




PRÁCTICA 6. PROCESAMIENTO EN SEGUNDO PLANO

Interfaces Persona
Computador
Depto. Sistemas Informáticos
y Computación
UPV

Índice

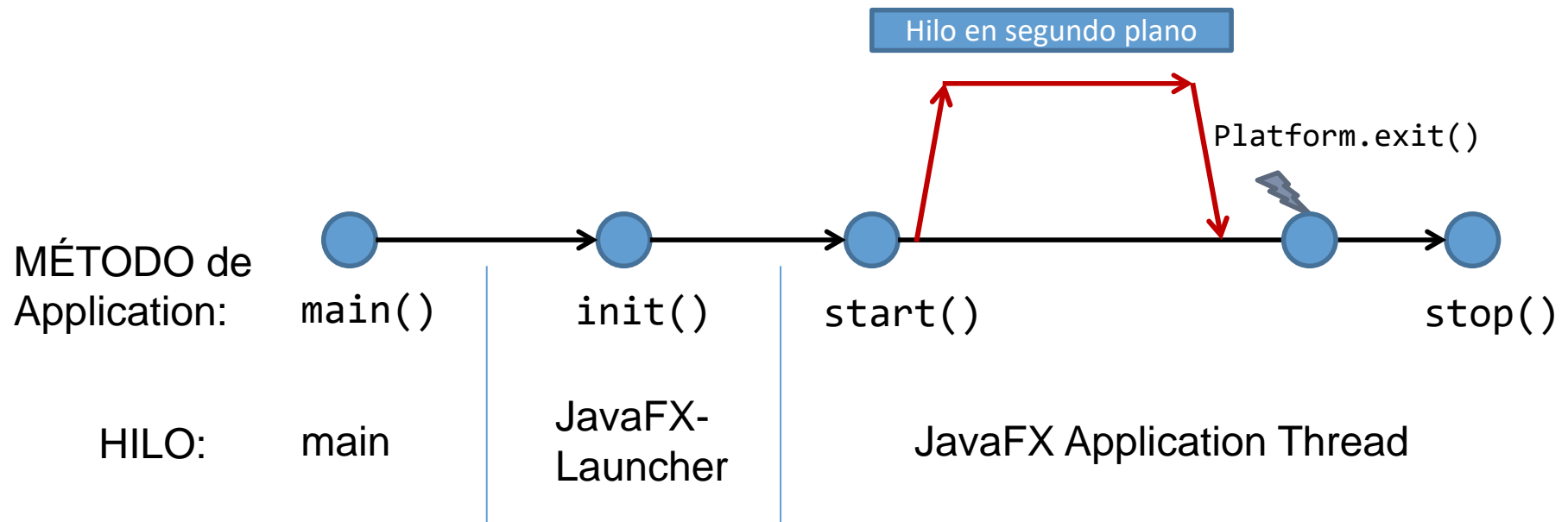
- Introducción
- Concurrencia en JavaFX
- La interface Worker<V>
- La clase Task<V>
- La clase Service<V>
- La clase WorkerStateEvent
- Cambiando el cursor
- Herramientas útiles
- Ejercicio
- Bibliografía

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 pueden usarse otros hilos:

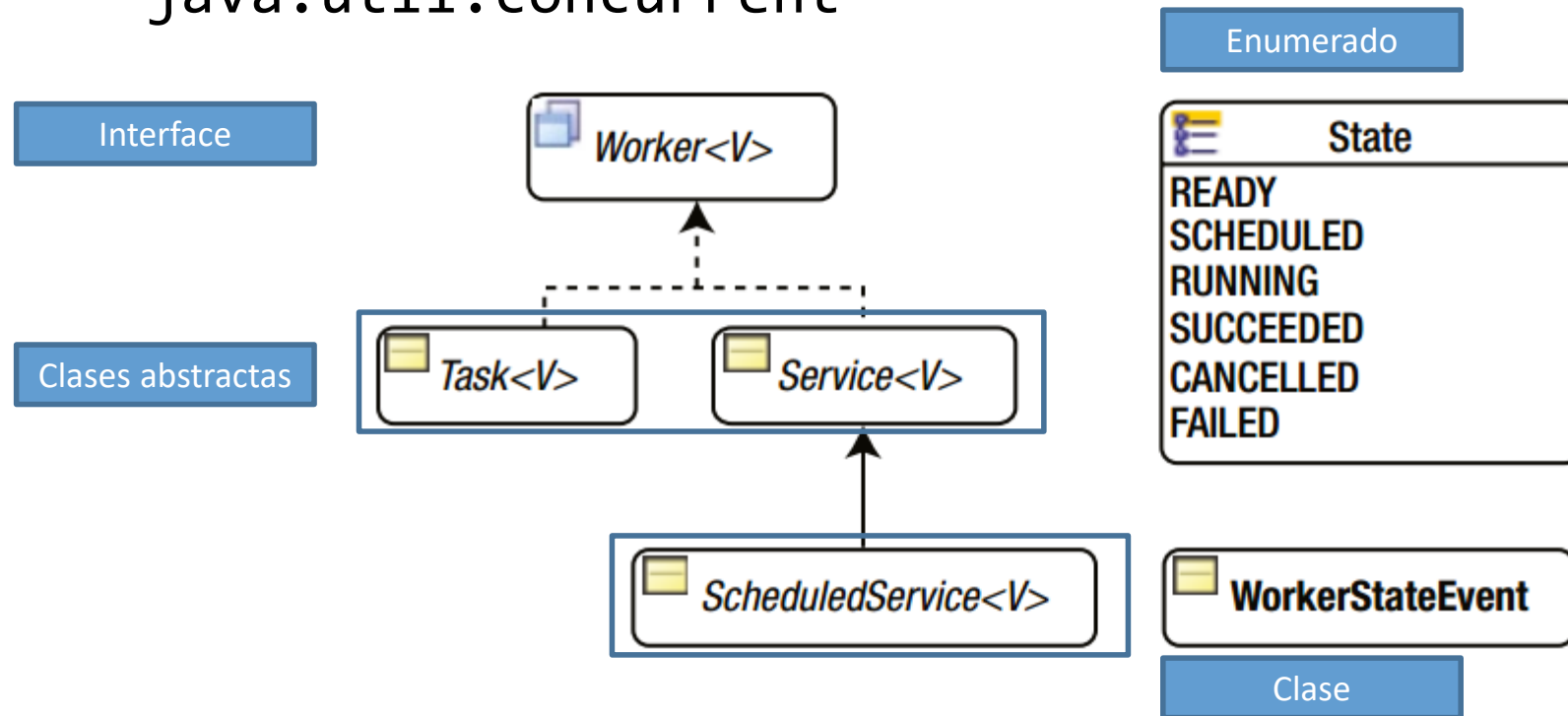


Introducción

- Tratamos de resolver los siguientes problemas:
 - Realizar procesamiento en segundo plano mientras se puede hacer algo en el primero. Se utiliza la interface `Worker` y las clases:
 - `Task`, `Service`, `ScheduledService`.
 - Actualizar la interfaz de usuario desde un hilo de control en segundo plano. Se puede utilizar
 - `Platform.runLater`

Concurrencia en JavaFX

- El marco de concurrencia de JavaFX está pensado para ser utilizado en el contexto de la interfaz gráfica de usuario y es relativamente pequeño ya que se basa en `java.util.concurrent`



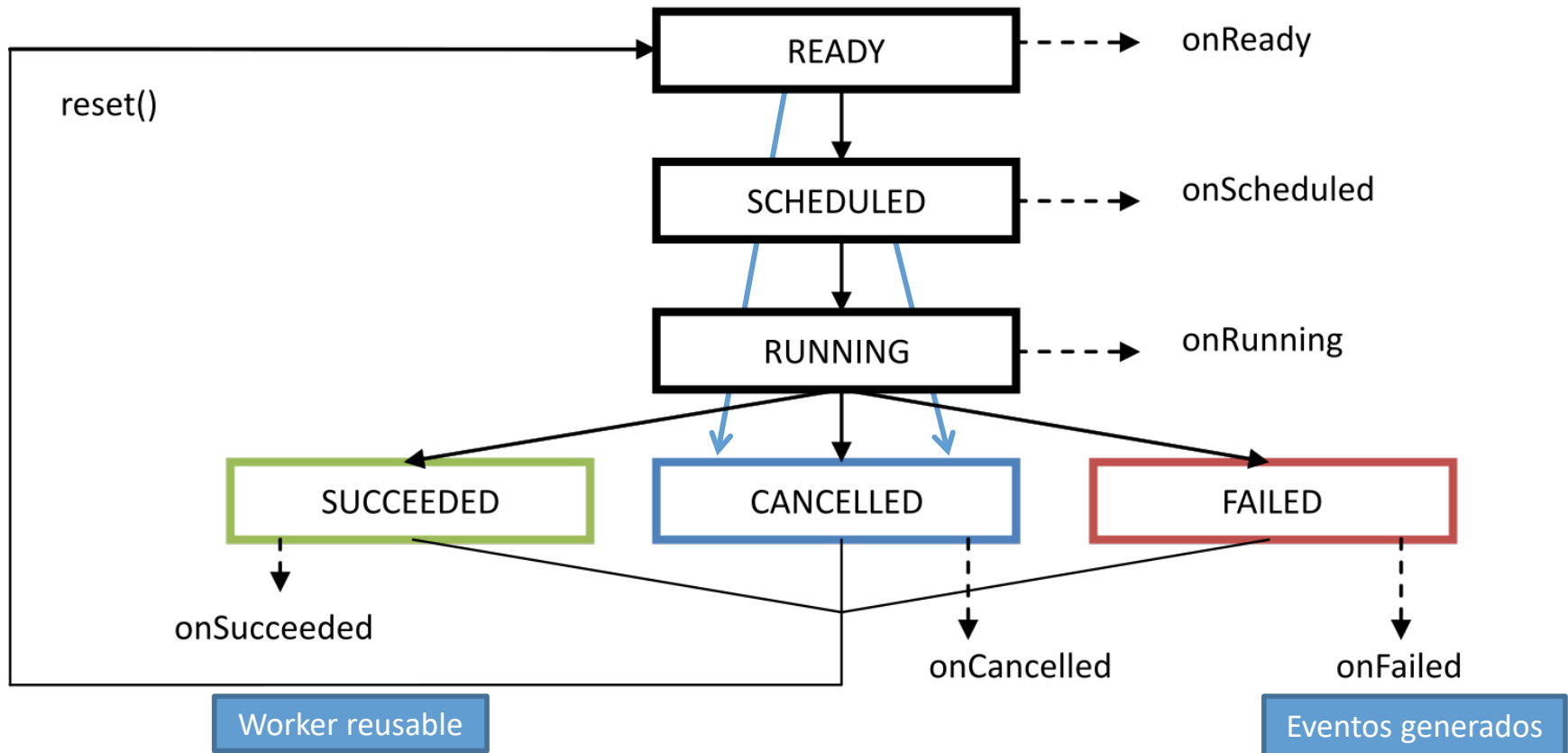
Hablaremos genéricamente de tareas en lugar de especificar `Worker`, `Task`, `Service`, etc.

Concurrencia en JavaFX

- Una instancia de `Worker` representa una tarea que debe ser ejecutada en uno o más hilos en segundo plano
- Las clases `Task`, `Service` y `ScheduledService` implementan a la interfaz `Worker` y son clases abstractas.
- Un instancia de `Task` representa una tarea **no reusable**.
- Una instancia de `Service` representa una tarea **reusable**.
- Una instancia de `ScheduledService` es una tarea que puede ser planificada y **ejecutada repetidas veces** después de un intervalo de tiempo (cuando termina se inicia otra vez)
- Los valores del tipo enumerado `State` representan **estados** en los que se encuentra un `Worker`
- Una instancia de `WorkerStateEvent` representa un evento que ocurre cuando el estado de `Worker` cambia

Concurrencia en JavaFX

- Ciclo de vida de las tareas



- Las no reusables no tienen el estado Ready.

Interface Worker<V>

- Es un tipo de tarea que puede ser ejecutado en uno o más hilos en segundo plano.
- El parámetro V es el tipo del resultado que produce el Worker, se pone Void (tal como está escrito) si no genera nada.
- El estado de la tarea es observable. El estado se publica en el hilo de control principal de JavaFX lo que permite la comunicación con el grafo de escena.

Interface Worker<V>

- Algunas propiedades que representan el estado interno:

title	: Nombre de la tarea
message	: Mensaje detallado durante la ejecución, para feedback
running	: True si está ejecutándose o en el estado Scheduled
state	: Valor del tipo enumerado
Progress	: Ratio entre workDone y totalWork
workDone	: Cantidad de trabajo hecho
totalWork	: Cantidad de trabajo a realizar
value	: Valor devuelto por la tarea cuando termina en éxito. null en otro caso. Si V es Void el valor es null.
exception	: Excepción que se genera si la tarea falla.

Si no se conocen toman
el valor -1

Utilización de Task<V>: definición

- Representa una tarea que se ejecuta una única vez, si termina, se cancela, o falla, no puede ser reiniciada.
- Para crear una Tarea tenemos que extender la clase Task<V> e implementar el método abstracto call()
- Si queremos crear una tarea que retorna una lista observable de números primos, la declaración es:

```
package p8tasklistanumerosprimos;
```

```
import javafx.collections.ObservableList;
```

```
import javafx.concurrent.Task;
```

```
// Devuelve una lista de número primos.
```

```
public class TareaEncontrarPrimos extends Task<ObservableList<Long>>
```

```
{
```

```
    @Override
```

```
    protected ObservableList<Long> call() throws Exception {
```

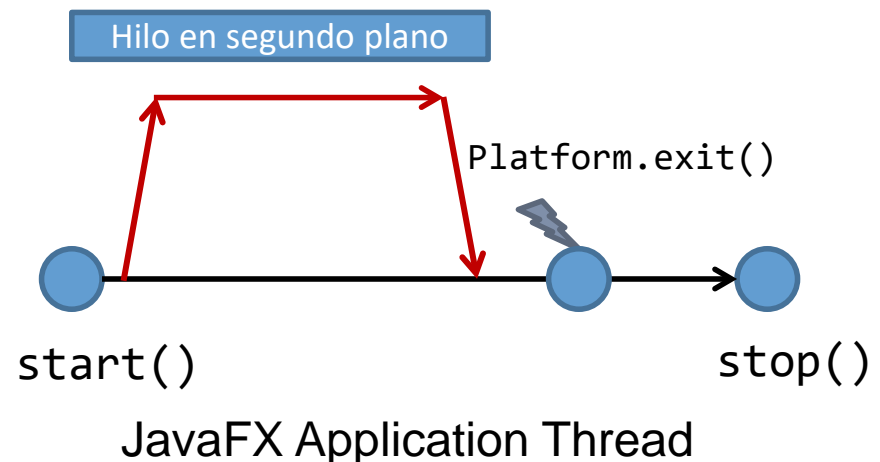
```
    }
```

```
}
```

Utilización de Task<V>: actualización

- Los siguientes métodos suelen utilizarse cuando la tarea está progresando y sirven para actualizar algunas de sus propiedades, se pueden observar desde el hilo principal de ejecución de JavaFX

```
protected void updateMessage(String message)
protected void updateProgress(double workDone, double totalWork)
protected void updateProgress(long workDone, long totalWork)
protected void updateTitle(String title)
protected void updateValue(V value)
```



Utilización de Task<V>: actualización

- Los métodos de actualización anteriores obviamente son llamados desde el método `call()` de la tarea.
- Para acceder desde el hilo principal (GUI) debe utilizarse

`Platform.runLater()` (explicado después)

- O bien vincular las propiedades (`bind`) de la tarea con elementos de la interfaz de usuario.

Utilización de Task<V>: oyentes

- La tarea contiene las siguientes propiedades que permiten definir **oyentes** de los cambios de estado de la tarea

```
onCancelled  
onFailed  
onRunning  
onScheduled  
onSucceeded
```

- El siguiente fragmento de código lleva un manejador de evento

```
Task<ObservableList<Long>> task = crear la tarea...  
task.setOnSucceeded(e -> { System.out.println("La tarea ha terminado.") });
```

Utilización de Task<V>: cancelación

- Con uno de los dos siguientes métodos se puede cancelar una tarea.

```
public final boolean cancel()  
public boolean cancel(boolean mayInterruptIfRunning)
```

- El primer método quita la tarea de la cola de ejecución o termina su ejecución.
- La segunda permite decidir si el hilo de control de la tarea ejecutándose puede ser interrumpido o no.
- Dentro del método `call()` de la tarea debe comprobarse si ha sido cancelada (`isCancelled()`), una vez detectada debe salirse del método `call`, en caso contrario `cancel(true)` no funcionará de manera apropiada.

Utilización de Task<V>: ejemplo

- Ejemplo una tarea que calcula el factorial

```
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;
    }
};
```

Estamos usando una clase anónima, más adelante definiremos una clase Factorial

Utilización de Task<V>: arranque

- La tarea se puede iniciar de dos formas:

```
// Programar la tarea en un hilo en segundo plano  
Thread backgroundThread = new Thread(task);  
backgroundThread.setDaemon(true);  
backgroundThread.start();
```

```
// Usar el servicio de ejecución para programar la tarea  
ExecutorService executor = Executors.newSingleThreadExecutor();  
executor.submit(task);
```

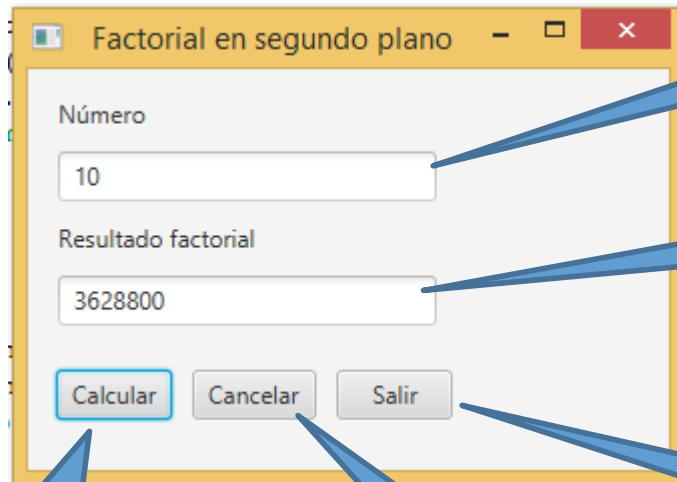
Task<V>: Esperas dentro de la tarea

- Si se llama a `Thread.sleep` se produce un retardo y si se cancela la tarea se genera una `InterruptedException`. Hay que comprobar la cancelación de nuevo:

```
class Factorial extends Task<Long> {  
    private Long calculaFactorial; //valor para el que se calcula el factorial  
    @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;  
    }  
}
```

Task<V>: ejemplo factorial

- Ejemplo de calculo del factorial como hilo secundario, definido desde el controlador de la aplicación.



Valor introducido

Resultado de la tarea

Sale de la aplicación

Crea la tarea de cálculo
en segundo plano

Cancela la tarea

Task<V>: ejemplo factorial

- En un archivo aparte o bien como clase local al controlador definimos la clase Factorial

```
// clase local para crear la tarea que calcula el factorial
class Factorial extends Task<Long> {

    private Long calculaFactorial; //valor para el que se calcula el factorial
    public Factorial() { }
    public Factorial(Long valor) { calculaFactorial = valor; }

    @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;
    }
}
```

Task<V>: ejemplo factorial

- Ahora tenemos que definir el código de los botones para lanzar la tarea y para cancelarla.

```
public class FXMLDocumentController implements Initializable {  
  
    private Factorial miTarea;  
    @FXML  
    private TextField numero; // campo de entrada  
    @FXML  
    private TextField factorialResultado; //campo de salida  
    ..  
}  
  
    Botón calcular  
@FXML void handleButtonCalcular(ActionEvent event) {  
    Long factorial;  
    // no se comprueban errores de formato en el número introducido  
    factorial = Long.parseLong(this.numero.getText());  
    miTarea = new Factorial(factorial);  
    factorialResultado.textProperty().bind(Bindings.convert(miTarea.valueProperty()));  
    Thread th = new Thread(miTarea);  
    th.setDaemon(true);  
    th.start();  
}
```

Lanza el hilo de la tarea

Task<V>: ejemplo factorial

- Botón cancelar tarea

```
@FXML void handleCancelar(ActionEvent event) {  
    miTarea.cancel();  
}
```

- Para ver el resultado en la interfaz vinculamos el campo de texto de la pantalla con valueProperty de la tarea

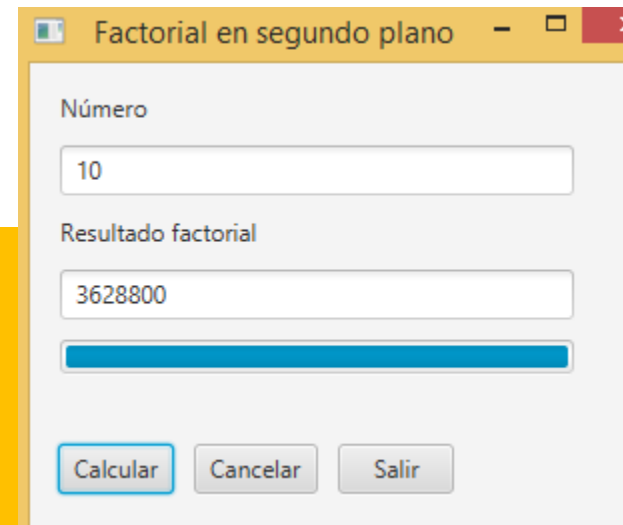
```
factorialResultado.textProperty().bind(Bindings.convert(miTarea.valueProperty()));
```

- ¿Cómo podemos ver el progreso del cálculo del factorial?
 - Dentro del método call() de la tarea usaremos updateProgress
 - En la interfaz de usuario situaremos una barra de progreso y vincularemos (Bind) la propiedad progressProperty de la barra con la propiedad progressProperty de la tarea

Task<V>: ejemplo factorial

- Para el progreso en la tarea

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



- Para el progreso en la interfaz de usuario (en controlador)

```
@FXML void handleButtonCalcular(ActionEvent event) {  
    Long factorial;  
    factorial = Long.parseLong(this.numero.getText()); // no se comprueban errores de formato  
    miTarea = new Factorial(factorial);  
    factorialResultado.textProperty().bind(Bindings.convert(miTarea.valueProperty()));  
    barraProgreso.progressProperty().bind(miTarea.progressProperty());  
    Thread th = new Thread(miTarea);  
    th.setDaemon(true);  
    th.start();  
}
```

Task<V>: ejemplo factorial

- Se puede usar la propiedad `runningProperty` para ocultar o mostrar elementos del interfaz mientras la tarea se ejecuta.
- Ocultamos el textbox de resultado mientras se calcula
- Deshabilitamos el botón calcular mientras está ejecutándose la tarea

```
@FXML    void handleButtonCalcular(ActionEvent event) {  
...  
factorialResultado.visibleProperty().bind(Bindings.not(miTarea.runningProperty()));  
// texto resultado deshabilitado  
calcular.disableProperty().bind(miTarea.runningProperty());  
//botón deshabilitado mientras se ejecuta la tarea  
    Thread th = new Thread(miTarea);  
    th.setDaemon(true);  
    th.start();  
}
```


Task<V> RunLater

Ejemplo factorial
Quitar el bind entre
factorialResultado y valueProperty
de la tarea. Pag. 23

- Platform.runLater se ejecuta en el hilo principal de una aplicación JavaFX, es útil para actualizar la interfaz de usuario desde un hilo de control en segundo plano, siempre que el procesamiento no sea muy largo en tiempo.

```
@Override protected Long call() throws Exception {  
    long f = 1;  
    for (long i = 2; i <= calculaFactorial; i++) {  
        if (isCancelled()) { break; }  
        f = f * i;  
        updateProgress(i, calculaFactorial);  
        updateValue(f);  
        Platform.runLater(() -> {  
            factorialResultado.setText(miTarea.valueProperty().get().toString());  
        });  
        try { Thread.sleep(100); }  
        catch (InterruptedException e) { if (isCancelled()) break; }  
    }  
    return f;  
}
```

La clase Task: resumen

- Usaremos esta clase para implementar el código que se ejecutará en un hilo en segundo plano:
 - Derivar una clase de Task
 - Sobrescribir el método `call`, con la lógica a ejecutar devolviendo el resultado si hubiese alguno. 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 Service<V>

- La clase abstracta Service<V> implementa a la interfaz Worker<V> y encapsula a Task<V>
- 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
- En el ejemplo anterior podemos convertir la Task<Long> en Service<Long>

La clase Service<V>

- Como clase local del controlador o en otro fichero

```
class MiServicio extends Service<Long> {  
  
    private Long factorial; // número para el factorial  
    public MiServicio(Long numero){ factorial = numero; }  
    @Override  
    protected Task<Long> createTask() {  
        return new Factorial(factorial);  
    }  
  
}
```

- El método createTask() se llama automáticamente cada vez que el servicio se inicia o se re-inicia

La clase Service<V>

- No tiene métodos updateXXX ya que estos están asociados a la tarea contenida en el servicio.
- Los oyentes de eventos de transiciones son los mismos que en el caso de la tarea, pero añade un oyente nuevo (onReady property)

La clase Service<V>

- Utilice el método start() para iniciar un servicio

```
MiServicio miservicio = new MiServicio(factorial2);  
miservicio.start();
```

- cancel() para cancelarlo
- reset(): pone las propiedades en su valor inicial. La llamada genera una excepción si está en el estado RUNNING O SCHEDULED
- Para reiniciar el servicio utilice restart(), internamente llama a cancel(), reset() y start()

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());
```

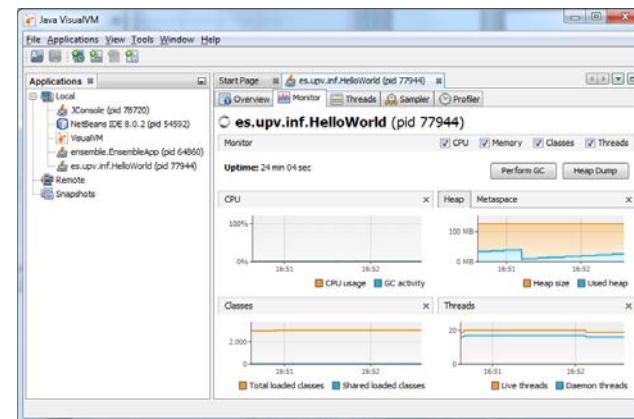
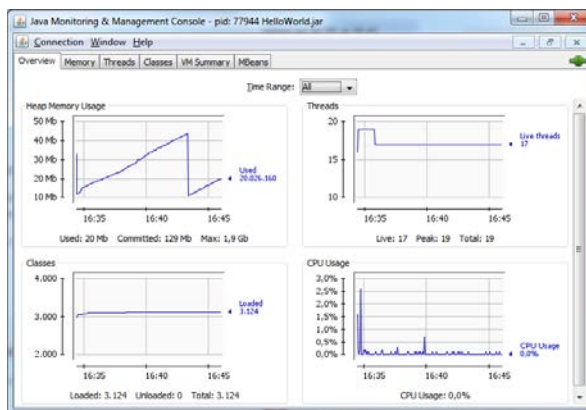
Cambiando el cursor

- 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;
}
```

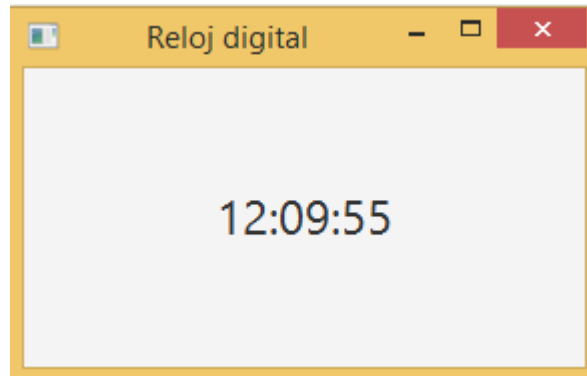

Herramientas útiles

- 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

- La interfaz de usuario que se muestra contiene un reloj digital



- Añadir dos botones, uno para que el reloj muestre el tiempo y otro para que lo pare.

Bibliografía

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