23.11 Multithreading in JavaFX

JavaFX applications present a unique set of challenges for multithreaded programming. All JavaFX applications have a single thread, called the **JavaFX application thread**, to handle interactions with the application's controls. Typical interactions include *rendering controls* or *processing user actions* such as mouse clicks. All tasks that require interaction with an application's GUI are placed in an *event queue* and are executed sequentially by the JavaFX application thread.

JavaFX's scene graph is not thread safe—its nodes cannot be manipulated by multiple threads without the risk of incorrect results that might corrupt the scene graph. Unlike the other examples presented in this chapter, thread safety in JavaFX applications is achieved not by synchronizing thread actions, but by *ensuring that programs manipulate the scene graph from only the JavaFX application thread*. This technique is called **thread confinement**. Allowing just one thread to access non-thread-safe objects eliminates the possibility of corruption due to multiple threads accessing these objects concurrently.

It's acceptable to perform brief tasks on the JavaFX application thread in sequence with GUI component manipulations—like calculating tips and totals in Chapter 12's **Tip Calculator** app. If an application must perform a lengthy

task in response to a user interaction, the JavaFX application thread cannot render controls or respond to events while the thread is tied up in that task. This causes the GUI to become unresponsive. It's preferable to handle long-running tasks in separate threads, freeing the JavaFX application thread to continue managing other GUI interactions. Of course, you must update the GUI with the computation's results from the JavaFX application thread, rather than from the worker thread that performed the computation.3

3. Like JavaFX, Swing uses a single thread for handling interactions and displaying the GUI. Swing's similar capabilities to the JavaFX concurrency features discussed in this section are located in package `javax.swing`. Class `SwingUtilities` method `invokeLater` schedules a `Runnable` for later execution in the so-called event-dispatch thread. Class `SwingWorker` performs a long-running task in a worker thread and can display results in the GUI from the event-dispatch thread.

`Platform` Method `runLater`

JavaFX provides multiple mechanisms for updating the GUI from other threads. One is to call the `static` method `runLater` of class `Platform` (package `javafx.application`). This method receives a `Runnable` and schedules it on the JavaFX application thread for execution at some point in the future. Such `Runnable`s should perform only small updates, so the GUI remains responsive.

Class `Task` and Interface `Worker`

Long-running or compute-intensive tasks should be performed on separate worker threads, not the JavaFX application thread. Package `javafx.concurrent` provides interface `Worker` and classes `Task` and `ScheduledService` for this purpose:

- A `Worker` is a task that should execute using one or more separate threads.
- Class `Task` is a `Worker` implementation that enables you to perform a task (such as a long-running computation) in a worker thread and update the GUI from the JavaFX application thread based on the task's results. `Task` implements several interfaces, including `Runnable`, so a `Task` object can be scheduled to execute in a separate thread. Class `Task` also provides methods that are guaranteed to update its properties in the JavaFX application thread—as you'll see, this enables programs to bind a `Task`'s properties to GUI controls for automatic updating. Once a `Task` completes, it cannot be restarted—performing the `Task` again requires a new `Task` object. We'll demonstrate `Task`s in the next two examples.
- Class `ScheduledService` is a `Worker` implementation that creates and manages a `Task`. Unlike a `Task`, a `ScheduledService` can be reset and restarted. It also can be configured to automatically restart both after successful completion and if it fails due to an exception.

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers

In the next example, the user enters a number *n* and the program gets the *n*th Fibonacci number, which we calculate using the recursive algorithm discussed in Section 18.5. Since the algorithm is time consuming for large values, we use a `Task` object to perform the recursive calculation in a worker thread. The GUI also provides a separate set of components that displays the next Fibonacci number in the sequence with each click of a `Button`. This set of components performs its short computation directly in the event dispatch thread. The program is capable of producing up to the 92nd Fibonacci number—subsequent values are outside the range that can be represented by a `long`. Recall that you can use class `BigInteger` to represent arbitrarily large integer values.

Creating a `Task`

Class `FibonacciTask` (Fig. 23.23) extends `Task<Long>` (line 5) to perform the recursive Fibonacci calculation in a *worker thread*. The instance variable `n` (line 6) represents the Fibonacci number to calculate. Overridden `Task` method `call` (lines 14–20) computes the *n*th Fibonacci number and returns the result. The `Task`'s type parameter `Long` (line 5) determines `call`'s return type (line 15). Inherited `Task` method `updateMessage` (called in lines 16 and 18) updates the `Task`'s `message` property in the *JavaFX application thread*. As you'll see in Fig. 23.25, this enables us to bind JavaFX controls to `FibonacciTask`'s `message` property to display messages while the task is executing.

```
1   // Fig. 23.23: FibonacciTask.java
2   // Task subclass for calculating Fibonacci numbe
3   import javafx.concurrent.Task;
4
5   public class FibonacciTask extends Task<Long>{
6      private final int n; // Fibonacci number to c
7
8      // constructor
9      public FibonacciTask(int n) {
10        this.n = n;
11     }
12
13     // long-running code to be run in a worker th
14     @Override
15     protected Long call() {
16        updateMessage("Calculating…");
17        long result = fibonacci(n);
18        updateMessage("Done calculating.");
19        return result;
20     }
21
22     // recursive method fibonacci; calculates nth
23     public long fibonacci(long number) {
24        if (number == 0 || number == 1) {
25           return number;
26        }
27        else {
28           return fibonacci(number - 1) + fibonacc
29        }
30     }
31  }
```

Fig. 23.23

`Task` subclass for calculating Fibonacci numbers in the

background.

When the worker thread enters the *running* state, `FibonacciTask`'s `call` method begins executing. First, line 16 calls the inherited `updateMessage` method to update the `FibonacciTask`'s `message` property, indicating that the task is calculating. Next, line 17 invokes recursive method `fibonacci` (lines 23–30) with instance variable `n`'s value as the argument. When `fibonacci` returns, line 18 updates `FibonacciTask`'s `message` property again to indicate that the calculation completed, then line 19 returns the result to the JavaFX application thread.

`FibonacciNumbers` GUI

Figure 23.24 shows the app's GUI (defined in `FibonacciNumbers.fxml`) labeled with its **fx:id**s. Here we point out only the key elements and the event-handling methods you'll see in class `FibonacciNumbersController` (Fig. 23.25). For the complete layout details, open `FibonacciNumbers.fxml` in Scene Builder. The GUI's primary layout is a `VBox` containing two `TitledPane`s. The controller class defines two event handlers:

- `goButtonPressed` is called when the **Go** `Button` is pressed—this launches the worker thread to calculate a Fibonacci number recursively.
- `nextNumberButtonPressed` is called when the **Next Number** `Button` is pressed—this calculates the next Fibonacci number in the sequence. Initially the app displays the Fibonacci of 0 (which is 0).

We do not show the JavaFX `Application` subclass (located in `FibonacciNumbers.java`), because it performs the same tasks you've seen previously to load the app's FXML GUI and initialize the controller.

Class `FibonacciNumbersController`

Class `FibonacciNumbersController` (Fig. 23.25) displays a window containing two sets of controls:

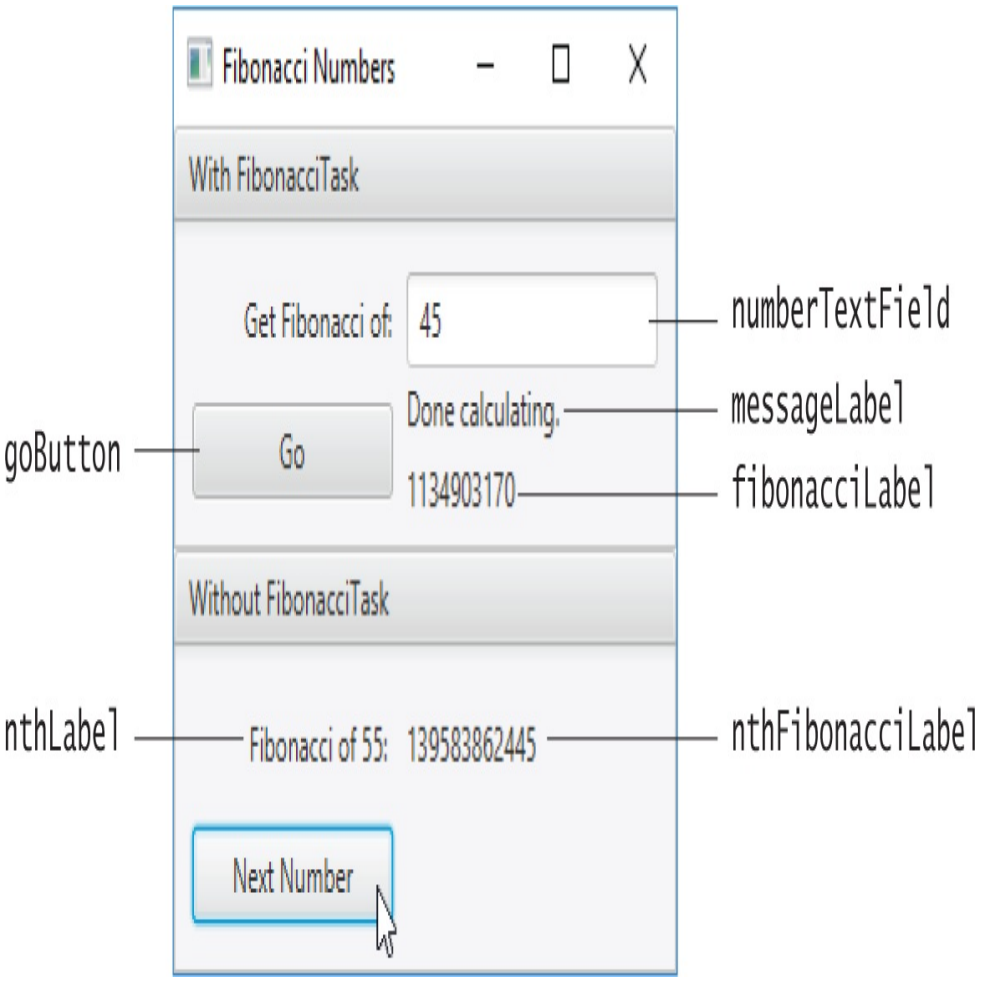


Fig. 23.24

`FibonacciNumbers` GUI with its **fx:id**s.

Description

- The **With FibonacciTask** `TitledPane` provides controls that enable the user to enter a Fibonacci number to calculate and launch a `FibonacciTask` in a worker thread. `Label`s in this `TitledPane` display the `FibonacciTask`'s message property value as it's updated and the final result of the `FibonacciTask` once it's available.
- The **Without FibonacciTask** `TitledPane` provides the **Next Number**

`Button` that enables the user to calculate the next Fibonacci number in sequence. `Label`s in this `TitledPane` display which Fibonacci number is being calculated (that is, "`Fibonacci of` *n*") and the corresponding Fibonacci value.

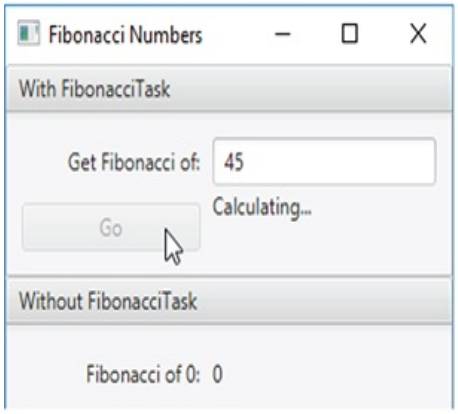
Instance variables `n1` and `n2` (lines 20–21) contain the previous two Fibonacci numbers in the sequence and are initialized to `0` and `1`, respectively. Instance variable `number` (initialized to `1` in line 22) keeps track of which Fibonacci value will be calculated and displayed next when the user clicks the **Next Number** `Button`—so the first time this `Button` is clicked, the Fibonacci of 1 is displayed.

```
 1   // Fig. 23.25: FibonacciNumbersController.java
 2   // Using a Task to perform a long calculation
 3   // outside the JavaFX application thread.
 4   import java.util.concurrent.Executors;
 5   import java.util.concurrent.ExecutorService;
 6   import javafx.event.ActionEvent;
 7   import javafx.fxml.FXML;
 8   import javafx.scene.control.Button;
 9   import javafx.scene.control.Label;
10   import javafx.scene.control.TextField;
11
12   public class FibonacciNumbersController {
13      @FXML private TextField numberTextField;
14      @FXML private Button goButton;
15      @FXML private Label messageLabel;
16      @FXML private Label fibonacciLabel;
17      @FXML private Label nthLabel;
18      @FXML private Label nthFibonacciLabel;
19
20      private long n1 = 0; // initialize with Fibon
21      private long n2 = 1; // initialize with Fibon
22      private int number = 1; // current Fibonacci
23
```

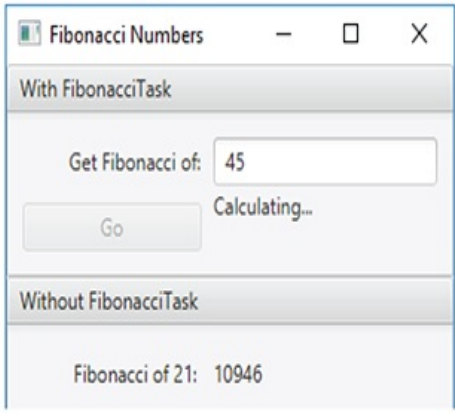
```
    // starts FibonacciTask to calculate in backg
    @FXML
    void goButtonPressed(ActionEvent event) {
        // get Fibonacci number to calculate
        try {
            int input = Integer.parseInt(numberText

            // create, configure and launch Fibonac
            FibonacciTask task = new FibonacciTask(

            // display task's messages in messageLa
            messageLabel.textProperty().bind(task.m

            // clear fibonacciLabel when task start
            task.setOnRunning((succeededEvent) -> {
                goButton.setDisable(true);
                fibonacciLabel.setText("");
            });

            // set fibonacciLabel when task complet
            task.setOnSucceeded((succeededEvent) ->
                fibonacciLabel.setText(task.getValue
                goButton.setDisable(false);
            });

            // create ExecutorService to manage thr
            ExecutorService executorService =
                Executors.newFixedThreadPool(1); //
            executorService.execute(task); // start
            executorService.shutdown();
        }
        catch (NumberFormatException e) {
            numberTextField.setText("Enter an integ
            numberTextField.selectAll();
            numberTextField.requestFocus();
        }
    }

    // calculates next Fibonacci value
    @FXML
```

```
64      void nextNumberButtonPressed(ActionEvent even
 65         // display the next Fibonacci number
66         nthLabel.setText("Fibonacci of " + number
67         nthFibonacciLabel.setText(String.valueOf(n
          68         long temp = n1 + n2;
                69         n1 = n2;
               70         n2 = temp;
                71         ++number;
                     72      }
                      73   }
```

a) Begin calculating Fibonacci of 45 in the background



b) Calculating other Fibonacci values while Fibonacci of 45 continues calculating



c) Fibonacci of 45 calculation finishes

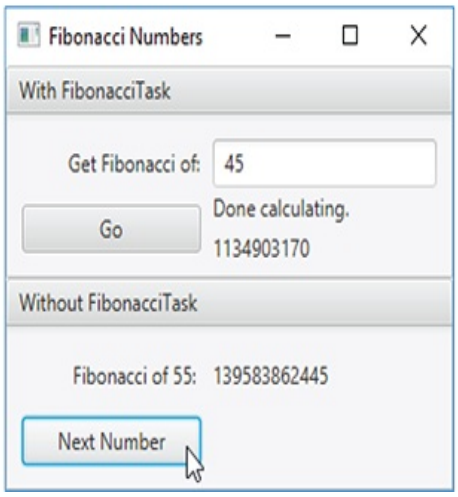


Fig. 23.25

Using a `Task` to perform a long calculation outside the JavaFX application thread.

Description

Method `goButtonPressed`

When the user clicks the **Go** `Button`, method `goButtonPressed` (lines 25–60) executes. Line 29 gets the value entered in the `numberTextField` and attempts to parse it as an integer. If this fails, lines 56–58 prompt the user to enter an integer, select the text in the `numberTextField` and give the `numberTextField` the focus, so the user can immediately enter a new value in the `numberTextField`.

Line 32 creates a new `FibonacciTask` object, passing the constructor the user-entered value. Line 35 binds the `messageLabel`'s `text` property (a `StringProperty`) to the `FibonacciTask`'s `message` property (a `ReadOnlyStringProperty`)—when `FibonacciTask` updates this property, the `messageLabel` displays the new value.

All `Worker`s transition through various states. Class `Task`—an implementation of `Worker`—enables you to register listeners for several of these states:

- Lines 38–41 use `Task` method `setOnRunning` to register a listener (implemented as a lambda) that's invoked when the `Task` enters the *running* state—that is, when the `Task` has been assigned a processor and begins executing its `call` method. In this case, we disable the `goButton` so the user cannot launch another `Fibonacci-Task` until the current one completes, then we clear the `fibonacciLabel` (so an old result is not displayed when a new `FibonacciTask` begins).
- Lines 44–47 use `Task` method `setOnSucceeded` to register a listener (implemented as a lambda) that's invoked when the `Task` enters the *succeeded* state—that is, when the `Task` successfully runs to completion. In this case, we call the `Task`'s `getValue` method (from interface `Worker`) to obtain the result, which we convert to a `String`, then display in the `fibonacciLabel`. Then we enable the `goButton` so the user can start a new `FibonacciTask`.

You also can register listeners for a `Task`'s *canceled*, *failed* and *scheduled* states.

Finally, lines 50–53 use an `ExecutorService` to launch the `FibonacciTask` (line 52), which schedules it for execution in a separate worker thread. Method `execute` does not wait for the `FibonacciTask` to finish executing. It returns immediately, allowing the GUI to continue processing other events while the computation is performed.

Method `nextNumberButtonPressed`

If the user clicks the **Next Number** `Button`, method

`nextNumberButtonPressed` (lines 63–72) executes. Lines 66–67 update the `nthLabel` to show which `Fibonacci` number is being displayed, then update `nthFibonacciLabel` to display `n2`'s value. Next, lines 68–71 add the previous two Fibonacci numbers stored in `n1` and `n2` to determine the next number in the sequence (which will be displayed on the next call to `nextNumberButtonPressed`), update `n1` and `n2` to their new values and increment `number`.

The code for these calculations is in method `nextNumberButtonPressed`, so they're performed on the *JavaFX application thread*. Handling such short computations in this thread does not cause the GUI to become unresponsive, as with the recursive algorithm for calculating the Fibonacci of a large number. Because the longer Fibonacci computation is performed in a separate worker thread, it's possible to click the **Next Number** `Button` to get the next Fibonacci number while the recursive computation is still in progress.

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes

Class `Task` provides additional methods and properties that enable you to update a GUI with a `Task`'s intermediate results as the `Task` continues executing in a `Worker` thread. The

next example uses the following `Task` methods and properties:

- Method `updateProgress` updates a `Task`'s `progress` property, which represents the percentage completion.
- Method `updateValue` updates a `Task`'s `value` property, which holds each intermediate value.

Like `updateMessage`, `updateProgress` and `updateValue` are guaranteed to update the corresponding properties in the JavaFX application thread.

A `Task` to Find Prime Numbers

Figure 23.26 presents class `PrimeCalculatorTask`, which extends `Task<Integer>` to compute the first *n* prime numbers in a *worker thread*. As you'll see in the `FindPrimesController` (Fig. 23.28), we'll bind this `Task`'s `progress` property to a `ProgressBar` control so the app can provide a visual indication of the portion of the `Task` that has been completed. The controller also registers a listener for the `value` property's changes—we'll store each prime value in an `ObservableList` that's bound to a `ListView`.

```
   1   // Fig. 23.26: PrimeCalculatorTask.java
2   // Calculates the first n primes, publishing the
          3   import java.util.Arrays;
```

```
4   import javafx.concurrent.Task;
5
6   public class PrimeCalculatorTask extends Task<In
7      private final boolean[] primes; // boolean ar
8
9      // constructor
10     public PrimeCalculatorTask(int max) {
11        primes = new boolean[max];
12        Arrays.fill(primes, true); // initialize a
13     }
14
15     // long-running code to be run in a worker th
16     @Override
17     protected Integer call() {
18        int count = 0; // the number of primes fou
19
20        // starting at index 2 (the first prime nu
21        // set to false elements with indices that
22        for (int i = 2; i < primes.length; i++) {
23           if (isCancelled()) { // if calculation
24              updateMessage("Cancelled");
25              return 0;
26           }
27           else {
28              try {
29                 Thread.sleep(10); // slow the thr
30              }
31              catch (InterruptedException ex) {
32                 updateMessage("Interrupted");
33                 return 0;
34              }
35
36              updateProgress(i + 1, primes.length)
37
38              if (primes[i]) { // i is prime
39                 ++count;
40                 updateMessage(String.format("Foun
41                 updateValue(i); // intermediate r
42
43                 // eliminate multiples of i
```

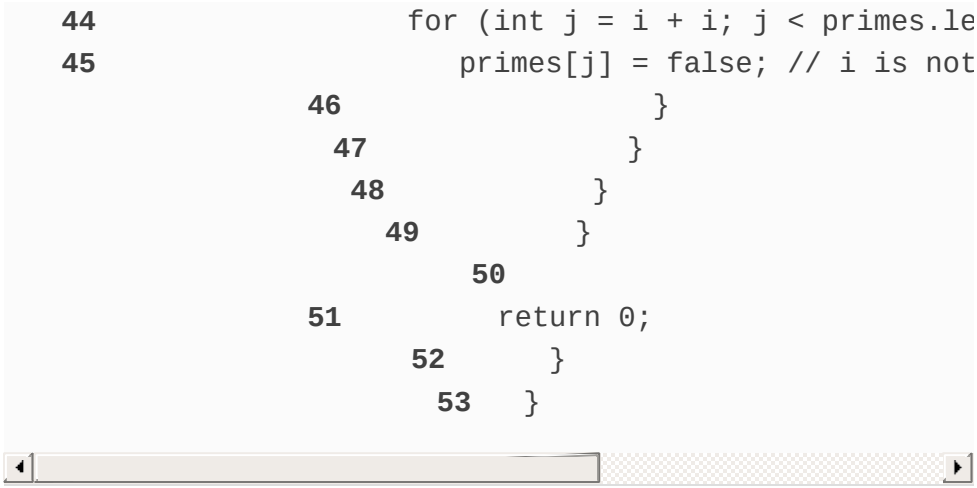


Fig. 23.26

Calculates the first *n* primes, publishing them as they are found.

Constructor

The constructor (lines 10–13) receives an integer indicating the upper limit of the prime numbers to locate, creates the `boolean` array `primes` and initializes its elements to `true`.

Sieve of Eratosthenes

`PrimeCalculatorTask` uses the `primes` array and the **Sieve of Eratosthenes** algorithm (described in Exercise 7.27) to find all primes less than `max`. The Sieve of Eratosthenes takes a list of integers and, beginning with the first prime

number, filters out all multiples of that prime. It then moves to the next prime, which will be the next number that's not yet filtered out, and eliminates all of its multiples. It continues until the end of the list is reached and all nonprimes have been filtered out. Algorithmically, we begin with element `2` of the `boolean` array and set the cells corresponding to all values that are multiples of 2 to `false` to indicate that they're divisible by 2 and thus not prime. We then move to the next array element, check whether it's `true`, and if so set all of its multiples to `false` to indicate that they're divisible by the current index. When the whole array has been traversed in this way, all indices that contain `true` are prime, as they have no divisors.

Overridden `Task` Method `call`

In method `call` (lines 16–52), the control variable `i` for the loop (lines 22–49) represents the current index for implementing the Sieve of Eratosthenes. Line 23 calls the inherited `Task` method `isCancelled` to determine whether the user has clicked the **Cancel** button. If so, line 24 updates the `Task`'s `message` property, then line 25 returns `0` to terminate the `Task` immediately.

If the calculation isn't canceled, line 29 puts the currently executing thread to sleep for 10 milliseconds. We discuss the reason for this shortly. Line 36 calls `Task`'s

`updateProgress` method to update the `progress` property in the JavaFX application thread. The percentage completion is determined by dividing the method's first argument by its second argument.

Next, line 38 tests whether the current `primes` element is `true` (and thus prime). If so, line 39 increments the `count` of prime numbers found so far and line 40 updates the `Task`'s `message` property with a `String` containing the `count`. Then, line 41 passes the index `i` to method `updateValue`, which updates `Task`'s `value` property in the JavaFX application thread—as you'll see in the controller, we process this *intermediate result* and display it the GUI. When the entire array has been traversed, line 51 returns `0`, which we ignore in the controller, because it is not a prime number.

Because the computation progresses quickly, publishing values often, updates can pile up on the JavaFX application thread, causing it to ignore some updates. This is why for demonstration purposes we put the worker thread to *sleep* for 10 milliseconds during each iteration of the loop. The calculation is slowed just enough to allow the JavaFX application thread to keep up with the updates and enable the GUI to remain responsive.

`FindPrimes` GUI

Figure 23.27 shows the app's GUI (defined in `FindPrimes.fxml`) labeled with its **fx:id**s. Here we point

out only the key elements and the event-handling methods you'll see in class `FindPrimesController` (Fig. 23.28). For the complete layout details, open `FindPrimes.fxml` in Scene Builder. The GUI's primary layout is a `BorderPane` containing two `ToolBar`s in the top and bottom areas. The controller class defines two event handlers:

- `getPrimesButtonPressed` is called when the **Get Primes** `Button` is pressed—this launches the worker thread to find prime numbers less than the value input by the user.
- `cancelButtonPressed` is called when the **Cancel** `Button` is pressed—this terminates the worker thread.

We do not show the JavaFX `Application` subclass (located in `FindPrimes.java`), because it performs the same tasks you've seen previously to load the app's FXML GUI and initialize the controller.

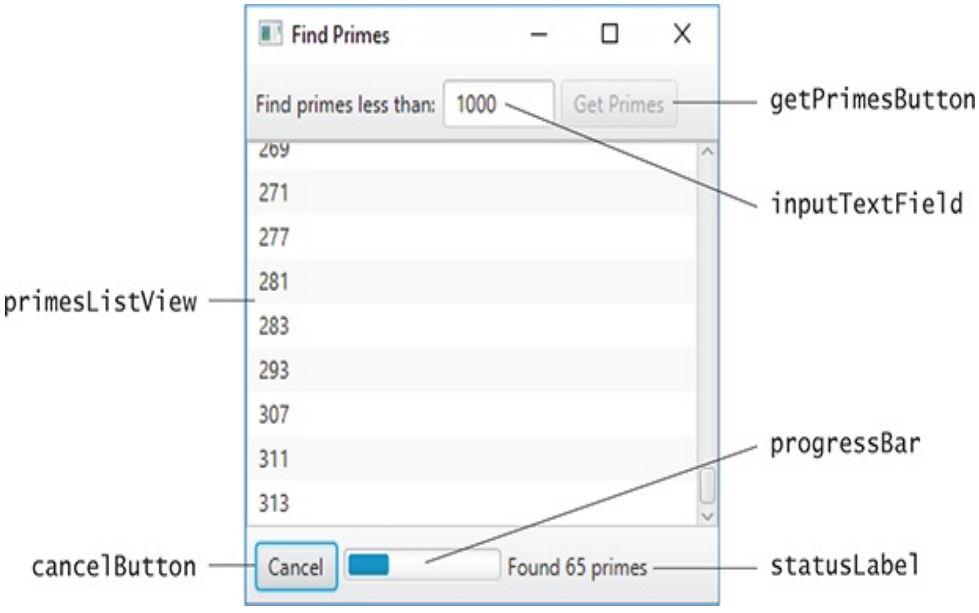


Fig. 23.27

`FindPrimes` GUI with its **fx:id**s.

Description

Class `FindPrimesController`

Class `FindPrimesController` (Fig. 23.28) creates an `ObservableList<Integer>` (lines 24–25) and binds it to the app's `primesListView` (line 30). The controller also provides the event handlers for the **Get Primes** and **Cancel** `Button`s.

```
 1  // Fig. 23.28: FindPrimesController.java
 2  // Displaying prime numbers as they're calculated
 3  import java.util.concurrent.Executors;
 4  import java.util.concurrent.ExecutorService;
 5  import javafx.collections.FXCollections;
 6  import javafx.collections.ObservableList;
 7  import javafx.event.ActionEvent;

 8  import javafx.fxml.FXML;
 9  import javafx.scene.control.Button;
10  import javafx.scene.control.Label;
11  import javafx.scene.control.ListView;
12  import javafx.scene.control.ProgressBar;
13  import javafx.scene.control.TextField;
14
15  public class FindPrimesController {
16     @FXML private TextField inputTextField;
```

```
17      @FXML private Button getPrimesButton;
18      @FXML private ListView<Integer> primesListVie
19      @FXML private Button cancelButton;
20      @FXML private ProgressBar progressBar;
21      @FXML private Label statusLabel;
22
23      // stores the list of primes received from Pr
24      private ObservableList<Integer> primes =
25          FXCollections.observableArrayList();
26      private PrimeCalculatorTask task; // finds pr
27
28      // binds primesListView's items to the Observ
29      public void initialize() {
30          primesListView.setItems(primes);
31      }
32
33      // start calculating primes in the background
34      @FXML
35      void getPrimesButtonPressed(ActionEvent event
36          primes.clear();
37
38      // get Fibonacci number to calculate
39      try {
40          int input = Integer.parseInt(inputTextFiel
41          task = new PrimeCalculatorTask(input); //
42
43          // display task's messages in statusLabel
44          statusLabel.textProperty().bind(task.messa
45
46          // update progressBar based on task's prog
47          progressBar.progressProperty().bind(task.p
48
49          // store intermediate results in the Obser
50          task.valueProperty().addListener(
51          (observable, oldValue, newValue) -> {
52              if (newValue != 0) { // task returns
53                  primes.add(newValue);
54                  primesListView.scrollTo(
55                  primesListView.getItems().size
56              }
```

```
            });

            // when task begins,
            // disable getPrimesButton and enable c
            task.setOnRunning((succeededEvent) -> {
               getPrimesButton.setDisable(true);
               cancelButton.setDisable(false);
            });

            // when task completes successfully,
            // enable getPrimesButton and disable c
            task.setOnSucceeded((succeededEvent) ->
               getPrimesButton.setDisable(false);
               cancelButton.setDisable(true);
            });

            // create ExecutorService to manage thr
            ExecutorService executorService =
               Executors.newFixedThreadPool(1);
            executorService.execute(task); // start
            executorService.shutdown();
         }
         catch (NumberFormatException e) {
            inputTextField.setText("Enter an intege
            inputTextField.selectAll();
            inputTextField.requestFocus();
         }
      }

      // cancel task when user presses Cancel Butto
      @FXML
      void cancelButtonPressed(ActionEvent event) {
         if (task != null) {
            task.cancel(); // terminate the task
            getPrimesButton.setDisable(false);
            cancelButton.setDisable(true);
         }
      }
   }
```

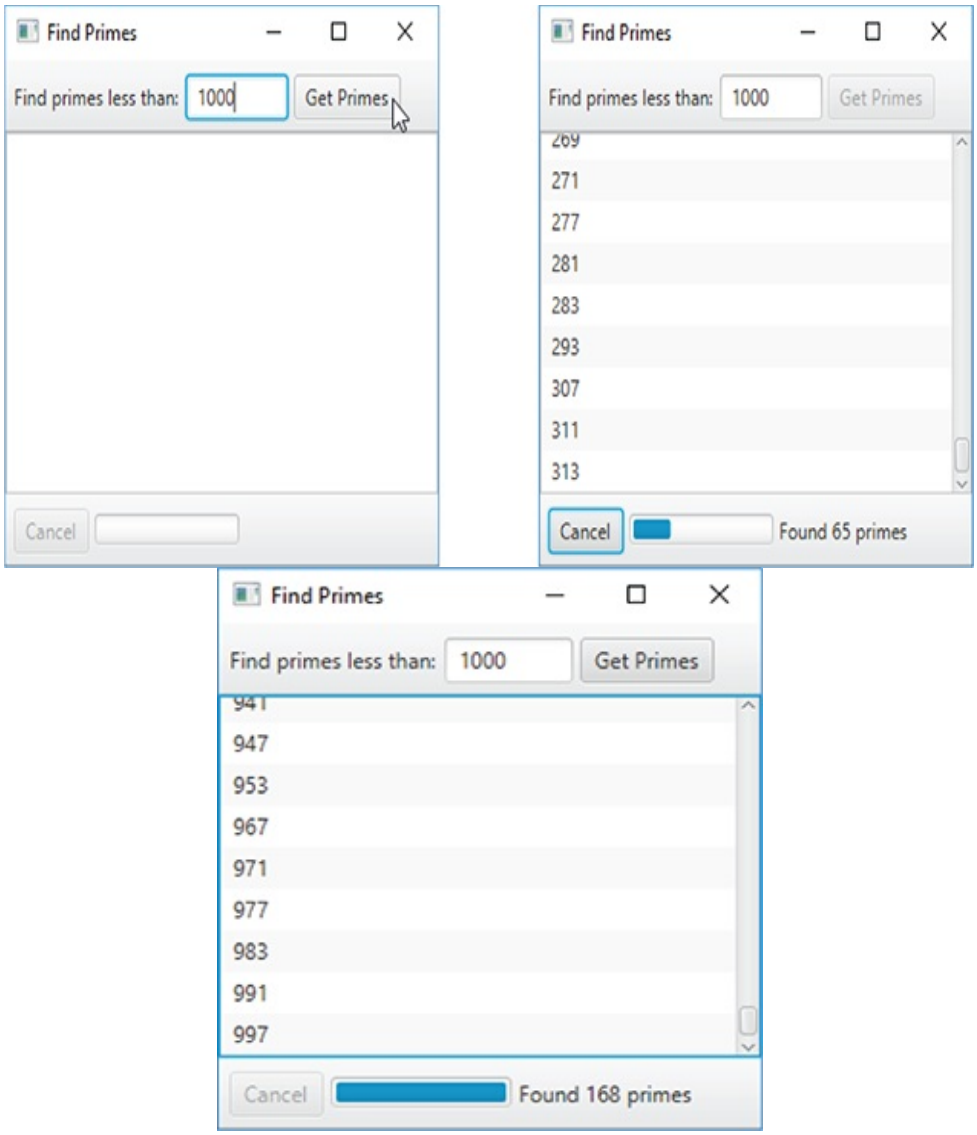


Fig. 23.28

Displaying prime numbers as they're calculated and updating a `ProgressBar`.

Description

Method `getPrimesButtonPressed`

When the user presses the **Get Primes** `Button`, method `getPrimesButtonPressed` creates a `PrimeCalculatorTask` (line 41), then configures various property bindings and event listeners:

- Line 44 binds the `statusLabel`'s `text` property to the task's `message` property to update the `statusLabel` automatically as new primes are found.
- Line 47 binds the `progressBar`'s `progress` property to the task's `progress` property to update the `progressBar` automatically with the percentage completion.
- Lines 50–57 register a `ChangeListener` (using a lambda) that gets invoked each time the task's `value` property changes. If the `newValue` of the property is not `0` (indicating that the task terminated), line 53 adds the `newValue` to the `ObservableList<Integer>` named `primes`—recall that this is bound to the `primesListView`, which displays the list's elements. Next, lines 54–55 scroll the `ListView` to its last element so the user can see the new values being displayed.
- Lines 61–71 register listeners for the task's *running* and *succeeded* state changes. We use these to enable and disable the app's `Button`s based on the task's state.
- Lines 74–77 launch the task in a separate thread.

Method `cancelButtonPressed`

When the user presses the **Cancel** `Button`, method `cancelButtonPressed` calls the `Prime-CalculatorTask`'s inherited `cancel` method (line 90) to terminate the task, then enables the `getPrimesButton` and disables the `cancelButton`.