Concurrency in JavaFX

This article describes the capabilities provided by the `javafx.concurrent` package to create multithreaded applications. You learn how to keep your JavaFX application user interface (UI) responsive by delegating time-consuming task execution to background threads.

Why Use the javafx.concurrent Package?

The JavaFX scene graph, which represents the graphical user interface of a JavaFX application, is not thread-safe and can only be accessed and modified from the UI thread also known as the JavaFX Application thread. Implementing long-running tasks on the JavaFX Application thread inevitably makes an application UI unresponsive. A best practice is to do these tasks on one or more background threads and let the JavaFX Application thread process user events.

If you implement a background worker by creating a Runnable object and a new thread, at some point, you must communicate with the JavaFX Application thread, either with a result or with the progress of the background task, which is error prone. Instead, use the JavaFX APIs provided by the `javafx.concurrent` package, which takes care of multithreaded code that interacts with the UI and ensures that this interaction happens on the correct thread.

Overview of the javafx.concurrent Package

The Java platform provides a complete set of concurrency libraries available through the `java.util.concurrent` package. The `javafx.concurrent` package leverages the existing API by considering the JavaFX Application thread and other constraints faced by GUI developers.

The `javafx.concurrent` package consists of the `Worker` interface and two basic classes, `Task` and `Service`, both of which implement the `Worker` interface. The `Worker` interface provides APIs that are useful for a background worker to communicate with the UI. The `Task` class is a fully observable implementation of the `java.util.concurrent.FutureTask` class. The `Task` class enables developers to implement asynchronous tasks in JavaFX applications. The `Service` class executes tasks.

The `WorkerStateEvent` class specifies an event that occurs whenever the state of a Worker implementation changes. Both the `Task` and `Service` classes implement the `EventTarget` interface and thus support listening to the state events.

The Worker Interface

The `Worker` interface defines an object that performs some work on one or more background threads. The state of the Worker object is observable and usable from the JavaFX Application thread.

The lifecycle of the Worker object is defined as follows. When created, the Worker object is in the `READY` state. Upon being scheduled for work, the Worker object transitions to the `SCHEDULED` state. After that, when the Worker object is performing the work, its state becomes `RUNNING`. Note that even when the Worker object is immediately started without being scheduled, it first transitions to the `SCHEDULED` state and then to the `RUNNING` state. The state of a Worker object that completes successfully is `SUCCEEDED`, and the `value` property is set to the result of this Worker object. Otherwise, if any exceptions are thrown during the execution of the Worker object, its state becomes `FAILED` and the `exception` property is set to the type of the exception that occurred. At any time before the end of the Worker object the developer can interrupt it by invoking the `cancel` method, which puts the Worker object into the `CANCELLED` state.

Release: JavaFX 2.1

Last Updated: June 2012
Download as PDF

[+] Show/Hide Table of Contents

Profiles

Irina Fedortsova
Technical Writer, Oracle

Irina has written tutorials and technical articles on Java and JavaFX technologies. She lives in St. Petersburg, Russia. In her spare time, she enjoys swing dancing, playing piano, and reading.

We Welcome Your Comments

Send us feedback about this document.

If you have questions about JavaFX, please go to the forum.

The progress of the work being done by the Worker object can be obtained through three different properties such as `totalWork`, `workDone`, and `progress`.

For more information on the range of the parameter values, see the API documentation.

The Task Class

Tasks are used to implement the logic of work that needs to be done on a background thread. First, you need to extend the `Task` class. Your implementation of the `Task` class must override the `call` method to do the background work and return the result.

The `call` method is invoked on the background thread, therefore this method can only manipulate states that are safe to read and write from a background thread. For example, manipulating an active scene graph from the `call` method throws runtime exceptions. On the other hand, the `Task` class is designed to be used with JavaFX GUI applications, and it ensures that any changes to public properties, change notifications for errors or cancellation, event handlers, and states occur on the JavaFX Application thread. Inside the `call` method, you can use the `updateProgress`, `updateMessage`, `updateTitle` methods, which update the values of the corresponding properties on the JavaFX Application thread. However, if the task was canceled, a return value from the `call` method is ignored.

Note that the `Task` class fits into the Java concurrency libraries because it inherits from the `java.utils.concurrent.FutureTask` class, which implements the `Runnable` interface. For this reason, a Task object can be used within the Java concurrency `Executor` API and also can be passed to a thread as a parameter. You can call the Task object directly by using the `FutureTask.run()` method, which enables calling this task from another background thread. Having a good understanding of the Java concurrency API will help you understand concurrency in JavaFX.

A task can be started in one of the following ways:

- By starting a thread with the given task as a parameter:

  ```
  Thread th = new Thread(task);
  th.setDaemon(true);
  th.start();
  ```

- By using the `ExecutorService` API:

  ```
  ExecutorService.submit(task);
  ```

The `Task` class defines a one-time object that cannot be reused. If you need a reusable Worker object, use the `Service` class.

Cancelling the Task

There is no reliable way in Java to stop a thread in process. However, the task must stop processing whenever `cancel` is called on the task. The task is supposed to check periodically during its work whether it was cancelled by using the `isCancelled` method within the body of the `call` method. Example 1 shows a correct implementation of the `Task` class that checks for cancellation.

Example 1

```
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {
    @Override protected Integer call() throws Exception {
        int iterations;
        for (iterations = 0; iterations < 100000; iterations++) {
            if (isCancelled()) {
                break;
            }
            System.out.println("Iteration " + iterations);
        }
        return iterations;
```

```
    }
};
```

If the task implementation has blocking calls such as `Thread.sleep` and the task is cancelled while in a blocking call, an `InterruptedException` is thrown. For these tasks, an interrupted thread may be the signal for a cancelled task. Therefore, tasks that have blocking calls must double-check the `isCancelled` method to ensure that the `InterruptedException` was thrown due to the cancellation of the task as shown in Example 2.

Example 2

```
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {
    @Override protected Integer call() throws Exception {
        int iterations;
        for (iterations = 0; iterations < 1000; iterations++) {
            if (isCancelled()) {
                updateMessage("Cancelled");
                break;
            }
            updateMessage("Iteration " + iterations);
            updateProgress(iterations, 1000);

            //Block the thread for a short time, but be sure
            //to check the InterruptedException for cancellation
            try {
                Thread.sleep(100);
            } catch (InterruptedException interrupted) {
                if (isCancelled()) {
                    updateMessage("Cancelled");
                    break;
                }
            }
        }
        return iterations;
    }
};
```

Showing the Progress of a Background Task

A typical use case in multithreaded applications is showing the progress of a background task. Suppose you have a background task that counts from one to one million and a progress bar, and you must update the progress on this progress bar as the counter runs in the background. Example 3 shows how to update a progress bar.

Example 3

```
import javafx.concurrent.Task;

Task task = new Task<Void>() {
    @Override public Void call() {
        static final int max = 1000000;
        for (int i=1; i<=max; i++) {
            if (isCancelled()) {
               break;
            }
            updateProgress(i, max);
        }
        return null;
    }
};
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```

First, you create the task by overriding the `call` method where you implement the logic of the work to be done and invoke the `updateProgress` method, which updates the `progress`, `totalWork`, and `workDone` properties of the task. This is important because you can now use the `progressProperty` method to retrieve the progress of the task and bind the progress of the bar to the progress of the task.

The Service Class

The `Service` class is designed to execute a Task object on one or several background threads. The `Service` class methods and states must only be accessed on the JavaFX Application thread. The purpose of this class is to help the developer to implement the correct interaction between the background threads and the JavaFX Application thread.

You have the following control over the Service object: you can start, cancel and restart it as you need. To start the Service object, use the `Service.start()` method.

Using the `Service` class, you can observe the state of the background work and optionally cancel it. Later, you can reset the service and restart it. Thus, the service can be defined declaratively and restarted on demand.

When implementing the subclasses of the `Service` class, be sure to expose the input parameters to the Task object as properties of the subclass.

The service can be executed in one of the following ways:

- By an Executor object, if it is specified for the given service
- By a daemon thread, if no executor is specified
- By a custom executor such as a ThreadPoolExecutor

Example 4 shows an implementation of the `Service` class which reads the first line from any URL and returns it as a string.

Example 4

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.concurrent.Service;
import javafx.concurrent.Task;
import javafx.concurrent.WorkerStateEvent;
import javafx.event.EventHandler;
import javafx.stage.Stage;

public class FirstLineServiceApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        FirstLineService service = new FirstLineService();
        service.setUrl("http://google.com");
        service.setOnSucceeded(new EventHandler<WorkerStateEvent>() {

            @Override
            public void handle(WorkerStateEvent t) {
                System.out.println("done:" + t.getSource().getValue());
            }
        });
        service.start();
    }

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

```
    private static class FirstLineService extends Service<String> {
        private StringProperty url = new SimpleStringProperty();

        public final void setUrl(String value) {
            url.set(value);
        }

        public final String getUrl() {
            return url.get();
        }

        public final StringProperty urlProperty() {
           return url;
        }

        protected Task<String> createTask() {
            final String _url = getUrl();
            return new Task<String>() {
                protected String call()
                    throws IOException, MalformedURLException {
                        String result = null;
                        BufferedReader in = null;
                        try {
                            URL u = new URL(_url);
                            in = new BufferedReader(
                                new InputStreamReader(u.openStream()));
                            result = in.readLine();
                        } finally {
                            if (in != null) {
                                in.close();
                            }
                        }
                        return result;
                }
            };
        }
    }
}
```

The WorkerStateEvent Class and State Transitions

Whenever the state of the Worker implementation changes, an appropriate event, defined by the `WorkerStateEvent` class, occurs. For example, when the Task object transitions to the `SUCCEEDED` state, the `WORKER_STATE_SUCCEEDED` event occurs, the `onSucceeded` event handler is called, after which the protected convenience method `succeeded` is invoked on the JavaFX Application thread.

There are several protected convenience methods such as `cancelled`, `failed`, `running`, `scheduled`, and `succeeded`, which are invoked when the Worker implementation transitions to the corresponding state. These methods can be overridden by subclasses of the `Task` and `Service` classes when the state is changed to implement the logic of your application. Example 5 shows a Task implementation that updates the status message on the task's success, cancellation, and failure.

Example 5

```
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {
    @Override protected Integer call() throws Exception {
        int iterations = 0;
        for (iterations = 0; iterations < 100000; iterations++) {
            if (isCancelled()) {
                break;
            }
            System.out.println("Iteration " + iterations);
        }
        return iterations;
    }
```

```
    @Override protected void succeeded() {
        super.succeeded();
        updateMessage("Done!");
    }

    @Override protected void cancelled() {
        super.cancelled();
        updateMessage("Cancelled!");
    }

@Override protected void failed() {
    super.failed();
    updateMessage("Failed!");
    }
};
```

Conclusion

In this article, you learned the basic capabilities provided by the `javafx.concurrent` package and became familiar with several examples of the `Task` and `Service` classes implementation. For more examples of how to create the Task implementation correctly, see the API documentation for the `Task` class.