# 8

# JavaFX Concurrency and Binding



ORACLE

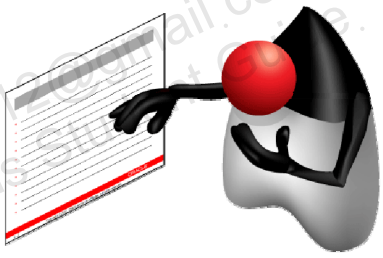
# Objectives

After completing this lesson, you should be able to:

- Describe and implement JavaFX concurrency
- Describe binding and properties in JavaFX, including simple binding and bi-directional binding

# Topics

- Working with JavaFX concurrency
- Binding in JavaFX



# Concurrency and JavaFX

The `javafx.concurrent` package handles multithreaded code that interacts correctly with the UI and ensures that this interaction happens on the correct thread.

- The `Worker` interface
  - **`Task`:** Is a fully observable implementation of the `java.util.concurrent.FutureTask` class; enables implementation of asynchronous tasks
  - **`Service`:** Executes tasks





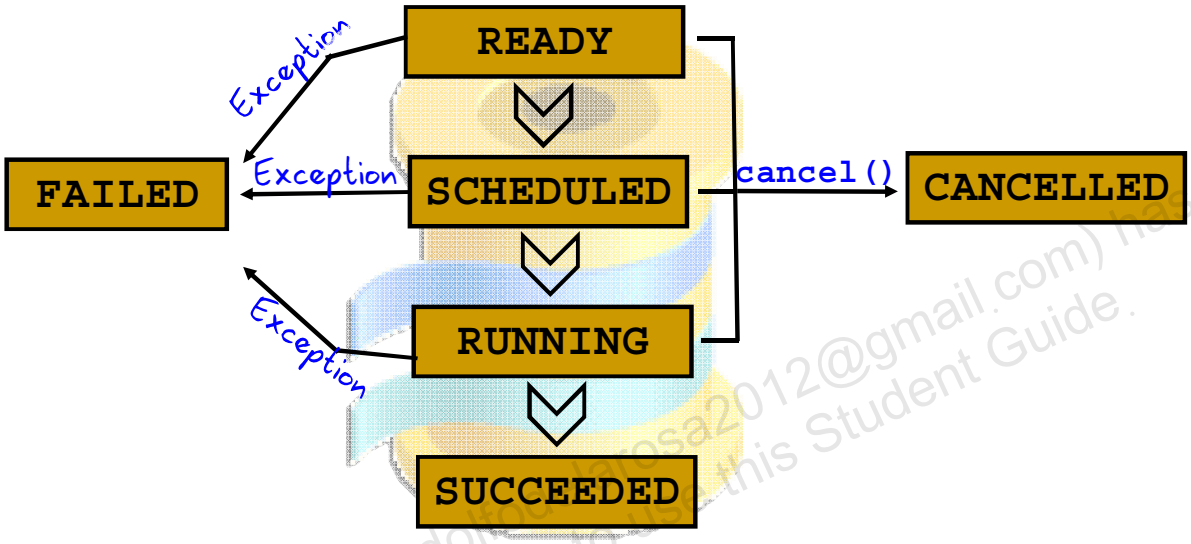


The Java platform provides a complete set of concurrency libraries available through the `java.util.concurrent` package. The `javafx.concurrent` package leverages the existing API by taking into account the JavaFX Application thread and other constraints faced by GUI developers.

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.

# Worker Interface

A `Worker` is an object that performs work in background threads. The `Worker` main life cycle includes `READY`, `SCHEDULED`, and `RUNNING`.




ORACLE







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

The life cycle of the Worker object is defined as follows. When created, the Worker object is in the READY state. After 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. In any state, the Worker object can be interrupted using the cancel method, which puts the Worker object in the CANCELLED state.

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 about the range of the parameter values, see the API documentation.

# `Task` Class

`Task` can be started in one of the following three ways (the first two listed are the preferred methods):

- Using the ExecutorService API: `ExecutorService.submit(task);`
- Using the run method: `task.run();`
- Starting a thread with the given task as a parameter: `new Thread(task).start();`

The `Task` class defines a thread and task that cannot be reused.

- You can call the `Task` object directly by using `FutureTask.run()`.
  - `public abstract class Task<V> extends java.util.concurrent.FutureTask<V> implements Worker<V>, EventTarget`







Tasks are used to implement the logic of work that needs to be done on a background thread. You first need to extend the `Task` class. Your implementation of the `Task` class must override the `call` method.

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.

# Example: Create the `Task`

```
import javafx.concurrent.Task;

public class CounterTask extends Task<Void>{

    @Override
    public Void call(){
        final int max = 10000000;
        updateProgress(0, max);
        for (int i=1; i<=max; i++){
            updateProgress(i, max);
        }

        return null;
    }

}
```

Overrides the `call()` method invoked on the background thread







In the example in the slide, the task is created by overriding the `call()` method. 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, 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.

The example is located in `D:\labs\08-FXConcurrency\examples\CounterBar`.

# Example: Run the `Task`

```
ExecutorService es = Executors.newSingleThreadExecutor();

. . .

  @Override
  public void handle(ActionEvent event) {
      System.out.println("Count Started");
      bar.progressProperty().bind(
                        countTask.progressProperty());
      es.execute(countTask);
  }
});
```

ExecutorService



In the example in the slide, the task is executed by `ExecutorService`.

The example is located in `D:\labs\08-FXConcurrency\examples\CounterBar`.

# `Service` Class

- The `Service` class is designed to execute a `Task` object on one or several background threads.
- `Service` class methods and states must only be accessed on the JavaFX application thread.
- It helps developers implement correct interactions between background threads and the JavaFX Application thread.
- You can start, stop, cancel, and restart a `Service` as needed.
- Use `Service.start()` to start a `Service`.
- `Service` can run a task more than once.







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.

# `Service` Class Execution

The `Service` can be executed in one of the following three 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`







If a `java.util.concurrent.Executor` is specified on the Service, it will be used to actually execute the background worker. Otherwise, a daemon thread will be created and executed. If you want to create non-daemon threads, specify a custom `Executor` (for example, you could use a `ThreadPoolExecutor`).

# Example: Create the `Service`

```
public class CounterService extends Service<Void>{
   //Create an instance of Task that returns the CounterTask.
    @Override
    protected Task<Void> createTask(){
        CounterTask ct = new CounterTask();
        return ct;
    }

}
```







In the example in the slide, the `Service` is built and creates an instance of `Task` that returns `CounterTask`.

The example is located in `D:\labs\08-FXConcurrency\examples`.

# Example: Run the `Service`

```
CounterService cs = new CounterService();

...

@Override
  public void handle(ActionEvent event) {
     System.out.println("Count Started");

     bar.progressProperty().bind(cs.progressProperty());
     if (cs.getState() == State.READY){
         cs.start();
     }
  }
});
```

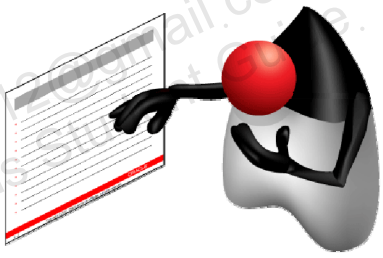
Starts the service



In the example in the slide, the Service is started by the `start()` method.
The example is located in `D:\labs\08-FXConcurrency\examples`.

# Topics

- Working with JavaFX concurrency
- Binding in JavaFX



# Data Binding in JavaFX

- Simplifies the task of synchronizing the view state and domain data model
- Enables enterprise design patterns
- Enables rapid application development in visual tools
- A binding observes its list of dependencies for changes, and then updates itself automatically after a change has been detected.
- The binding API uses a High-Level API that provides a simple way to create bindings for the most common use cases.





ORACLE



Binding is a powerful mechanism for expressing direct relationships between variables. When objects participate in bindings, changes made to one object will automatically be reflected in another object. This can be useful in a variety of applications. For example, binding can be used in a bill invoice tracking program, where the total of all bills is automatically updated whenever an individual bill is changed; or, binding can be used in a GUI that automatically keeps its display synchronized with the application's underlying data.

Data binding lends itself to enterprise design patterns such as the Presentation Model and Supervising Controller, which are patterns similar to the MVC pattern. The Supervising Controller leverages the data-binding facilities of the view framework, thus leading to fewer methods in the view that need to be mocked up.

Bindings are assembled from one or more sources called *dependencies*. A binding observes its list of dependencies for changes, and then updates itself automatically after a change has been detected.

The High-Level API provides a simple way to create bindings for the most common use cases. Its syntax is easy to learn and use, especially in environments that provide code completion, such as the NetBeans IDE. It works for functions such as arithmetic operations, boolean operations, calculating minimum and maximum values, comparisons, and so on.

# Data Binding in JavaFX

In the following example, `bind` is used to tie the progress bar to the `cs` (CounterService).

```
CounterService cs = new CounterService();

...

@Override
  public void handle(ActionEvent event) {
     System.out.println("Count Started");
     bar.progressProperty().bind(cs.progressProperty());
     if (cs.getState() == State.READY){
         cs.start();
     }
  }
});
```

Bind to the cs.progressProperty



In the example in the slide, `bind` is used to tie the progress bar to the counter service. As the counter service is invoked, the progress bar is invoked.

The example is located in `D:\labs\08-FXConcurrency\examples\CounterBar`.

# Quiz

Which of the following is an interface that receives notifications of changes to an `ObservableMap`?

a. `ObservableMap`
b. `javafx.collections`
c. `MapChangeListener`
d. `ObservableList`







**Answer: c**

# Summary

In this lesson, you should have learned how to:

- Describe and implement JavaFX concurrency
- Describe binding and properties in JavaFX, including simple binding and bi-directional binding

ORACLE

# Practice 8: Overview

- Practice 8-1: Displaying Service State Information
- Practice 8-2: Adding a Second Service to Your Application