2. Programación concurrente

Parte III. *Callable*, *Future* y *framework* JavaFX Async

Programación de Servicios y Procesos

Nacho Iborra
Álvaro Pérez
Javier Carrasco

Tabla de contenido

5. Callable and Future

Además de *Runnable*, los ejecutores soportan otro tipo de tareas llamadas ***Callable***. Los *callables* son interfaces funcionales igual que los *runnables*, pero en lugar de ser nulos, devuelven un valor. La interfaz *Callable* define el tipo de datos devueltos mediante genéricos.

5.1. Uso de *Callable*

En este ejemplo, se creará un hilo *callable* usando un *ExecutorService* (*ThreadPoolExecutor* implementa *ExecutorService*, por tanto, es lo mismo). Al enviar un *Callable* a un *Executor*, devolverá un objeto ***Future***. Un *Future* es solo una interfaz que tiene los métodos necesarios para obtener el resultado devuelto por *Callable*.

Sin embargo, cada vez que se llame a ***get()*** en el objeto *Future*, el hilo actual se bloquea hasta que el hilo *callable* devuelva algo. Por lo que es una buena idea llamar antes al método ***isDone()***, solo para comprobar si el *callable* ha finalizado su tarea. Además, no se utiliza el método *execute()* para añadir *Futures* al ejecutor: se utilizará el método ***submit()*** en su lugar, para obtener el resultado más tarde.

```
public static void main(String[] args) {
    Callable<Integer> callInt = () -> {
        try {
            TimeUnit.SECONDS.sleep(3);
            return 20;
        } catch (InterruptedException e) {
            throw new IllegalStateException("task interrupted", e);
        }
    };

    ExecutorService executor = Executors.newFixedThreadPool(1);

    // Calling submit executes the thread and returns a Future
    Future<Integer> future = executor.submit(callInt);

    executor.shutdown();

    System.out.println("future done? " + future.isDone());
    Integer result;
    try {
        result = future.get(); // It BLOCKS main thread until it returns!
        System.out.println("future done? " + future.isDone());
        System.out.println("Result: " + result); // Prints 20
    } catch (InterruptedException ex) { }
    catch (ExecutionException ex) { }
}
```

5.1.1. Pasando un *timeout*

Al llamar a ***get()*** en el objeto *Future* para recuperar el resultado, se puede pasar un tiempo de espera, de modo que cuando pase ese tiempo, si el hilo no ha terminado, lanzará un ***TimeoutException***. Puede ser una buena idea cancelar la tarea cuando eso suceda:

```
try {
    result = future.get(1, TimeUnit.SECONDS); // Blocks 1 second maximum
    System.out.println("Result: " + result);
} catch (InterruptedException ex) {
} catch (ExecutionException ex) {
} catch (TimeoutException ex) { // When the timeout expieres...
    System.err.println("The thread took more than 1 second to complete!");
    executor.shutdownNow(); // Cancel immediately all pending tasks
}
```

5.1.2. Lanzar varias tareas *callables* al mismo tiempo

Se puede lanzar más de un hilo callable al mismo tiempo usando un ejecutor. Si se pasa una lista de *Callables* usando ***invokeAll()***, devolverá una lista de *Futures*. Iterando sobre la lista *Future* y llamando a *get()* se deberían obtener los resultados. Se obtendrán todos los resultados cuando termine el último hilo.

```
public static Callable<Integer> getSumCallable(int num1, int num2, int secondsSleep) {
    return () -> {
        try {
            TimeUnit.SECONDS.sleep(secondsSleep);
            return num1 + num2;
        } catch (InterruptedException e) {
            throw new IllegalStateException("task interrupted", e);
        }
    };
}

public static void main(String[] args) {
    List<Callable<Integer>> callables = Arrays.asList(
            getSumCallable(3, 6, 2),
            getSumCallable(5, 8, 3),
            getSumCallable(12, 3, 1)
    );
    ExecutorService executor = Executors.newWorkStealingPool();
    List<Future<Integer>> futures;

    try {
        futures = executor.invokeAll(callables);
        executor.shutdown();

        futures.forEach(future -> {
            try {
                System.out.println(future.get());
            } catch (InterruptedException | ExecutionException e) {
                throw new IllegalStateException(e);
            }
        });
    } catch (InterruptedException ex) { }
}
```

En este caso, se ha creado un método estático que devuelve el objeto *callable*. Se llama a este método varias veces para agregar muchas llamadas a la lista. Después se invocan todos desde el ejecutor.

Si no se quiere esperar hasta finalizar todas las tareas y, se desea obtener solo el resultado de la primera tarea que finalice, se puede usar ***invokeAny()***. Esto devolverá un único objeto *Future* que debería obtener el resultado del primer hilo que finaliza sin errores. Cuando una tarea termine primero y devuelva un valor, el resto de tareas se cancelan.

```
ExecutorService executor = Executors.newWorkStealingPool();
try {
    // Blocks and returns first result
    int firstResult = executor.invokeAny(callables);
    executor.shutdown();
    System.out.println(firstResult); // 15 ---> 12 + 3 finishes in 1 second
} catch (InterruptedException ex) {
} catch (ExecutionException ex) { }
```

5.1.3. Ejecutores programados

Si se quisiera ejecutar una tarea periódicamente, en lugar de hacerlo manualmente, se puede usar un ***ScheduledExecutorService***. En primer lugar, se verá un ejemplo de una tarea que no se ejecuta periódicamente, sino que tiene un retraso y espera unos segundos antes de comenzar. Este tipo de servicio ejecutor devuelve un objeto ***ScheduledFuture***.

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

try {
    // Usage: schedule(Callable/Runnable, delay, Time unit)
    ScheduledFuture<Integer> schedFuture = executor.schedule(
            getSumCallable(3, 6, 2), 3, TimeUnit.SECONDS);

    executor.shutdown();
    TimeUnit.MILLISECONDS.sleep(1500); // Sleeps for about 1.5 seconds
    long remainingDelay = schedFuture.getDelay(TimeUnit.MILLISECONDS);
    System.out.printf("Remaining Delay: %dms\n", remainingDelay); // 1498ms
    int result = schedFuture.get(); // blocks 3.5 sec. (1.5 delay + 2
    System.out.println("Result: " + result);
} catch (InterruptedException ex) {
} catch (ExecutionException ex) { }
```

Para ejecutar una tarea programada (tarea que se ejecuta cada X veces), deberá llamarse a uno de estos dos métodos: ***scheduleAtFixedRate()*** o ***scheduleWithFixedDelay()***.

scheduleAtFixedRate() siempre lanzará un hilo nuevo cada X veces y no le importa el tiempo de ejecución de la tarea. Por ejemplo, si se programa para ejecutar una nueva tarea cada 3 segundos pero la tarea necesita 5 segundos, el ejecutor intentará ejecutar una nueva tarea cuando hayan pasado 3 segundos desde que lanzó la anterior (la segunda tarea se ejecutará en el mismo tiempo que el primero, y así sucesivamente).

Utilizando ***scheduleWithFixedDelay()***, por lo general, puede ser una mejor idea, porque la demora para la siguiente tarea comenzará cuando finalice la tarea actual (no cuando comience). Es importante no llamar a ***executor.shutdown()*** hasta que se quiera cancelar la tarea programada.

```
public static void main(String[] args) {
    Runnable task = () -> {
        System.out.println("Time now: " + LocalTime.now().toString());
    };

    ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

    // Delay (1 second), runs every 3 seconds
    executor.scheduleWithFixedDelay(task, 1, 3, TimeUnit.SECONDS);
```

```
    BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
    String command;
    try {
        do {
            // When user presses "q" and "enter", program will end
            command = in.readLine();
        } while (!command.equals("q"));

        executor.shutdown(); // Cancel the scheduled task
    } catch (IOException e) { }
}
```

Ejercicio 11

Crea un proyecto llamado ***CallableWordCounting***. Lanza 3 hilos *callables* al mismo tiempo usando el método ***invokeAll()*** del ejecutor.

Cada hilo leerá un archivo de texto diferente (crea 3 archivos de texto con mucho texto) y buscará cuántas veces aparece un texto en ese archivo. Al final, devolverá el número de veces que ha aparecido el texto en el archivo (*Integer*). **Pista**: Crea una clase que implemente ***Callable<Integer>*** y se le pase al constructor el nombre del archivo y el texto a buscar, o crear un método estático que reciba estos 2 parámetros y devuelva un lambda C*allable.*

El hilo principal obtendrá los resultados y los sumará, imprimiendo el número total de veces que la palabra o el texto ha aparecido en cada uno de los archivos (observa que no se necesita ninguna sección sincronizada o variable atómica para este ejercicio).

5.2. Uso de *CompletableFuture*

CompletableFuture es una clase que implementa las interfaces ***Future*** y ***CompletionStage***. *CompletionStage* representa una promesa. Esto es una gran ventaja sobre la metodología anterior, porque no bloquea el hilo actual mientras espera a que finalice la tarea. En su lugar, se le proporciona una devolución de llamada (una función que se ejecutará después de que finalice la tarea y devuelva su resultado). Este tipo de ejecutores admiten *runnables* y *callables*.

Este ejemplo usa un *runnable* (esto es, un *CompletableFuture* que no devuelve nada) para imprimir un mensaje en la pantalla después de 3 segundos. Se lanzará con el método ***runAsync()***.

```
public static void main(String[] args) {
    // runAsync receives a runnable that doesn't return anything
    CompletableFuture<Void> compRunnable = CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(3);
            System.out.println("Task completed");
        } catch (InterruptedException ex) { }
    });

    // thenRun runs another task (in main thread) when the current task finishes
    compRunnable.thenRun(() -> System.out.println("CompletableFuture finish"));
```

```
    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println("Press enter to exit (let the task finish first)");
    try {
        in.read();
    } catch (IOException ex) { }
}
```

Si se utiliza *Callable* en lugar de *Runnable*, entonces se debería usar el método ***supplyAsync()*** en lugar de *runAsync()*, se puede usar esto para procesar el resultado devuelto por la tarea asíncrona (este procedimiento también es asíncrono). En este ejemplo, el *callable* devuelve un número entero aleatorio entre 0 y 100, y estos datos se almacenan en un objeto *CompletableFuture*.

```
public static void main(String[] args) {
    CompletableFuture<Integer> compRunnable = CompletableFuture.supplyAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(3);
            return (new Random()).nextInt(100); // Return random 0 100
        } catch (InterruptedException ex) {
            return -1;
        }
    });

    // thenAccept receives the r esult of the previous task to process
    compRunnable.thenAccept((num) -> System.out.println("Number generated: " + num));

    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println("Press enter to exit (let the task finish first)");
    try {
        in.read();
    } catch (IOException ex) { }
}
```

CompletableFuture puede verse como una versión asincrónica de Stream en Java. Permite aplicar/encadenar múltiples filtros al resultado obtenido en primer lugar. Para ejecutar tareas intermedias, utiliza ***thenAccept()*** (no devuelve nada) o ***thenApply()*** (devuelve otro resultado).

Este ejemplo usa *CompletableFuture* para obtener una cadena formateada con el nombre de una persona y una edad (separados por un punto y coma). Una vez que se obtienen los datos, lanza una segunda tarea asincrónica para dividir la cadena y devolver un objeto *Person* con esos atributos. Finalmente, lanza una tercera tarea asincrónica para imprimir la persona en la pantalla.

```
public static void main(String[] args) {
    CompletableFuture<String> compRunnable = CompletableFuture.supplyAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(3);
            return "Peter;28";
        } catch (InterruptedException ex) {
            return "Error;0";
        }
    });

    // thenApply gets the previous result and returns another (Person)
    CompletableFuture<Person> comPerson = compRunnable.thenApply((str) -> {
        String[] parts = str.split(";");
        return new Person(parts[0], Integer.parseInt(parts[1]));
```

```
    });

    // thenRun runs the final task with the last processed result
    comPerson.thenAccept((person) -> System.out.println(person));

    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println("Press enter to exit (let the task finish first)");
    try {
        in.read();
    } catch (IOException ex) { }
}
```

¿Qué sucede si la tarea original lanza una excepción y hay que recuperarla? (devolver datos válidos que se puedan procesar). Se puede usar el método ***exceptionally()***. Este método actuará como una declaración *catch* para las excepciones que se lanzan en la tarea anterior (debe devolver el mismo tipo de datos que la tarea anterior). En este ejemplo, se verá cómo encadenar todas estas tareas sin usar variables intermedias:

```
public static void main(String[] args) {
    CompletableFuture.supplyAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException ex) { }
        return "Peter;28";
    }).exceptionally((error) -> { // Only if previous step throws an error
        System.err.println("Error: " + error.getMessage());
        return "Error;0";
    }).thenApply((str) -> { // Process the string and return a Person
        String[] parts = str.split(";");
        return new Person(parts[0], Integer.parseInt(parts[1]));
    }).thenAccept((person -> System.out.println(person)));

    InputStreamReader in = new InputStreamReader(System.in);
    System.out.println("Press enter to exit (let the task finish first)");
    try {
        in.read();
    } catch (IOException ex) { }
}
```

Finalmente (aunque la API de *CompletableFuture* tiene muchas más posibilidades), se verá cómo ejecutar una tarea cuando un número (mayor que 1) de *CompletableFutures* finaliza sus tareas, usando ***CompletableFuture.allOf***. En este ejemplo, se crearán tareas que intentarán hacer ping a algunos servidores, y al final, se mostrarán los resultados y finalizará el programa.

```
public class ThreadsExamples {
    public static Deque<String> ipMessages = new ConcurrentLinkedDeque<>();

    public static CompletableFuture<Void> pingIp(String address) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                InetAddress inet = InetAddress.getByName(address);
                if (inet.isReachable(4000)) { // 4 seconds
                    return true;
                }
            } catch (UnknownHostException ex) {
            } catch (IOException ex) { }
            return false;
        }).thenAccept((result) -> {
```

```
            ipMessages.add(address + (result ? " ping OK" : " unreachable"));
        });
    }

    public static void main(String[] args) {
        CompletableFuture<Void> allTasks = CompletableFuture.allOf(
                pingIp("google.com"),
                pingIp("wikipedia.org"),
                pingIp("apache.org"),
                pingIp("facebook.com")
        );

        allTasks.thenRun(() -> {
            System.out.println("All tasks finished");
            System.out.println(ipMessages);
        });

        while (!allTasks.isDone()) {
            try {
                TimeUnit.MILLISECONDS.sleep(500);
            } catch (InterruptedException ex) { }
        }
    }
}
```

Si en lugar de ***allOf()***, utilizas ***CompletableFuture.anyOf()***, se ejecutaría *thenRun()* al finalizar la primera tarea, en lugar de esperar a todas.

Ejercicio 12

Crea un proyecto llamado ***FastestWordCounting***. Este ejercicio será similar al ejercicio 11, pero con algunas diferencias.

Crea un objeto *CompletableFuture<Integer>* en lugar de un objeto *Callable<Integer>* para leer cada archivo.

Esta vez, utiliza el método ***CompletableFuture.anyOf()***. Este método también devolverá un *CompletableFuture<Integer>*, que recibirá el valor del hilo que termina primero. Cuando este primer hilo termine (*thenRun*), muestra un mensaje como este:

```
El primer hilo ha terminado y ha encontrado el texto "gato" 24 veces.
```

6. Tareas asincrónicas de JavaFX

Cuando se desarrolla una aplicación gráfica con una biblioteca Java determinada, como *Swing*, JavaFX o incluso una aplicación de Android, se debe tener en cuenta que todas utilizan un **hilo único para procesar todos los eventos de la UI**. Esto se debe a que los controles o nodos que se colocan en una escena son hilos no seguros, por lo que son rápidos (ya que no necesitan ningún mecanismo de sincronización), pero es necesario acceder a ellos desde un solo hilo. En el caso de JavaFX, por ejemplo, este hilo debe ser el hilo principal de la aplicación JavaFX. Como consecuencia de esto, no debería haber ninguna tarea de ejecución prolongada en este hilo principal de la aplicación, porque toda la aplicación se bloquearía hasta que finalice esta tarea.

> Este problema no se aplica a las animaciones en JavaFX. Cuando se utilizan una clase de transición, o un *KeyFrame* y clases *Timeline* para definir una transición, administran sus propios hilos para realizar la animación fuera del hilo principal de la aplicación, y se puede interactuar con la aplicación mientras se ejecuta la animación.

6.1. Presentando el problema. Resolviéndolo con hilos básicos

Observa este problema con un ejemplo de JavaFX:

```
public class Example_JavaFXThreads extends Application {
    // Copy progress
    int progress;

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        Label lblProgress = new Label("");
        Button btnStart1 = new Button("Start copy (1)");

        btnStart1.setOnAction(e -> {
            for (int progress = 1; progress <= 10; progress++) {
                try {
                    Thread.sleep(1000);
                    lblProgress.setText("" + (progress * 10) + "% completed");
                } catch (Exception ex) { }
            }
        });

        VBox vb = new VBox(20);
        vb.setAlignment(Pos.CENTER);
        vb.getChildren().addAll(lblProgress, btnStart1);
        Scene scene = new Scene(vb, 300, 400);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

La aplicación del ejemplo simula la copia de un archivo grande, cuando se pulsa la el botón *“Start copy (1)”*, se imprime un mensaje cada segundo que muestra el porcentaje de archivo que se ha copiado por ahora. Si lanzas la aplicación y haces clic en el botón *“Start copy (1)”*, descubrirás que:

- La etiqueta no actualiza su porcentaje, como debería.
- Si intentas cerrar la aplicación mientras la tarea se está ejecutando, no se cerrará.

¿Por qué pasa esto? Como se ha dicho antes, todo el manejo de eventos de la aplicación se ejecuta en el hilo principal de la aplicación. Entonces, cuando está durmiendo y cambiando el texto de la etiqueta en el bucle de eventos, no se está ejecutando nada más (toda la aplicación está esperando que termine este evento).

6.1.1. Intentando solucionar el problema. Primer intento

Se podría pensar que, para resolver el problema mostrado en el ejemplo anterior, se podría simplemente llamar a un hilo que copie el archivo y actualice el progreso en la etiqueta. Vamos a hacerlo. Para mantener el programa original en su versión original, se añadirá un nuevo botón, *“Start copy (2)”*, y se creará un hilo en su *ActionEvent* para hacer la misma tarea que se hizo antes en el controlador de eventos del primer botón.

Se añadirá el botón con su controlador de eventos:

```
Button btnStart2 = new Button("Start copy (2)");

// "Start copy (2)" event: calling a thread to do the task
btnStart2.setOnAction(e -> {
    Thread t = new Start2Thread(lblProgress);
    t.start();
});
```

Después, se añadirá la clase de hilo. Se pasará la etiqueta como parámetro para tenerla accesible. En el método *run()* se copia el mismo código que se utilizó para el evento de *btnStart1*.

```
public class Start2Thread extends Thread {
    // Progress label to update its text
    Label lblProgress;

    public Start2Thread(Label lblProgress) {
        this.lblProgress = lblProgress;
    }

    @Override
    public void run() {
        for (int progress = 1; progress <= 10; progress++) {
            try {
                Thread.sleep(1000);
                lblProgress.setText("" + (progress * 10) + "% completed");
            } catch (Exception ex) { }
        }
    }
}
```

Añade el btnStart2 a la UI en la siguiente línea.

```
vb.getChildren().addAll(lblProgress, btnStart1, btnStart2);
```

Si haces clic en este segundo botón, se lanzará un *IllegalStateExcepcion*. El motivo se encuentra en la siguiente línea de código dentro del método *run()*:

```
lblProgress.setText("" + (progress * 10) + "% completed");
```

Como se dijo antes, nadie más que el hilo principal de la aplicación puede acceder a la interfaz de usuario, ya que sus controles son hilos no seguros.

6.1.2. Intentando solucionar el problema. Segundo y último intento

El problema al usar un hilo secundario es que no se puede acceder a los elementos de la interfaz de usuario desde él. Para evitar esto, algunas bibliotecas y *frameworks* como JavaFX o Android proporcionan algunas formas de pasar tareas de esos hilos secundarios a la aplicación principal. En este caso, se necesita pasar a la aplicación principal la tarea de actualizar el texto de *lblProgress*. Se puede reemplazar esta línea de código en el método *run()* del hilo:

```
lblProgress.setText("" + (progress * 10) + "% completed");
```

Por estas otras:

```
int finalProgress = progress;
Platform.runLater(() -> lblProgress.setText("" + (finalProgress * 10) + "% completed"));
```

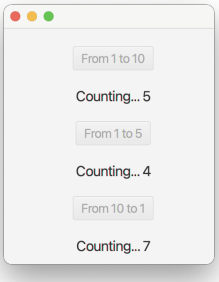
Se introduce un nuevo método: el ***Platform.runLater()***, cuya misión es programar la(s) tarea(s) especificadas para que se ejecuten en el hilo principal de la aplicación en un momento no especificado en el futuro (no se puede controlar cuándo). De esta forma, siempre que se intente actualizar el estado de un control desde fuera del hilo principal de la aplicación, se puede llamar a este método para asegurar que no se lanzará ninguna excepción. Se puede verificar si se está en el hilo principal de la aplicación o no usando el método ***isFxApplicationThread()***.

Ejercicio 13

Crea un proyecto llamado ***My3Counters***. Deberá tener 3 botones y 3 etiquetas:

- Un botón con el texto *"From 1 to 10"* que iniciará un hilo que cuenta del 1 al 10, mostrando el número actual en la etiqueta correspondiente, y durmiendo 1 segundo después de mostrar cada número.
- Un botón con el texto *"From 1 to 5"*, con su etiqueta correspondiente, para contar del 1 al 5 (un número por segundo también).
- Un botón con el texto *"From 10 to 1"*, con su etiqueta correspondiente, para contar de 10 a 1 (un número por segundo también).

Tan pronto como se haga clic en un botón, comenzará su conteo correspondiente, y el botón se deshabilitará (utiliza el método *setDisable()* del objeto *Button*). Se podrán ejecutar las tres tareas al mismo tiempo si se desea. Aquí puedes ver una captura de pantalla de la aplicación.



6.2. El framework de concurrencia de JavaFX

El ejemplo mostrado anteriormente solo usa los conceptos básicos de una aplicación JavaFX y los conceptos básicos del manejo de hilos aprendidos hasta ahora, y los combina para crear una aplicación gráfica multi-hilo. Sin embargo, esta no es la forma "correcta" de crear este tipo de aplicaciones ya que, en cuanto la aplicación se vuelva cada vez más complicada, estos métodos básicos explicados (como *Platform.runLater()*) no serán suficiente.

Para crear de forma más sólida aplicaciones multi-hilo, JavaFX proporciona un *framework* de concurrencia compuesto por los siguientes elementos:

- El interfaz ***Worker***. Representa todas las tareas que deben realizarse en uno o más hilos adicionales. Tiene una enumeración interna llamada ***Worker.State***, con todos los estados posibles de la tarea (*READY*, *RUNNING*, *CANCELLED*, etc).
- La clase abstracta ***Task*** implementa la interfaz *Worker* para definir tareas que se pueden ejecutar solo una vez (no se pueden reutilizar).
- La clase abstracta ***Service*** también implementa la interfaz *Worker* para definir tareas que se pueden ejecutar más de una vez (se pueden reutilizar).
- La clase abstracta ***ScheduledService*** es un subtipo de la clase *Service* para definir tareas que pueden programarse para que se ejecuten repetidamente después de un intervalo de tiempo determinado.
- ***WorkerStateEvent*** es un evento que se dispara cada vez que el estado de un *Worker* cambie, de forma que se puedan ejecutar algunas instrucciones o métodos cuando esto suceda.

6.2.1. Uso de *Service*

A continuación, se verá un ejemplo de creación de un *Service* y su ejecución en segundo plano. Se solucionará el mismo problema mostrado en el ejemplo anterior (la simulación de una copia de archivo) con un *service*. En este nuevo ejemplo, se añadirá la posibilidad de cancelar la copia mientras esté ejecutándose, una propiedad esencial de la clase *Service*. La clase de servicio se vería así:

```
public class FileService extends Service<String> {

    @Override
    protected Task<String> createTask() {
        return new Task<String>() {
            @Override
            protected String call() throws Exception {
                for (int progress = 1; progress <= 10; progress++) {
                    try {
                        Thread.sleep(1000);
                        updateMessage("" + (progress * 10) + "% completed");
                    } catch (Exception ex) { }
                }
                return "Copy completed";
            }
        };
    }
}
```

Solo se extiende de la clase *Service*. Se puede parametrizar para establecer un tipo de retorno y, en este caso, se devolverá un *String* cuando finaliza el servicio, solo para que veas cómo funciona este valor de retorno. Se debe sobrecargar el método ***createTask()*** de la clase abstracta *Service*. Dentro de este método se crea una clase anónima de la tarea que se va a lanzar (debido a que *Task* es una clase abstracta también, es necesario crear una subclase *Task* o devolver una *Task* a través de una clase anónima). En el método *call()* de esta tarea, se realiza el trabajo (la simulación de copia de archivo). Observa que, en lugar de obtener la etiqueta y establecer su texto, simplemente se llama al método *updateMessage()*. En la aplicación principal se enlazará el texto de la etiqueta con este mensaje para actualizar el texto automáticamente.

El controlador principal de JavaFX sería así:

```
public class FXServiceExampleController {
    @FXML
    private Button btnStart;
    @FXML
    private Label lblProgress;
    @FXML
    private Button btnCancel;

    FileService service;

    @FXML
    private void start(ActionEvent event) {
        setProperties(true, false);
        service.start();
    }
```

```
    @FXML
    private void cancel(ActionEvent event) {
        setProperties(false, true);
        service.cancel();
    }

    public void initialize() {
        service = new FileService();
        // Events to be fired when service finishes/cancels/fails...
        service.setOnSucceeded(e -> {
            setProperties(false, true);
            System.out.println(service.getValue());
            service.reset();
        });
        service.setOnCancelled(e -> {
            setProperties(false, true);
            service.reset();
        });
        service.setOnFailed(e -> {
            setProperties(false, true);
            service.reset();
        });

        //  Bind label text property to service
        lblProgress.textProperty().bind(service.messageProperty());
        btnCancel.setDisable(true);
    }

    // Method to disable/enable buttons and set label's text from events
    private void setProperties(boolean disableStart, boolean disableCancel) {
        btnStart.setDisable(disableStart);
        btnCancel.setDisable(disableCancel);
    }
}
```

Desde el método *start()* se han introducido algunas nuevas instrucciones interesantes:

- Los métodos de *Service* para iniciarlo, cancelarlo o resetearlo, según el evento que se esté manejando. Se inicia el servicio desde el evento del botón *Start*, se cancela desde el evento del botón *Cancel*, y se reiniciará desde *WorkerStateEvents*, controlados por los métodos *setOnSucceeded()*, *setOnCancelled()* y *setOnFailed()*. Por ejemplo, cada vez que se cancela el servicio, el evento *setOnCancelled* se activará y luego se reiniciará el servicio.
- La **vinculación** de la propiedad texto de la etiqueta hasta la propiedad del mensaje de servicio, se estable en esta línea:

  ```
  lblProgress.textProperty().bind(service.messageProperty());
  ```

  Gracias a esto, el servicio puede actualizar el texto de la etiqueta a partir de su código. Si ejecutas el ejemplo, observa que, tan pronto como el servicio finaliza o se cancela, la etiqueta de texto se vacía.

  El servicio también tiene algunas otras propiedades, como *titleProperty*, *valueProperty*, ... que se pueden usar para vincularlos a otros controles de la aplicación, si quieres. Esto es útil cuando quieres actualizar varios controles de un mismo servicio.

Sin embargo, si vinculas un control a una propiedad, el valor del control no se puede establecer fuera de esta propiedad. En otras palabras, si quieres establecer el texto de la etiqueta directamente (con su método *setText()*) en la aplicación principal, se lanzará una excepción. Es necesario desvincular temporalmente el control (con el método *unbind()*), para establecer el valor y vincularlo nuevamente a la propiedad.

- El valor de retorno del servicio se utiliza dentro del método *setOnSucceeded()*. Cuando el servicio finalice correctamente, devolverá un *String* con el texto *"Copy completed"*. Puedes comprobar esto en la salida estándar gracias a la siguiente línea de código:

```
System.out.println(service.getValue());
```

El método *setProperties()* se utiliza en algunos eventos para actualizar el estado *"disable"* de ambos botones (cuando se inicia la copia, se deshabilita el botón *Start*, por ejemplo) y el texto de la etiqueta.

Ejercicio 14

Crea un proyecto llamado ***My3CountersService***, que será una copia del proyecto *My3Counters* del ejercicio 13. En este caso, debes utilizar un *Servicie* para implementar las 3 tareas. Tan pronto como finalice un recuento dado, el botón correspondiente debe volver a habilitarse y se podrá volver a iniciar.

AYUDA: Debes implementar un *Servicie void*. Como *Service* es una clase parametrizada, cuando quieras que devuelva un resultado *void*, debes usar el parámetro <*Void*>. En el método *"call"*, debes devolver un tipo *Void*, y puedes hacerlo utilizando una instrucción *"return null"* al final del método.

6.2.2. Uso de *ScheduledService*

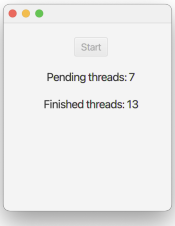
Si se quiere ejecutar una tarea periódicamente utilizando un *Servicie*, ***ScheduledService*** es más adecuado para ese trabajo que usar un bucle con períodos de inactividad dentro de un servicio. Este tipo de ejecutor de tareas es muy similar a lo que acabas de ver, pero se repite automáticamente después de un período de tiempo hasta que se cancele.

Se puede establecer un retraso antes de la primera llamada de ejecución llamado a ***setDelay()***. El tiempo que esperará antes de comenzar de nuevo después de terminar se establece con ***setPeriod()***. Si todo va bien, cada vez que termine este servicio la tarea, llamará a la función pasada ***setOnSucceeded()***.

```
public class FXServiceExampleController {
    @FXML
    private Button button;
    @FXML
    private Label threadsPending;
    @FXML

    private Label threadsFinished;
    private ScheduledService<Boolean> schedServ;
    private ThreadPoolExecutor executor;
```

```
    public void initialize() {
        schedServ = new ScheduledService<Boolean>() {
            @Override
            protected Task<Boolean> createTask() {
                return new Task<Boolean>() {
                    @Override
                    protected Boolean call() throws Exception {
                        Platform.runLater(() -> {
                            threadsPending.setText("Pending threads: " +
                                    (executor.getTaskCount() -
                                            executor.getCompletedTaskCount()));
                            threadsFinished.setText("Finished threads: " +
                                    executor.getCompletedTaskCount());
                        });
                        return executor.isTerminated();
                    }
                };
            }
        };

        schedServ.setDelay(Duration.millis(500)); // Will start after 0.5s
        schedServ.setPeriod(Duration.seconds(1)); // Runs every second after

        schedServ.setOnSucceeded(e -> {
            if (schedServ.getValue()) {
                // Executor finished
                schedServ.cancel(); // Cancel service (stop it).
                button.setDisable(false);
            }
        });
    }

    @FXML
    private void startThreads(ActionEvent event) {
        button.setDisable(true);
        executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(
                Runtime.getRuntime().availableProcessors()
        );

        for (int i = 0; i < 20; i++) {
            executor.execute(() -> {
                Random rnd = new Random();
                try {
                    TimeUnit.MILLISECONDS.sleep(500 + rnd.nextInt(5000));
                } catch (InterruptedException ex) {
                }
            });
        }

        executor.shutdown();
        schedServ.restart();// Start the scheduled service (or restart it)
    }
}
```

El código anterior iniciará un ejecutor con 20 tareas que tardarán entre 0,5 y 5,5 segundos en completarse. El *ScheduledService* se iniciará tras 0,5 segundos y se ejecutará cada segundo, examinando al ejecutor (mostrando cuántas tareas están pendientes y cuántas finalizadas) y regresando si el ejecutor ha terminado de ejecutar todas sus tareas. Cuando el ejecutor finalice, el *ScheduledService* se cancelará.



Ejercicio 15

Crea un proyecto llamado ***ScheduledChronometer***. Crea una vista con un *TextField* donde se escribirá un número de segundos y los botones ***Start*** y ***Pause***.

Cuando se presione el botón *Start*, se iniciará un ***ScheduledService*** que se lanzará cada segundo. Este servicio disminuirá el número (a partir del valor en *TextField*) y lo devolverá. Cuando llegue a 0, se detendrá (cancelará).

Si el servicio se está ejecutando y presionas *Pause*, se detendrá, el texto del botón cambiará por *"Resume"*. Si presionas *Resume* de nuevo, comenzará desde el último valor y el texto del botón volverá a cambiar por *"Pause"*.