Programación para Ingeniería Telemática

Curso 2018-19

Práctica 8: java.util.concurrent.

1. Objetivos de aprendizaje. Estructura de la práctica.

Objetivos

- Repasar conceptos de orientación a objetos de prácticas anteriores: herencia, *downcasting*, iteradores, genéricos, factorías y manejadores de eventos.
- Utilizar la biblioteca `java.util.concurrent` para proporcionar servicios asíncronos de carga de ficheros.

En esta práctica repasaremos los contenidos de temas anteriores modificando y extendiendo una aplicación para incorporar en ella servicios asíncronos.

Contenidos de la memoria

Esta memoria se estructura en cinco secciones con los siguientes contenidos:

Sección 1: Objetivos y contenido de la práctica.

Sección 2: Los paquetes `common`, `game`, `views` y **`async`.** En esta sección se presenta el código de partida que tendrá que usarse para realizar esta práctica en concreto. Este código es básicamente el proporciona en la práctica 7, pero con algunas simplificaciones y adiciones que se explican en esta sección.

Sección 3: Enunciado de los ejercicios a realizar.

Los ejemplos y ejercicios que se vayan mostrando en las prácticas y o bien son semejantes a los que pueden pedirse en los exámenes de la asignatura o bien constituyen un paso previo para ser capaces de realizarlos.

Antes de hacer los ejercicios es aconsejable tener una visión de conjunto, por ello **se recomienda leer detenidamente el boletín de prácticas y entender el código suministrado antes de abordar la resolución de los ejercicios.**

2. Los paquetes `common`, `game`, `views` y `async`.

2.1. El paquete `common`

Se ha modificado y simplificado el fichero FileUtilities, de forma que sólo contiene los métodos usados en la práctica para guardar juegos en formato JSON y leerlos.

En la práctica NO tendrán que usar directamente los métodos de `FileUtilities`. Sin embargo, es conveniente saber cómo funciona el método de lectura, ya que introduce un retardo artificial.

Método de lectura de ficheros:

```
public static ArrayList<String> readFile(String fileName)
                                        throws FileNotFoundException
```

- Devuelve una lista formada por la representación JSON de cada *frame* del juego.
- Cada frame está a su vez constituido por los elementos del juego existentes en un tick de reloj.
- Al final de la lectura de cada *frame*, `readFile` introduce un retardo de 100 ms para simular el efecto que se produce cuando el servicio solicitado requiere un tiempo significativo.

2.2. El paquete `game`

Las interfaces y clases de esta práctica (figura 1) son parecidas a las de la práctica 7, con algunas modificaciones y adiciones (resaltadas en la figura) que se comentan a continuación.

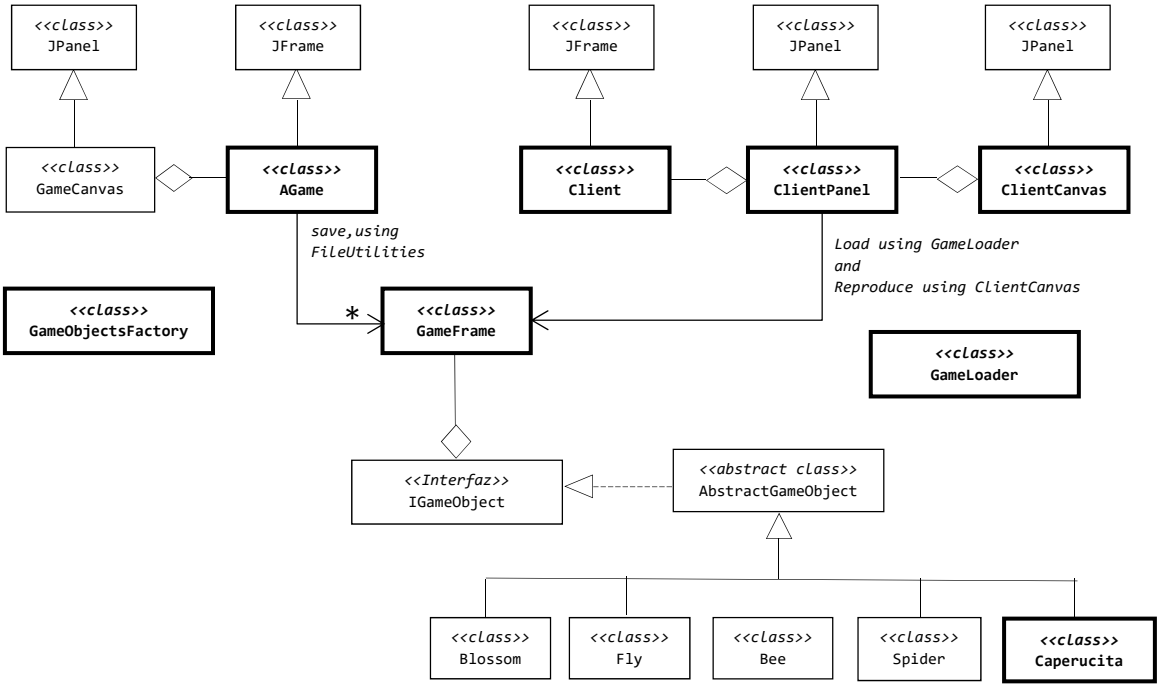


Figura 1: Tipos de datos

- La clase **`AGame`** es un juego sencillo que sólo funciona en forma automática y cuyo cometido es generar ficheros que representan juegos.

  Para la realización de la práctica NO tendrán que modificar este fichero, pero podrá utilizarlo para generar nuevos ficheros con juegos. Para ello, ejecute el fichero `AGame`. Cada vez que pulse la barra espaciadora se guardará la anterior partida y se iniciará una nueva. Los ficheros generados se guardan en el directorio `"src/main/resources/games"` con el nombre `g_#`, donde # es un número que se reinicia cada vez que se rearranca el programa. Cada vez que rearranque `AGame` sobreescribirá los ficheros anteriores.

  La carpeta `games`, proporcionada junto al resto de paquetes, contiene juegos previamente generados con `AGame`, si bien con nombres que siguen otra pauta (`p1`, `p2`, ...) con objeto de que no los pierda al ejecutar el programa sucesivas veces. Debe copiar esta carpeta en el directorio del proyecto con la siguiente ruta: *`proyecto`*`/src/main/resources/`

- La clase **`Caperucita`** es una implementación sencilla del personaje principal del juego que puede moverse de forma automática.
- La clase **`GameObjectsFactory`** es una factoría de objetos del juego.

  Para la realización de la práctica NO tendrán que modificar este fichero, pero podrá pedirse implementar y/o usar una factoría semejante en el examen final (las factorías ya se vieron en prácticas anteriores).

- La clase **`GameFrame`** guarda una lista de **`IGameObjects`**. El programa **`AGame`** genera un *frame* en cada tick de reloj que contiene los elementos del juego existentes en ese tick. El conjunto de todos los *game frames* de un juego se guarda posteriormente en un fichero cuando acaba la partida.

  Para la realización de la práctica NO tendrán que modificar este fichero, pero podrá pedirse implementar y/o modificar una clase semejante en el examen final (las listas de objetos, los iteradores y la serialización/deserialización a/desde JSON ya se vieron en prácticas anteriores).

- La clase **`Client`** es una ventana desde la que se puede pedir la carga y reproducción de juegos. Incluye cuatro paneles que representan a cuatro clientes, como puede verse en la figura 2. Todos los clientes son instancias de la clase `ClientPanel` y por consiguiente todos se comportan igual. Por ello, explicaremos su funcionamiento en el punto dedicado a la clase `ClientPanel`.

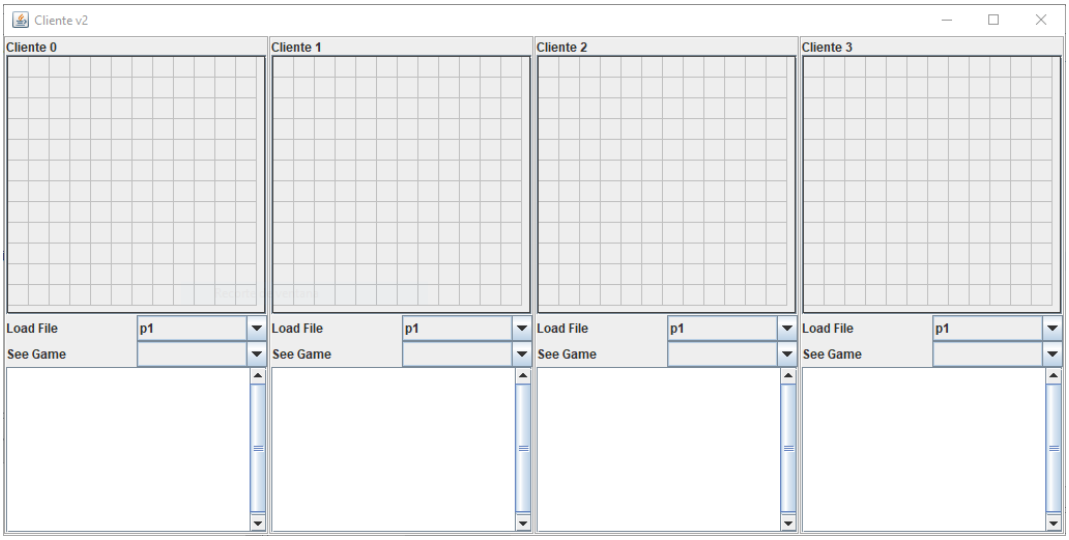


Figura 2: La aplicación `Client`

- La clase **`ClientPanel`** es un panel que proporciona acceso a servicios de (1) carga de ficheros previamente guardados y (2) reproducción de ficheros previamente cargados. También muestra información sobre el estado de los servicios solicitados. La figura 3 muestra esquemáticamente los elementos de `ClientPanel` y su funcionamiento.

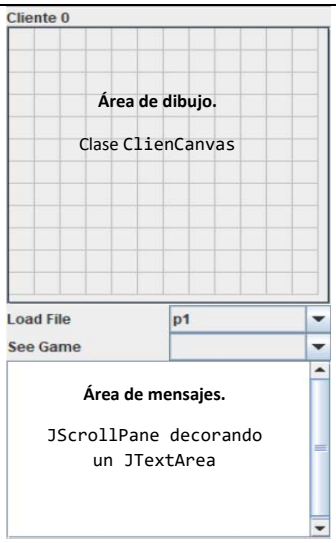
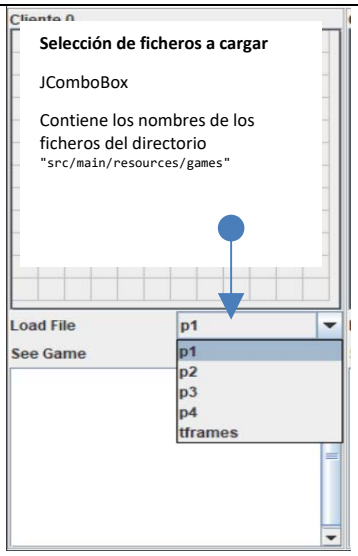
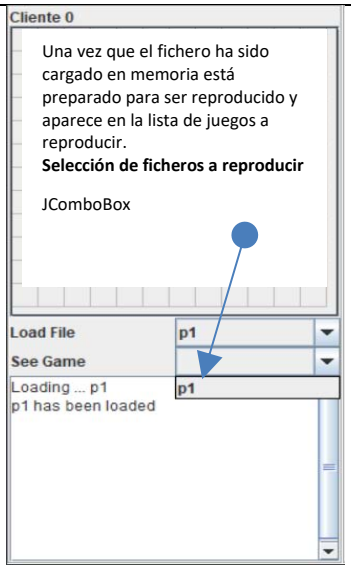
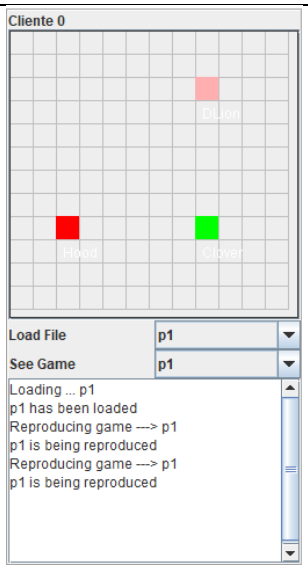
<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>3.a) Cada panel de cliente se estructura en tres partes: (1) área de dibujo, (2) área de mensajes y (3) área de controles.</td>
<td>3.b) El área de controles tiene dos listas: (1) lista de juegos guardados y que pueden cargarse en memoria y (2) lista de juegos cargados en memoria listos para reproducirse.</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>3.c) Cada vez que se carga un nuevo juego se añade a la lista de reproducibles. La carga de un juego no es instantánea, debido al retardo que introduce la lectura del fichero. ***Mientras se carga un fichero todos los clientes se quedan bloqueados.***</td>
<td>3.d) Se pueden reproducir los ficheros de la lista de reproducibles. ***La reproducción de los ficheros no produce el bloqueo*** de la aplicación, ya que se realiza en un hilo aparte controlado por la máquina virtual de Java.</td>
</tr>
</table>

Figura 3: Estructura de `ClientPanel`

Cuando ejecute `Client` podrá observar que la carga de los ficheros produce el bloqueo de la aplicación. Sin embargo la reproducción de los ficheros no es bloqueante.

Uno de los ejercicios de la práctica será proporcionar una nueva versión de esta clase (`AsyncClientPanel`).

- La clase **`ClientCanvas`** es una versión ligeramente modificada de `GameCanvas`, que proporciona una superficie de dibujo a un `ClientPanel` donde reproducir los juegos previamente cargados. `ClientCanvas` proporciona el método `playMovie` para reproducir un juego:

  ```
  void playMovie(ArrayList<GameFrame> movie)
  ```

  ***Desde el punto de vista de* `ClientPanel`, `ClientCanvas` *es un servidor que le proporciona el servicio* `playMovie`.**

  La reproducción de gráficos consume muchos recursos computacionales y normalmente el entorno de trabajo (en este caso la máquina virtual de java y la biblioteca swing) no deja hacerla de forma directa, sino solicitarla.

  En realidad `playMovie` lo único que hace es encolar los *frames* a reproducir. El proceso de reproducción puede resumirse como sigue (los detalles pueden verse en el código fuente):

  – `ClientCanvas` mantiene una cola interna de *frames*.
  – Cada vez que se invoca `playMovie` se añaden a dicha cola los frames incluidos en el argumento.
  – Periódicamente, se extrae un *frame* de la cola y se solicita su reproducción.

  Uno de los ejercicios de la práctica será proporcionar una nueva versión de esta clase (`ClientCanvasScheduled`).

- La clase **`GameLoader`** es un decorador de FileUtilities, pensado para ocultar al programador los detalles de deserialización de los objetos guardados en formato JSON. Así, el método readFile de `FileUtilities`:

  ```
  static ArrayList<String> readFile(String fileName)
  ```

  ha sido sustituido (decorado) por el método

  ```
  ArrayList<GameFrame> loadFramesFromFile()
  ```

  Las diferencias entre ambas clases (`FileUtilities` y `GameLoader`) y ambos métodos (`readFile` y `loadFrameFromFiles`) van más allá de la simple ocultación de complejidad (deserialización) que ofrece `GameLoader`. Obsérvese:

  – El método `readFile` es estático, mientras que `loadFrameFromFile` no es estático.
  – `loadFrameFromFile` no tiene argumentos. El objeto `GameLoader` obtiene el fichero a cargar en su constructor.

  Las instancias de `GameLoader` podrían ser encoladas en un servidor y servidas más adelante de forma síncrona o asíncrona (no se hace así en esta práctica, pero podría hacerse).

  ***Desde el punto de vista de* `ClientPanel`, *los objetos de* `GameLoader` *son servidores que le proporcionan el servicio* `loadFramesFromFile`.** El servicio se obtendría, por ejemplo, de la siguiente manera (invocación síncrona):

  ```
  ArrayList<GameFrame> movie = new GameLoader(file).loadFramesFromFile();
  ```

  Es decir, primero obtenemos (creamos) el servidor pasándole el fichero a cargar y después solicitamos el servicio.

  Uno de los ejercicios de la práctica será proporcionar una nueva versión de esta clase (`GameLoadTask`).

2.3. El paquete `views`

El paquete `views` ha sido reducido con objeto de contener sólo vistas esquemáticas. No obstante, los estudiantes pueden ampliar este paquete y utilizar las vistas que deseen (no sería evaluable).

2.4. El paquete `async`

El paquete `async` contiene los ficheros de partida para la realización de la práctica. Todos ellos, salvo una interfaz, son versiones de alguno de los ya presentados. Los detalles se explicarán en el siguiente apartado (Ejercicios a realizar).

En términos generales, se trata de sustituir el servicio síncrono (y por tanto bloqueante) de carga de ficheros por otro asíncrono, utilizando ejecutores proporcionados por el paquete `java.util.concurrent`. El funcionamiento sería:

- Se solicita la carga de un fichero desde la interfaz gráfica.
- Se crea una tarea encargada de realizar tal carga de forma asíncrona, de forma que el cliente puede seguir con otras tareas o solicitar otros servicios sin verse bloqueado.
- Se asigna la tarea a un ejecutor.
- Cuando el servicio (la tarea) termina se recoge el resultado (juego cargado en memoria) y se pone a disposición del usuario para su reproducción.

Uno de los retos a resolver será cómo notificar al cliente que el resultado ya está listo. Aunque a partir de Java 8 `java.util.concurrent` proporciona una forma relativamente sencilla de realizar esta notificación, en esta práctica desarrollaremos una solución más artesanal para evitar entrar en los detalles de `java.util.concurrent`. Esta solución nos permitirá además profundizar en el patrón observador (*callbacks*) y observar más directamente algunas de las dificultades de la programación concurrente. Los contenidos de este paquete son los siguientes:

- La interfaz **`IAsyncLoaderObserver`** define un único método:

  ```
  void loadComplete(String key)
  ```

  Este método es invocado por la tarea que realiza el servicio al terminar la carga del fichero y se ejecuta en el cliente que solicitó el servicio. Por ello, la implementación debe retornar lo antes posible y no contener llamadas bloqueantes. En caso contrario puede bloquear a la tarea invocante. Obsérvese que la relación que hay entre la tarea que realiza el servicio y el cliente que solicita el servicio es análoga a la que hay entre un botón y su escuchador. Aquí el evento se corresponde con el final de la tarea y el manejador del evento sería el método `loadComplete` del cliente.

- La clase **`AsyncClient`** es una versión de `Client` que usa un `AsyncClientPanel` y que puede fijar el tipo de ejecutor que este usa para ejecutar las tareas.

- La clase **`AsyncClientPanel`** es una versión de `ClientPanel` que solicita los servicios de carga de ficheros de forma asíncrona usando un ejecutor.

- La clase **`GameLoadTask`** extiende `GameLoader` para proporcionar servicios asíncronos de carga de ficheros.

- La clase **`ClientCanvasScheduled`** es una versión de `ClientCanvas` que sustituye el temporizador por una tarea periódica.

3. Ejercicios a realizar

Antes de realizar los ejercicios, es conveniente que eche un vistazo al código, especialmente a las variables de instancia (no gráficas) de `AsyncClientPanel`.

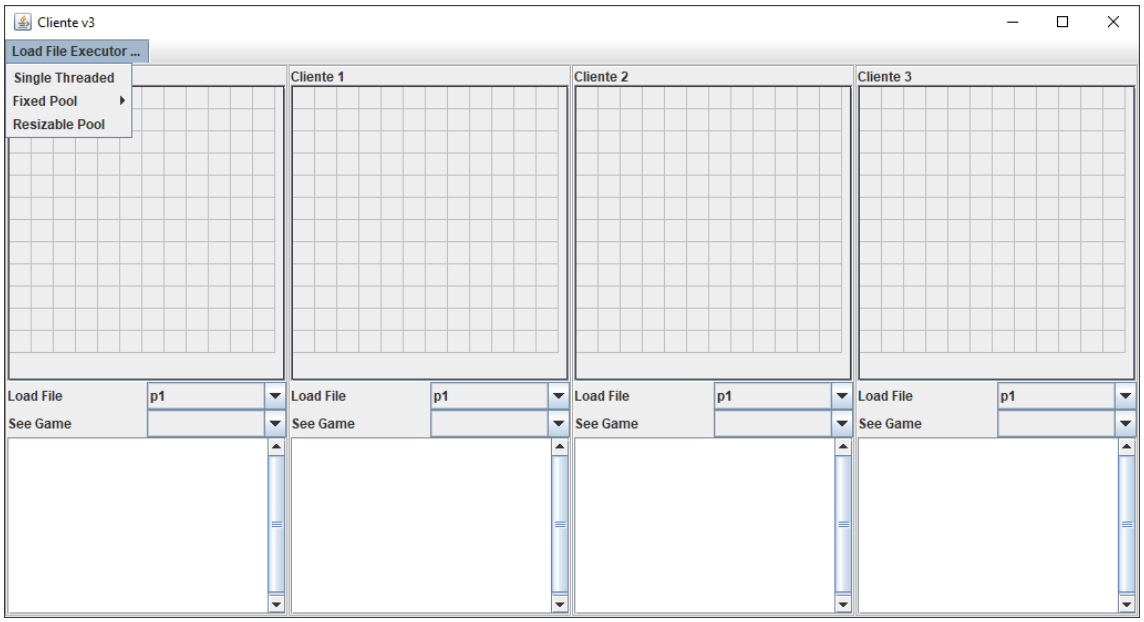
Ejercicio 1:

Complete la implementación de `AsynClient` de forma que se pueda escoger el cargador de ficheros, tal y como se muestra en la figura.

En función de la opción escogida se invocará el método `setLoader` de los `AsyncClientPanel` con el argumento adecuado. Debe asignarse el mismo (y único) ejecutor a todos los paneles clientes. En el caso de escogerse la entrada *Resizable Pool* se creará un `CachedThreadPool`.

El ejecutor a asignar por defecto será un `SingleThreadExecutor`.

No es necesario en este ejercicio que los `AsyncClientPanel` funcionen, basta con asegurarse de que obtienen el ejecutor adecuado.



Ejercicio 2:

Para poder probar este ejercicio tendrá que hacer los dos apartados, ya que uno no funciona sin el otro.

Apartado 1:

1) En AsyncClientPanel complete el manejador de eventos de cbFilesToLoad para solicitar al ejecutor (fileLoader) que ejecute una tarea GameLoadTask y devuelva un futuro asociado a la misma.
2) Guarde el futuro obtenido en la tabla pendingRequest, asociándole como clave el nombre del fichero solicitado.

Apartado 2: Complete la implementación de `GameLoadTask` de forma que el método `call` obtenga el fichero pedido en el constructor e invoque el método `loadComplete` del cliente justo antes de acabar.

Ejercicio 3:

En `AsyncClientPanel` complete el manejador de eventos de `cbReadyToPlay` para terminar la recogida de datos descargados en manejador de `cbFilesToUpload`.

Para ello, (1) inspeccione el código de loadComplete y (2) siga las indicaciones de los comentarios del manejador de eventos de `cbReadyToPlay`. Los pasos 1 y 2 se dan ya resueltos.

Si ha llegado hasta aquí podrá cargar de forma asíncrona los juegos y reproducirlos seleccionándolos en cbReadyToPlay.

Ejercicio 4:

A partir del código de `ClientCanvas` defina una nueva clase `ClientCanvasScheduled` que en lugar de un temporizador utilice una tarea periódica para la reproducción de los juegos y úsela en los paneles.