkdir L04-Colecciones1
> cd L04-Colecciones1
```

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 *L04-Colecciones1*.

Desde la carpeta **plantillas-L04**, copia las subcarpetas que os hemos proporcionado: **visorImágenes-L04**, **notas-L04** y **centroMedico-L04** en tu zona de trabajo *L04-Colecciones1*. Cada una de ellas contiene un proyecto maven.

02

Organizamos un proyecto en paquetes de clases Java

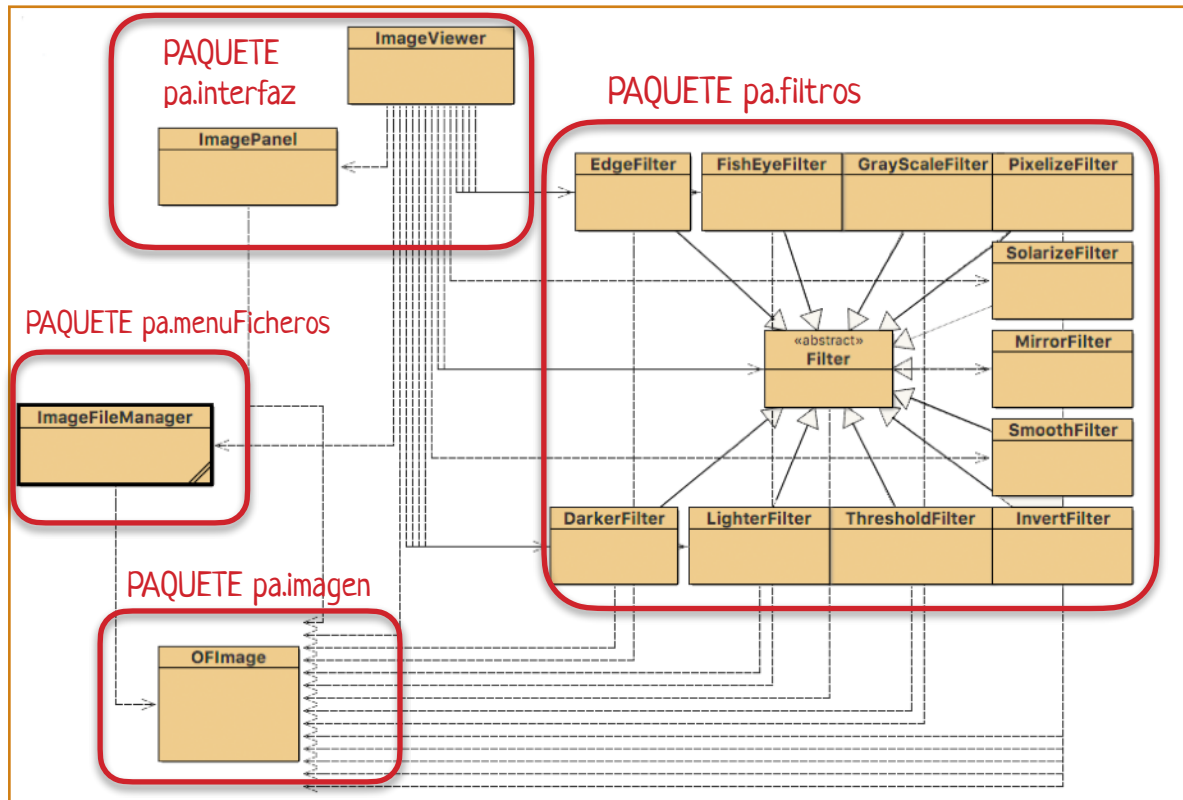


Vamos a comenzar a trabajar con el proyecto **visorImágenes-L04**. Importa el proyecto desde Eclipse. Se pide:

- El proyecto está organizado en un **único paquete** que contiene TODAS las clases de dicho proyecto (paquete **sinOrganizar**). Ya hemos visto que un paquete es una agrupación lógica de clases. Físicamente, ese paquete se almacenará como una carpeta.

Fíjate que cada fichero *.java* con la implementación de una clase tiene indicado en la primera línea el nombre del paquete al que pertenece dicha clase. Dado que dos paquetes pueden contener una clase diferente pero con el mismo nombre, identificaremos una clase como **nombrePaquete.nombreClase**. Así, por ejemplo, la clase **sinOrganizar.Filter** hace referencia a la clase **Filter**, que pertenece al paquete **sinOrganizar**.

Figura 1. Diagrama de clases *visorImágenes-L04*



- La **Figura 1** muestra un diagrama de clases en el que se sólo se muestran los nombres de las clases y sus relaciones entre ellas. Aparece un tipo de flecha que todavía no hemos visto en clase, pero en cualquier caso, dos clases unidas por una flecha siempre indica que hay alguna relación entre ellas. Nosotros solo conocemos, de momento, la relación de dependencia "use". Mas adelante veremos que existen otro tipo de relaciones posibles entre las clases. En cualquier caso, tener todas las clases en el mismo paquete no es nada práctico, y es muy importante **modularizar** adecuadamente nuestro proyecto.

Por ello tenéis que crear los paquetes que se muestran en rojo, de forma que contengan (agrupen) cada uno de ellos las clases indicadas en la figura. Para crear los paquetes puedes hacerlo desde el menú contextual del elemento "src/main/java" y seleccionas **New → Package**. Indica el nombre y marca la casilla "**Create package-info.java**" (más adelante veremos la utilidad de marcar esta casilla).

Nota: podemos crear una jerarquía de paquetes (paquetes que contienen a otros paquetes), para lo cual usamos un "punto". Por ejemplo, si creas el paquete **pa.filtros**, estás creando el paquete **pa**, y "dentro" de él, el sub paquete **filtros**. Físicamente estarás creando la carpeta **pa** y la subcarpeta **filtros**.

Ahora **moveremos** las clases a sus nuevos paquetes. Puedes seleccionar varios ficheros java (manteniendo pulsada la tecla ctrl.) y a continuación desde el menú contextual selecciona **Refactor → Move...** Fíjate que al mover una clase, el nombre del **paquete** de la clase es diferente al de la original!!! Eclipse, de forma automática, "reubica" la clase al nuevo paquete al que la hemos movido.

- Borra** el paquete *sinOrganizar*.
- Para asegurarnos de que todo se ha "movido" correctamente. Desde cada paquete, seleccionaremos **Source → Organize Imports...** Si Eclipse te muestra un error indicando que ha tenido problemas al organizar los imports, entonces tendrás organizar los imports fichero a fichero para solucionar posibles conflictos y/o ambigüedades.

IMPORTANTE: Todas las clases que pertenecen al mismo PAQUETE son accesibles entre ellas. Es decir, si desde la clase `SmoothFilter` necesitamos usar un objeto `DarkFilter`, podremos hacerlo porque "lógicamente" ambas clases pertenecen al mismo paquete. Sin embargo, desde un paquete, NO se tiene acceso "directo" a ningún otro paquete, y sólo podremos usar clases de otros paquetes si las IMPORTAMOS. Por ejemplo, si queremos usar un objeto de tipo `DarkFilter` desde la clase `ImagePanel`, tendremos que IMPORTAR la clase `DarkFilter` antes de la definición de la clase `ImagePanel`, para poder usarla.

- Vamos a crear una clase para probar nuestro proyecto (en `src/main/java`), con **New→Class**. La nueva clase pertenecerá al paquete **pa** y se llamará **Demo**. Marca la casilla "`public static void main...`", para que te cree de forma automática el método `main()`. Implementa el método `main()`. En este caso, sólo necesitas crear un objeto de tipo **pa.interfaz.ImageViewer** (invocando al constructor correspondiente).

Nota: debes crear una variable local de tipo `ImageViewer`. Verás que Eclipse marcará esa línea con un aviso de error. Fíjate que la clase **ImageViewer** que quieres usar está en el paquete **pa.interfaz**. Para poder tener acceso al contenido del paquete **pa.interfaz** desde tu clase **pa.Demo**, debes **IMPORTAR** dicha clase. Esto lo puedes hacer escribiendo (después de la primera línea y antes de la definición de la clase) la sentencia **import pa.interfaz.ImageViewer;**

- Ahora vamos a construir nuestro proyecto. Crea una **Run Configuration** (que deberás guardar en el directorio **eclipse-launch-files**) la carpeta con nombre **visor-L04-Demo-run**. En el apartado 1 del enunciado de **L02** tienes las instrucciones para crear una nueva **Run Configuration**. En el campo **Goals** usa:

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

IMPORTANTE: Recuerda revisar la información de las pestañas Main, JRE y Common para asegurarte de que la Run Configuration se crea correctamente.

- desde Main configuraremos el nombre de la Run Configuration, el directorio base y las goals a ejecutar
- desde JRE siempre marcaremos "Workspace default JRE"
- desde Common siempre marcaremos "Shared file" para guardar el fichero `.launch` en la carpeta `eclipse-launch-files`

Ejecuta **visor-L04-Demo-run**. Puedes usar las imágenes de la carpeta **src/main/resources/imagenes** del proyecto **visorImagenes-L04** para probar la aplicación.

IMPORTANTE: Para terminar correctamente la ejecución de la aplicación tienes dos opciones:

1. Cerrar la aplicación desde el menú **File→Quit**
2. Si cierras la ventana usando el botón rojo con la "x" de dicha ventana, tendrás que pulsar además el icono con un cuadrado rojo de la vista Console de Eclipse. Si no lo haces así, el proceso seguirá en ejecución (hablaremos de los procesos más adelante). El icono se volverá gris si el proceso ha terminado.

- Observa que la ventana gráfica muestra el título "ImageViewer (sin organizar)". Cambiaremos este título modificando la primera línea del método privado **makeFrame()** de la clase **pa.interfaz.ImageViewer**. El nuevo título será: **"ImageViewer Prácticas PA"**

Eclipse, cuando muestra la descripción abstracta de una clase asocia un cuadrado rojo a los métodos privados y un círculo verde a los métodos públicos.

En general cualquier elemento público tendrá asociado un icono VERDE y si el elemento es privado su color será ROJO, con independencia de la forma del icono (cuadrado, círculo...)



Nota: Observa que en la vista Project también puedes ver la descripción abstracta de las clases desplegándolas (la información que muestra es la misma que la de la vista *Outline*). Si quieres puedes cerrar la vista *Outline* o mantenerla (lo que te resulte más cómodo).



Sube a GitHub el trabajo realizado hasta el momento:

```
> git add . //desde el directorio que contiene el repositorio git
> git commit -m"Terminado el ejercicio 1: visorImágenes, de L04"
> git push
```

03

Documentamos nuestro proyecto



El proyecto que os hemos proporcionado (**visorImágenes**), contiene comentarios Javadoc. Por ejemplo:

```
/**
 * ImageViewer is the main class of the image viewer application. It builds and
 * displays the application GUI and initialises all other components.
 *
 * To start the application, create an object of this class.
 *
 * @author Michael Kölling and David J. Barnes.
 * @version 3.1
 */
public class ImageViewer {...}
```

Comentarios



Javadoc

Los comentarios con este formato especial son utilizados por la herramienta **Javadoc** para generar de forma automática documentación sobre los paquetes, clases, propiedades y métodos de nuestra aplicación.

Eclipse nos muestra la información *Javadoc* desde la **vista javadoc**, tal y como hemos indicado en clase.

También podemos ver la documentación javadoc desde el **editor** de código. Si posicionamos el ratón sobre algún elemento (clase, paquete, método) con comentarios javadoc, Eclipse nos muestra una ventana emergente con dicha documentación

Los comentarios *Javadoc* se caracterizan por:

- Comienzan por "/*" y terminan por "*/"
- Al inicio de cada línea se utiliza el carácter "*"
- En la primera línea indicamos una descripción corta del elemento a documentar

```
/**
 * [descripción corta]
 * <p>
 * [descripción larga]
 *
 * [ @author, @version, @params, @returns,
 *   y otras "tags" ]
 */
```



- Podemos incluir una descripción más larga, para ello usaremos la etiqueta `<p>` en la siguiente línea separando así ambas descripciones
- También podemos usar diferentes *tags* (van precedidas por `@`) con información específica, por ejemplo el autor del código, la versión, los parámetros, el valor devuelto... Pondremos un *tag* por línea. Los *tags* estarán separados de las descripciones por una línea en blanco
- Cada comentario javadoc se escribe inmediatamente antes del elemento que queremos documentar: clase, propiedad, método.
- Para documentar los paquetes, incluiremos los comentarios javadoc en un fichero de texto con nombre **package-info.java**. Este fichero estará en la misma carpeta que las clases de ese paquete. Cada paquete tendrá su propio fichero **package-info.java**.

```
/**
 * Descripción corta del paquete
 * <p>
 * Descripción larga del paquete
 *
 * @autor yo
 */
package paqueteQueQuieroComentar;
```

Contenido del fichero package-info.java

Cuando hemos creado los paquetes en el ejercicio anterior hemos marcado la casilla para que Eclipse generara los ficheros **package-info.java** de forma automática.

Si se nos ha olvidado marcar dicha casilla siempre podemos crear el fichero de forma "manual" desde al menú contextual del paquete (con la opción opción **New → File**).

Cuando se ejecuta la herramienta **Javadoc**, se genera de forma automática la documentación de las clases y métodos de la aplicación en formato **html** (y por lo tanto, visibles a través de un navegador). La documentación sólo se genera si hemos incluido comentarios javadoc en el código.

Teniendo en cuenta el proyecto **visorImágenes-L04**, tal y como ha quedado al final del ejercicio anterior, se pide:

- Crea comentarios JavaDoc para la nueva clase **pa.Demo**. Recuerda poner comentarios significativos, que sirvan de ayuda para que otro desarrollador sea capaz de usar correctamente las clases documentadas
- Crea comentarios JavaDoc para cada uno de los paquetes: **pa.interfaz**, **pa.filtros**, **pa.imagen** y **pa.menuFicheros**.
- Prueba a cambiar alguno de los comentarios de las clases **pa.interfaz.ImageViewer** y **pa.filtros.EdgeFilter**. Por ejemplo, tradúcelos, o añade alguna otra información.
- Genera la documentación JavaDoc. Para ello crearemos una nueva Run Configuration con el nombre **visor-L04-javadoc**. En este caso, en el campo Goals pondremos: **javadoc:3.6.0:javadoc**

La documentación Javadoc se genera en el directorio **target**, concretamente en la carpeta **target/site/apidocs**. Dicha documentación está formada por varios ficheros html. Abre el fichero **index.html** desde el menú contextual con la opción **Open With → Web Browser**. Si no se abre dicho fichero en el navegador revisa las preferencias generales de eclipse **General → Web Browser**, y asegúrate de tener marcado "Use external web browser" y también debes tener seleccionado el navegador.

Navega por la documentación generada, busca los comentarios JavaDoc que has creado y/o cambiado en los ficheros fuente y comprueba que se muestran de forma correcta.

Sube a GitHub el trabajo realizado hasta el momento:

```
> git add . //desde el directorio que contiene el repositorio git
> git commit -m"Terminado el ejercicio 2: comentarios javadoc, de L04"
> git push
```


04

Trabajando con Arrays (de tipos primitivos)



Vamos a usar el proyecto Maven **notas-L04** que os hemos proporcionado

Crea una nueva *Run Configuration* (de tipo *Maven build*) con el nombre **notas-L04-run** (usa las goals: **clean compile exec:java -Dexec.mainClass=<nombreClase>**). En donde **<nombreClase>** tendrás que sustituirlo por el nombre de la clase que contiene el método `main()`, que será la clase `Demo` y estará ubicada en el paquete `pa`. Recuerda que debes guardar esta *Run Configuration* en la carpeta `eclipse-launch-files` (dicha carpeta ya la tienes creada)

Indicamos los pasos a seguir:

- Crea una nueva clase **pa.NotasAsignatura**. Se trata de una clase que contiene las notas de una determinada asignatura y permite calcular la nota media, la máxima, la mínima, mostrar las notas y también un histograma de frecuencias de notas. A la derecha mostramos la representación abstracta de dicha clase.

Observa que en las firmas de los métodos NO hemos incluido los nombres de los parámetros, solamente hemos indicado su tipo. Y usaremos un método privado. Recuerda que NO podremos usarlo desde un código fuera de la clase.

Recuerda también que los campos siempre serán privados. En este caso, desde fuera de la clase, se puede consultar el valor de dichos campos, pero NO se pueden modificar. Si los campos son privados, podremos cambiar el nombre, o incluso el tipo de los atributos sin afectar al resto de clases.

NotasAsignatura
- nombreAsignatura: String
- notas: float []
+ NotasAsignatura(String, float[])
+ getNombreAsignatura(): String
+ getNotas: float[]
+ calcularValorMinimo(): float
+ calcularValorMaximo(): float
+ calcularMedia(): float
- calcularFrecuencias(): int[]
+ mostrarHistograma(int,int)
+ mostrarNotas_en_columnas(int)

Implementa la clase **pa.NotasAsignatura**. Detallamos qué es lo que debe hacer cada método:

- El **constructor** inicializa los atributos de la clase con los valores pasados por parámetro. El campo **nombreAsignatura** es una cadena de caracteres que representa el nombre de la asignatura. El campo **notas** es el conjunto de notas cuyos valores y pueden tener decimales. Asumiremos que las notas de los alumnos siempre van a ser valores comprendidos entre 0 y 10 (no hay que validar los valores de los parámetros).
- Los métodos **getNombreAsignatura()** y **getNotas()** son métodos *accessor*, para los campos **nombreAsignatura** y **notas**, respectivamente.
- Los métodos **calcularValorMinimo()**, y **calcularValorMaximo()**, devuelven el valor de la nota mínima y máxima, respectivamente, de la *colección de notas*.
- El método **calcularMedia()** calcula y devuelve la nota media de las notas contenidas en la colección de notas.
- El método **calcularFrecuencias()** calcula y devuelve un array con el número de notas comprendidas en varios intervalos de valores. Se contemplan 10 intervalos, de la siguiente forma:
 - En la posición 0 almacenamos el número de notas cuyo valor está comprendido entre 0 y 1 (pero sin incluir el 1).
 - En la posición 1 se guarda el número de notas cuyo valor está comprendido entre 1 y 2 (sin incluir el 2), y así sucesivamente
 - En la posición 10 guardamos el número de notas cuyo valor es 10

Por ejemplo, la nota 3,6 debe contabilizarse en la posición 3 del array de frecuencias, la nota 4,9 se contabilizará en la posición 4, y la nota 10 se contabiliza en la posición 10 del array. En el fichero de plantillas **frecuencias.txt** tenéis explicado cómo averiguar a qué intervalo pertenece cada nota, calculando su parte entera.

- El método **mostrarHistograma()** muestra por pantalla el histograma de frecuencias en un rango de intervalos pasado por parámetro (se pasan los valores iniciales y finales del rango). Se considera un rango válido si los valores iniciales y finales están comprendidos entre 0 y 10, y el valor inicial es menor o igual al valor final. Si los valores pasados por parámetro incumplen esta restricción, no se dibuja el histograma y se muestra por pantalla el mensaje "Rango de valores inválido".

Por ejemplo, si se invoca a **mostrarHistograma(3,6)**, solo mostraremos cuatro columnas. Cada una de ellas tendrá tantos asteriscos como notas estén comprendidas en los intervalos [3,4), [4,5), [5,6), [6,7) respectivamente

Para implementar este método debes fijarte en la salida por pantalla de la **Figura 2** (a partir de la línea "Histograma de notas de la asignatura").

En el fichero de plantillas histograma.txt

Cada una de las líneas del histograma llevará al principio un "tabulador". Para ello usaremos el carácter '\t' en una sentencia **System.out.println()**, **System.out.print()**, o **System.out.printf()**.

Este método requiere calcular los valores de las frecuencias de notas, antes de dibujar por pantalla el histograma, para lo que usaremos el método privado **calcularFrecuencias()**

- El método **mostrarNotas_en_columnas()** imprime por pantalla el listado de notas, en forma de tabla con un número de columnas indicado por parámetro. Cada fila llevará al inicio un tabulador. Las notas se imprimen por pantalla con un decimal. Y las columnas tienen una anchura de 6 caracteres. Para encolumnar las notas puedes usar la siguiente cadena de formato:

7,0	0,2	3,2
5,8	6,4	1,4
4,7	5,5	1,6
5,0	3,0	1,0
8,2	1,8	

`System.out.printf("%6.1f", dato)`

6 → ancho de columna
.1 → número de decimales
f → indica que el dato es de tipo float

\t 6 6 6

Si el valor pasado por parámetro es negativo se imprimirá por pantalla el mensaje: "Error: El número de columnas indicado debe ser mayor que cero".

- Crea una clase **pa.Demo** con un método **main()** para probar la aplicación de forma que se muestre en pantalla la información de la **Figura 2**.

NOTA: Recuerda que en rojo mostramos las salidas por pantalla de las sentencias **System.out.println()** que debes usar en el método **main()**.



DATOS DE ENTRADA: La colección de notas que debéis usar para realizar el ejercicio estará formada por los siguientes valores:

{7.0f, 0.2f, 3.2f, 5.8f, 6.4f, 1.4f, 4.7f, 5.5f, 1.6f, 5, 3, 1, 8.2f, 1.8f}

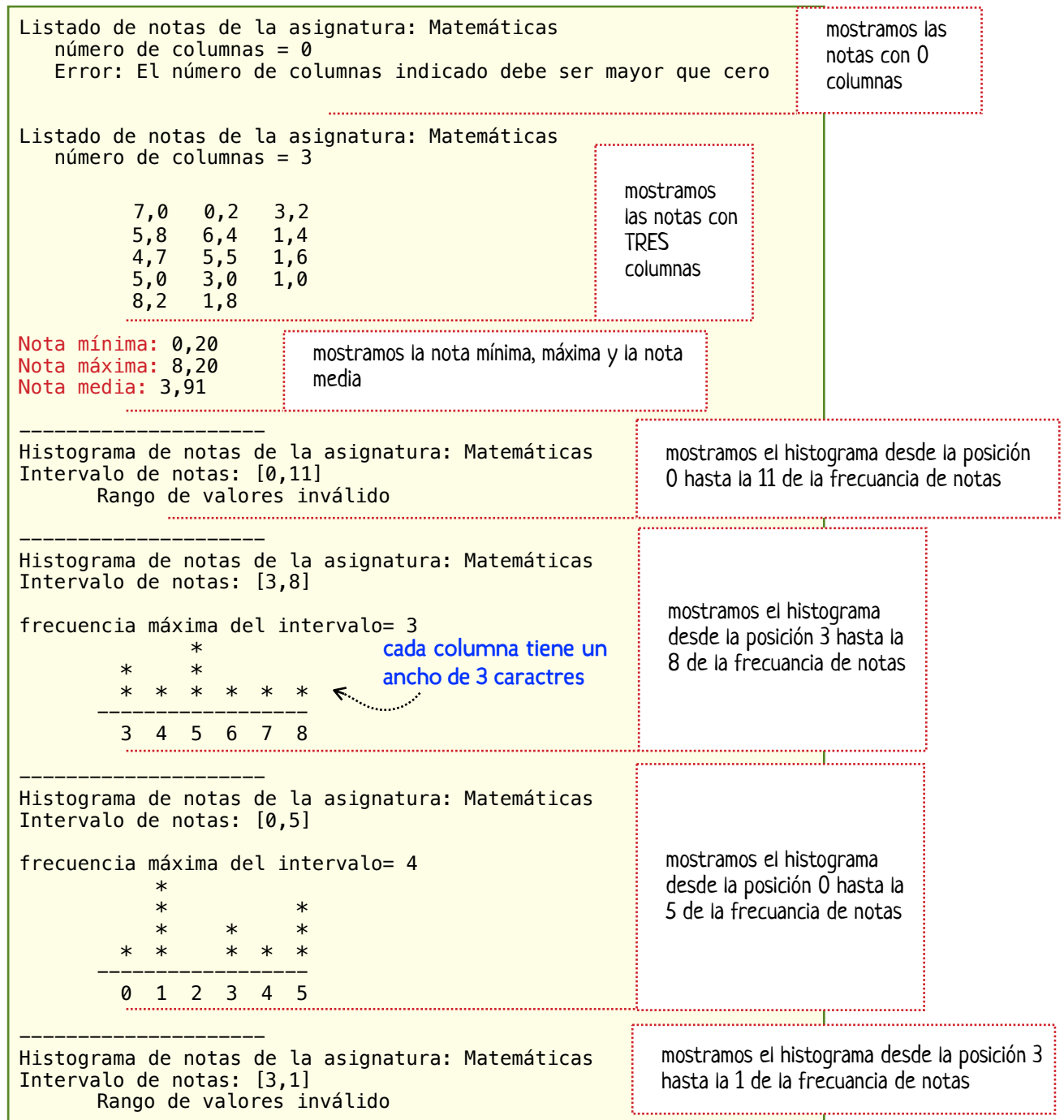


Figura 2. Salida por pantalla del método *main()*

Sube GitHub el trabajo realizado hasta el momento:

```
> git add . //desde el directorio que contiene el repositorio git
> git commit -m"Terminado el ejercicio 3 de L04: notas"
> git push
```




05

Trabajando con Arrays de Objetos



Vamos a usar el proyecto Maven **centroMedico-L04** que os hemos proporcionado.

Verás que el proyecto está formado por las clases *pa.Cita* y *pa.Medico*.

Lo primero que haremos será crear una nueva clase **Prueba** (esta clase NO pertenece a ningún paquete!!, debes seleccionar "default Package" en el campo *Package* al crear la clase) con un método **main()**. Esta clase será la que usaremos para probar nuestra aplicación.

Antes de seguir con el código vamos a crear las siguientes **Run Configurations** (en la carpeta *eclipse-launch-files*, que tendrás que crear previamente) de nuestro proyecto maven:

- **centroMedico-L04-run**: usa las goals que ya conoces para borrar el directorio target, compilar y ejecutar el método main de la clase Prueba (del paquete pa)
- **centroMedico-L04-javadoc**: genera la documentación javadoc del proyecto. Por defecto no se incluyen los comentarios javadoc para los métodos privados. Para incluir dichos comentarios en el documento html debes usar la goal: **javadoc:3.10.0:javadoc -Dshow=private**

IMPORTANTE: Recuerda que SIEMPRE tienes que lanzar la construcción del proyecto desde una *Run configuration* que hayas creado, o bien usar alguna de las que proporcionemos. No construyas el proyecto pulsando el triángulo verde de la barra de menú!!!

A continuación implementa los siguientes **métodos** de la clase **Medico**. Tienes indicada toda la información necesaria en los comentarios Javadoc de cada método:

- métodos privados **diaValido()** y **franjaValida()**.
- métodos privados **dia()** y **hora()**
- método público **reservarCita()**. Para implementar este método debes usar los métodos privados **diaValido()** y **franjaValida()**. Fíjate que uno de los parámetros de *reservarCita()* representa la franja horaria (de tipo String), A partir de dicho *String*, elegiremos el índice apropiado del atributo *horario* para cada una de las horas de dicha franja.
- método público **printHorario()**. Para implementar este método debes usar los métodos privados **dia()** y **hora()**

A continuación mostramos un ejemplo de horario:

Horario del doctor: 001		Especialidad:Oftalmólogo							
	9:00h	10:00h	11:00h	12:00h	16:00h	17:00h	18:00h	19:00h	
Lunes	---	---	---	---	---	---	---	---	
Martes	---	---	---	---	---	---	---	---	
Miércoles	Pedro	Maria	Lucas	Ana	---	---	---	---	
Jueves	---	---	---	---	---	---	---	---	
Viernes	---	---	---	---	Carlos	---	---	---	

"%-11s" "%-7s"

Puedes formatear la primera columna de forma que escribas ajustando el texto a la izquierda, con un ancho de 11 puntos ("%11s"). El signo '-' indica que la cadena de caracteres (s) se alineará a la izquierda.

El resto de columnas puedes formatearlas con un ancho de 7 puntos alineando el texto a la izquierda ("%7s")



Implementa el método **programa_principal.Prueba.main()**, de forma que:

- Creamos un objeto de tipo Medico, cuyo identificador sea "001" y su especialidad "Oftalmólogo"
- Imprimimos su horario.
- Mostramos el mensaje: "Iniciamos el proceso de reserva de Citas ..."
- A continuación solicitamos las citas indicadas en la siguiente tabla (todas para el médico creado):

Paciente:	Pedro	María	Lucas	Ana	Gloria	Matías	Eva	Carlos
Día a reservar:	2	2	2	2	2	3	8	4
Franja horaria:	mañana	mañana	mañana	mañana	mañana	otro	tarde	tarde

La fila "Día a reservar", contiene valores enteros. Cada valor representa el día de la semana en el que un determinado paciente quiere solicitar la cita (0 = lunes, ..., 4 = viernes)

Las filas "Paciente" y "Franja horaria" contienen valores de tipo String con el nombre de cada paciente y la franja horaria en la que quiere solicitar la cita.

Si para algún paciente no es posible concertar la cita, se imprimirá por pantalla el mensaje "Cita no posible para <paciente>" en donde <paciente> es el nombre el paciente cuya cita no puede concertarse.

Si la cita se crea con éxito, se mostrarán por pantalla los datos de la cita para ese paciente

- Indicamos el total de citas procesadas y el total de citas que no ha sido posible realizar (ver los mensajes en rojo en la salida por pantalla que incluimos al final del ejercicio.
- Finalmente imprimimos por pantalla de nuevo el horario del médico, mostrando todas sus citas.

A continuación mostramos la salida por pantalla resultante de ejecutar el método *Prueba.main()*.

Recuerda subir a GitHub el trabajo realizado hasta el momento:

```
> git add . //desde el directorio que contiene .git
> git commit -m"Terminado el ejercicio 4 de L04: centro médico"
> git push
```



Horario del doctor: 001 Especialidad:Oftalmólogo

	9:00h	10:00h	11:00h	12:00h	16:00h	17:00h	18:00h	19:00h
Lunes	---	---	---	---	---	---	---	---
Martes	---	---	---	---	---	---	---	---
Miércoles	---	---	---	---	---	---	---	---
Jueves	---	---	---	---	---	---	---	---
Viernes	---	---	---	---	---	---	---	---

Iniciamos el proceso de reserva de Citas ...

Recordatorio de cita
Paciente: Pedro
Dia consulta: Miércoles
Horario: 9:00h
Médico: 001
Especialidad: Oftalmólogo

En rojo mostramos los
System.out.println que
debéis usar en el main()

Recordatorio de cita
Paciente: Maria
Dia consulta: Miércoles
Horario: 10:00h
Médico: 001
Especialidad: Oftalmólogo

Debes usar un bucle para
procesar cada cita

Recordatorio de cita
Paciente: Lucas
Dia consulta: Miércoles
Horario: 11:00h
Médico: 001
Especialidad: Oftalmólogo

Recordatorio de cita
Paciente: Ana
Dia consulta: Miércoles
Horario: 12:00h
Médico: 001
Especialidad: Oftalmólogo

Los nombres Gloria,
Matias y Eva, debes
obtenerlos accediendo al
array de nombres

Cita no posible para Gloria
Cita no posible para Matias
Cita no posible para Eva

Recordatorio de cita
Paciente: Carlos
Dia consulta: Viernes
Horario: 16:00h
Médico: 001
Especialidad: Oftalmólogo

Los valores 8 y 3 debes
calcularlos en el main() y
mostrarlos en el mensaje

Total de citas procesadas: 8
Total de citas no posibles: 3

Horario del doctor: 001 Especialidad:Oftalmólogo

	9:00h	10:00h	11:00h	12:00h	16:00h	17:00h	18:00h	19:00h
Lunes	---	---	---	---	---	---	---	---
Martes	---	---	---	---	---	---	---	---
Miércoles	Pedro	Maria	Lucas	Ana	---	---	---	---
Jueves	---	---	---	---	---	---	---	---
Viernes	---	---	---	---	Carlos	---	---	---

06

Empaquetando nuestro proyecto Maven



Si abres los ficheros pom.xml de los tres proyectos maven de esta práctica, verás que el pom.xml de *visorImágenes* y *centroMedico* tienen más líneas.

Dichas líneas adicionales nos van a permitir "**empaquetar**" nuestro proyecto maven, de forma que tengamos en **un único fichero**, con extensión **.jar**, todos los ejecutables y cualquier otro fichero adicional que necesitemos, como por ejemplo, el conjunto de imágenes de la carpeta resources para el proyecto *visorImágenes*. Fíjate que en el fichero pom.xml hemos indicado qué clase es la que contiene el método main().

Podemos ejecutar un fichero con extensión **.jar** desde un terminal, y funcionará exactamente igual que haciéndolo desde eclipse, con la ventaja de que sólo tenemos que guardar un único fichero (y no necesitaremos el IDE).

Recuerda que nuestra aplicación puede estar formada por muchos ficheros .java, y es necesario que todos ellos estén "accesibles" para poder ejecutarlo (no nos puede faltar ninguno). En realidad, no es que nuestra aplicación "pueda" estar formada por varios ficheros, **ESTARÁ** formada por varios ficheros si hemos dividido nuestro problema en partes para poder solucionarlo más fácilmente.

Por eso, la posibilidad de empaquetar todo nuestro trabajo en un único fichero, evitará "perder" accidentalmente alguna clase, algún paquete, o cualquier otro fichero que se nos olvide copiar.

Para poder empaquetar las clases de nuestro proyecto, primero tendremos que **compilar** todos los ficheros java para así obtener los ejecutables (ficheros .class), y además, añadir, si es necesario, cualquier fichero adicional (como las imágenes del proyecto *visorImágenes*).

Afortunadamente, Maven nos permite hacer todo eso con un único comando:

```
mvn package
```

Dicho comando compila todos los ficheros .java (que están en src/main/java) y copia todo el contenido de src/main/resources en la carpeta target. Finalmente comprime dichos ficheros y los empaqueta en un único fichero con extensión **.jar**.

Por ejemplo, para la aplicación "centroMedico-L04", generará el fichero "*centroMedico-L04-1.0-SNAPSHOT.jar*" (Este nombre se puede cambiar indicándolo en el pom.xml pero no vamos a hacerlo).

Para ejecutar el fichero empaquetado simplemente tendremos que usar el comando:

```
java -jar <nombre_fichero.jar>
```

Vamos a **generar** los **.jar** de los proyectos *centroMedico* y *visorImágenes*. Para ello debes crear dos nuevas *Run Configurations* (cada una en el proyecto correspondiente):

- **centroMedico-L04-package**: en el proyecto *centroMedico-L04*. Debes usar las goals: **clean package**
- **visorImágenes-L04-package**: en el proyecto *visorimágenes-L04*. Debes usar las goals: **clean package**

Una vez que hayas creado las dos nuevas *Run Configurations*, ejecutalas. Verás que en el directorio target de cada proyecto aparecerán los ficheros **.jar**.

Ahora puedes cerrar Eclipse.

Abre un terminal y sitúate en el directorio *target* de *centroMedico-L04*. Para ejecutar la aplicación usa el comando:

```
java -jar centroMedico-L04-1.0-SNAPSHOT.jar
```

Nota: No se te ocurra teclear todo ese nombre de fichero carácter a carácter!!! Recuerda que simplemente tendrás que teclear "java -jar ce" y pulsar el **TABULADOR** para autocompletar el nombre del fichero.

De la misma forma ejecuta el jar generado para el proyecto visorImágenes desde el terminal.


Puedes copiar los ficheros .jar en cualquier carpeta y ejecutarlos sin problema (o copiar el .jar en un usb para ejecutarlo en otro ordenador), sin necesidad de abrir Eclipse. Es una gran ventaja, ¿no te parece?

Dado que el fichero .jar está dentro de *target*, dicho fichero no se guardará en GitHub. Pero no es problema porque podemos volver a generarlo en cualquier momento a partir de los fuentes y el fichero pom.xml.

Finalmente puedes ver el contenido e incluso descomprimir cada uno de los ficheros *jar* con los comandos:

jar -tvf <nombre_fichero_jar> :muestra el contenido del fichero .jar

jar -xvf <nombre_fichero_jar> :extrae los ficheros empaquetados en el fichero .jar

Por último, también podríamos abrir un nuevo "Terminal" desde Eclipse y hacer lo mismo sin salir del IDE. Primero mostramos la **vista Terminal**: desde *Window→Show View→Terminal* (Si no te aparece búscalo en *Others→Terminal→Terminal*). Y en dicha vista seleccionamos el icono  cada vez que quieras abrir un nuevo terminal (debes crear un "Local terminal").

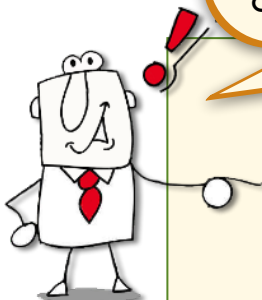
Si queremos **construir** nuestro proyecto maven desde un Terminal también podemos hacerlo sin necesidad de usar el ide. Simplemente ejecutamos el comando **mvn**, seguido de la goal de la correspondiente *Run Configuration* que queramos ejecutar. Debes estar situado en el directorio que contiene el fichero *pom.xml* del proyecto maven que quieras construir. Por ejemplo:

mvn compile exec:java -Dexec.mainClass=pa.Demo

Y obtendríamos exactamente el mismo resultado que si ejecutamos la Run Configuration "visor-L04-Demo-run" desde Eclipse.

RECUERDA!

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



- Cómo hemos modularizado cada uno de los proyectos de prácticas (por niveles de abstracción)
- Qué es un paquete en java y para qué sirve
- Qué es un comentario javaDoc y para qué sirve
- Cómo puedo generar una documentación JavaDoc a partir de mi código fuente.
- Cuándo debo usar un array en mi programa
- Cómo puedo declarar e inicializar un array de objetos
- Cómo puedo recorrer la colección de elementos del array usando diferentes tipos de bucles
- Cuándo puedo y/o debo usar cada tipo de bucle
- ¿Para qué sirve un fichero jar, qué contiene, y cómo puedo generarlo?