



PRÁCTICA 8. PROCESAMIENTO EN SEGUNDO PLANO

Interfaces Persona Computador


Depto. Sistemas Informáticos y Computación

UPV

Índice

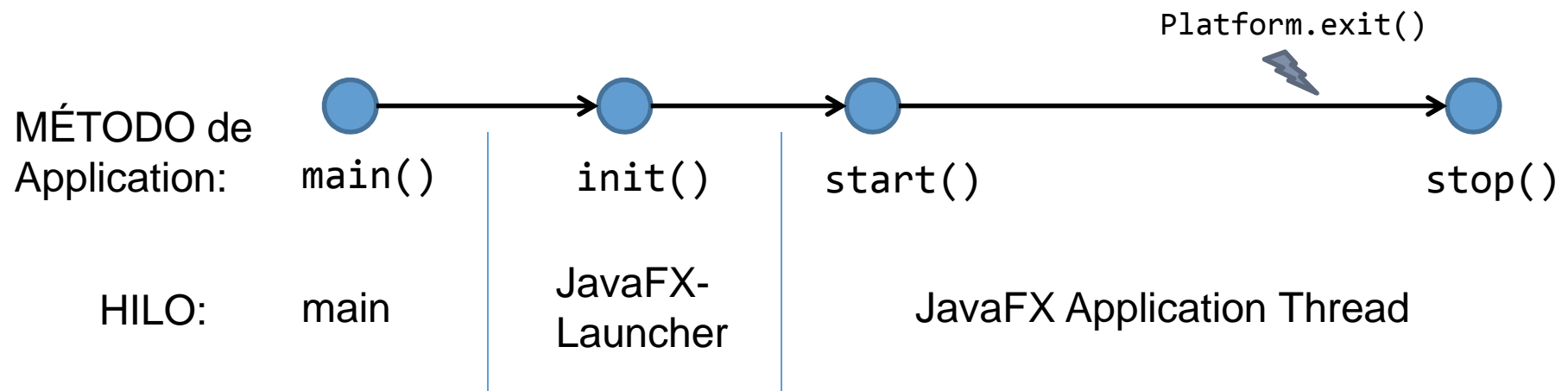
- Introducción
- Hilos y nodos
- Estados de las tareas
- La clase Task
- La clase WorkerStateEvent
- La clase Service
- Cambiando el cursor
- Herramientas útiles
- Ejercicio
- Bibliografía

Introducción

- Si has intentado hacer una tarea costosa en tiempo en un manejador de evento de JavaFX (por ejemplo, abrir un fichero grande, o bajar un fichero de Internet), te habrás dado cuenta que la interfaz se queda “congelada”
 - Los manejadores de eventos no deberían realizar tareas pesadas
- La forma adecuada de realizar tareas que pueden necesitar un tiempo para su realización es:
 - Indicar al usuario de alguna forma la duración de la tarea (p.e., una barra de progreso o, al menos, un cursor de espera) 
 - Lanzar la tarea en otro hilo
 - Cuando la tarea acabe, actualizar la vista de la escena

Hilos en JavaFX

- La mayor parte del tiempo, las aplicaciones JavaFX se ejecutan en el JavaFX Application thread, pero hay otros hilos:



Hilos y nodos

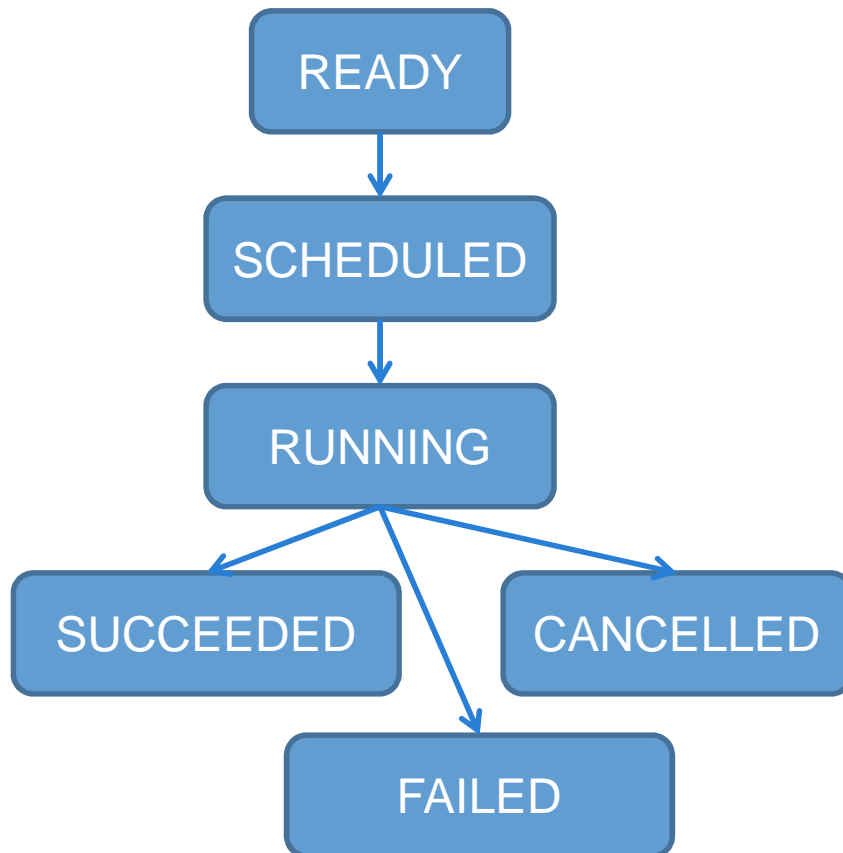
- JavaFX ofrece una serie de clases para facilitar la creación de hilos trabajadores y su sincronización con el hilo de la GUI
 - En el paquete `javafx.concurrent`, que se apoya y es compatible con el estándar `java.util.concurrent`
 - `Worker` (interface): especifica la API que debe tener una tarea que se ejecutará en un hilo trabajador y se comunicará con el hilo de JavaFX
 - `Task` (clase): contiene la lógica que se ejecutará en el hilo trabajador, y métodos para comunicarse con el hilo de JavaFX
 - `Service` (clase): se encarga de ejecutar tareas
 - `WorkerStateEvent` (clase): representa un evento que se produce cuando una tarea cambia de estado

Hilos y nodos

- Se pueden construir y modificar nodos en cualquier hilo, siempre que no se estén usando en la escena de una ventana visible
 - Sólo se puede modificar el grafo de escena de una ventana visible en el JavaFX Application Thread

Estado de las tareas

- El siguiente diagrama muestra el ciclo de vida de una tarea:



Propiedades principales de Task
(ReadOnly<T>Property):

- Double totalWork, workDone
- Double progress (-1, 0..1)
- Boolean running
- Object<Worker.State> state
- Object<V> value
- Throwable exception
- String message, title

La clase Task

- Usaremos esta clase para implementar el código que se ejecutará en un hilo trabajador:
 - Derivar una clase de Task
 - Sobrescribir el método `call`, con la lógica a ejecutar devolviendo el resultado. Dentro de este método:
 - NO se puede manipular el grafo de escena
 - Se puede llamar a los métodos `updateProgress`, `updateMessage` y `updateTitle` para informar a JavaFX del estado de ejecución
 - Comprueba regularmente si se ha cancelado la tarea (`isCancelled()`) y, en ese caso, terminar la ejecución inmediatamente
- Los objetos Task no son reutilizables (debes lanzar uno nuevo cada vez)

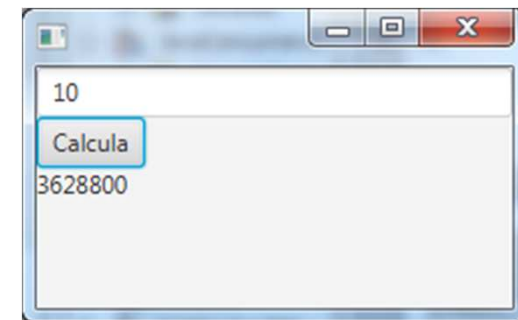
La clase Task

- Ejemplo:

```
import javafx.concurrent.Task;
Task<Long> task = new Task<Long>() {
    @Override
    protected Long call() throws Exception {
        long f = 1;
        for (long i = 2; i <= calculaFactorial; i++) {
            if (isCancelled()) {
                break;
            }
            f = f * i;
        }
        return f;
    }
};
```

```
@Override public void start(Stage primaryStage) {  
    TextField num = new TextField();  
    Label res = new Label();  
    Button btn = new Button("Calcula factorial");  
    btn.setOnAction(new EventHandler<ActionEvent>() {  
        @Override public void handle(ActionEvent event) {  
            final long calculaFactorial = Long.parseLong(num.getText());  
            // Aquí va el código de la traspas anterior  
            res.textProperty().bind(Bindings.convert(task.valueProperty()));  
            Thread th = new Thread(task);  
            th.setDaemon(true);  
            th.start();  
        }  
    });  
    VBox root = new VBox();  
    root.getChildren().addAll(num, btn, res);  
    Scene scene = new Scene(root, 300, 250);  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}
```

Lanzando la tarea



Esperas dentro de la tarea

- Si se llama a `Thread.sleep` u otro método bloqueante dentro de la tarea y el usuario cancela la tarea, se genera una `InterruptedException`. Hay que comprobar la cancelación de nuevo:

```
Task<Long> task = new Task<Long>() {
    @Override protected Long call() throws Exception {
        long f = 1;
        for (long i = 2; i <= calculaFactorial; i++) {
            if (isCancelled()) {
                break;
            }
            f = f * i;
            try { Thread.sleep(100); }
            catch (InterruptedException e) { if (isCancelled()) break; }
        }
        return f;
    }
};
```

La clase Task

- Se puede usar la propiedad `runningProperty` para ocultar o mostrar elementos del interfaz mientras la tarea se ejecuta:

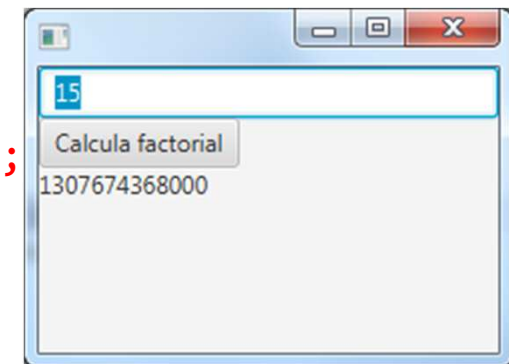
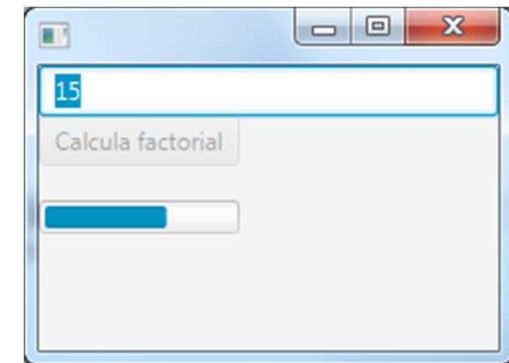
```
Label res;  
Button btn;  
// La etiqueta mostrará el resultado  
res.textProperty().bind(Bindings.convert(task.valueProperty()));  
// Pero no será visible mientras que se esté ejecutando la tarea  
res.visibleProperty().bind(Bindings.not(task.runningProperty()));  
// Además, deshabilitamos el botón mientras que haya una tarea activa  
btn.disableProperty().bind(task.runningProperty());
```

Mostrando el progreso

- En cualquier tarea que consuma un tiempo (a partir de 1 o 2 s) es conveniente hacer saber al usuario que se está ejecutando:

```
ProgressBar bar = new ProgressBar(0.0);
```

```
Task<Long> task = new Task<Long>() {  
    @Override protected Long call() throws Exception {  
        long f = 1;  
        for (long i = 2; i <= calculaFactorial; i++) {  
            [...]  
            updateProgress(i, calculaFactorial);  
        }  
        return f;  
    }  
};  
bar.progressProperty().bind(task.progressProperty());  
bar.visibleProperty().bind(task.runningProperty());  
[...]  
root.getChildren().addAll(num, btn, res, bar);
```



La clase WorkerStateEvent

- En cada cambio de estado, la clase que implementa Worker genera un evento distinto. Cómo usarlos:

- Desde fuera de Task

```
Label status = new Label();
task.setOnRunning(new
    EventHandler<WorkerStateEvent>() {
        @Override
        public void handle(WorkerStateEvent event) {
            status.setText("Calculando...");
        }
    });
task.setOnSucceeded(new
    EventHandler<WorkerStateEvent>() {
        @Override
        public void handle(WorkerStateEvent event) {
            status.setText("Terminado");
        }
    });
```

- Usando los métodos de ayuda de Task

```
Task<Long> task = new Task<Long>() {
    @Override protected Long call() {
        [...]
    }
    @Override protected void running() {
        super.running();
        updateMessage("Calculando...");
    }
    @Override protected void succeeded() {
        super.succeeded();
        updateMessage("Terminado");
    }
}
status.textProperty()
    .bind(task.messageProperty());
```

Ejecutar código en el hilo de JavaFX

- Hay ocasiones que podemos querer modificar el estado de JavaFX desde otro hilo. No podemos hacer la modificación directamente, pero podemos usar la función:
 - `javafx.application.Platform.runLater(Runnable runnable)`
- Dicha función ejecuta el `runnable` en el hilo de JavaFX en algún momento futuro
- Por ejemplo:

```
Platform.runLater(new Runnable() {  
    @Override public void run() {  
        customer.setFirstName(rs.getString("FirstName"));  
        // etc  
    }  
});
```

La clase Service

- La clase Service también implementa la interfaz Worker
- A diferencia de Task, un objeto Service se puede reutilizar (iniciar, parar, volver a iniciar, etc.)
 - Aunque internamente lo que hace es crear un nuevo Task cada vez
- La clase Service es de más alto nivel que Task, y se encarga de la creación de Threads mediante Executors
- La clase ScheduledService se encarga de reiniciar una tarea en cuanto se acaba, para implementar tareas repetitivas
- Más información en la documentación de JavaFX

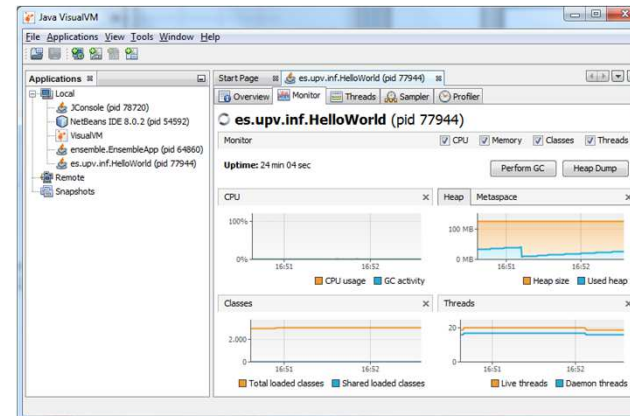
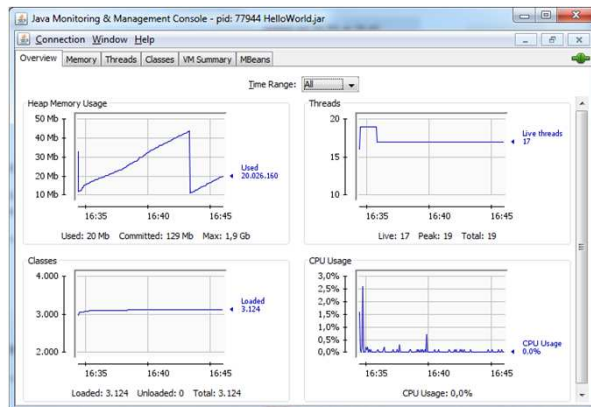
Cambiando el cursor

- Otra acción habitual al lanzar una tarea larga es cambiar el cursor a uno del tipo espera:

```
final Scene _scene = scene;
@Override
protected Long call() throws Exception {
    Platform.runLater(new Runnable() {
        @Override public void run() {
            _scene.setCursor(Cursor.WAIT);
        }});
    long f = 1;
    for (long i = 2; i <= calculaFactorial; i++) {
        if (isCancelled()) {
            break;
        }
        f = f * i;
    }
    Platform.runLater(new Runnable() {
        @Override public void run() {
            _scene.setCursor(Cursor.DEFAULT);
        }});
    return f;
}
```

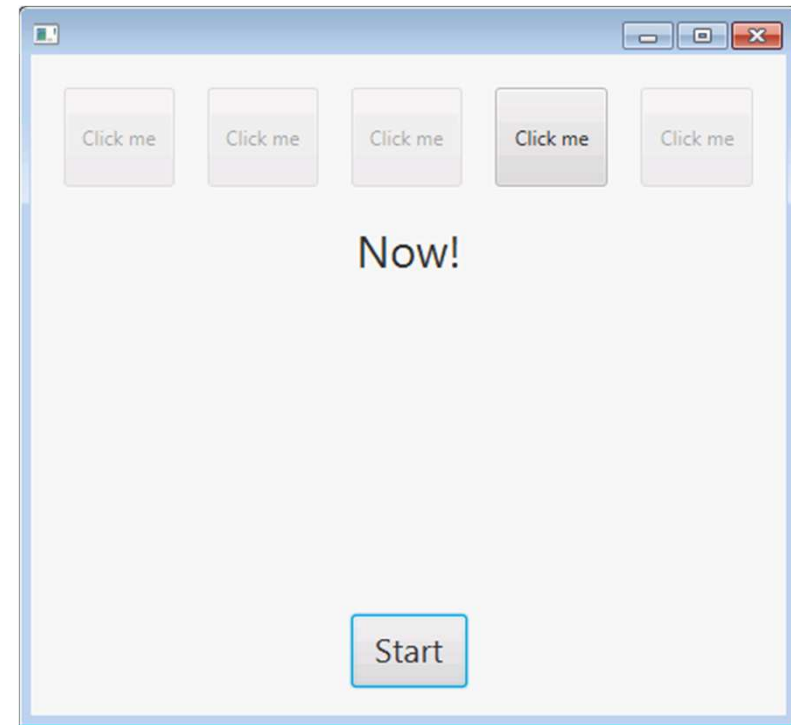
Herramientas útiles

- Las siguientes herramientas del JDK te pueden resultar útiles para estudiar el estado de un programa Java
 - jconsole: muestra en tiempo real información sobre aplicaciones Java en ejecución
 - jps: muestra en consola la lista de aplicaciones Java en ejecución, con su identificador
 - jstack: muestra la pila de ejecución de un programa Java
 - jvisualvm: como jconsole, pero con más opciones



Ejercicio

- Se ha implementado un juego para medir la velocidad de reflejos del usuario.
- Inicialmente, los botones de arriba están deshabilitados. Al pulsar el botón *Start*, pasará un tiempo aleatorio entre 1 y 6 segundos, y se habilitará un botón al azar
- Entonces se medirá el tiempo transcurrido desde que se habilita el botón hasta que el usuario lo pulsa
- El programador ha introducido todo el código en la *Java Application Thread*, por lo que no funciona. Arréglalo.



Bibliografía

- [https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Task.html](https://docs.oracle.com/javase/8/javafx/api/javafx/ concurrent/Task.html)
- <https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm>