Handling UI Tasks

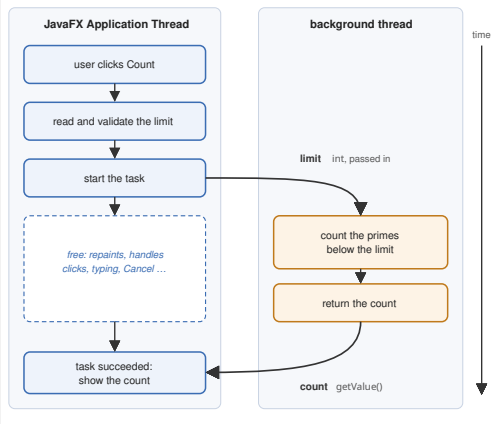
- long-running tasks in an event handler freeze the application
  - the (single) JavaFX Application thread runs every event handler and does all of the painting and layout
  - when it is busy, everything else waits in a queue
- demo – FrozenPrimeCounterApp

```
countButton_.setOnAction(e -> {
  int limit = Integer.parseInt(limitField_.getText().trim());
  int count = 0;
  for ( int n = 2 ; n < limit ; n++ ) {
    if ( PrimeCounter.isPrime(n) ) {
      count++;
    }
  }
  resultLabel_.setText("Primes below " + limit + ": " + count);
});
```


Handling UI Tasks

- rule 1: keep slow work out of the UI thread
  - run long tasks in a background thread
- rule 2: touch the UI only from the UI thread
  - event handlers already run on the UI thread
  - other threads hand their UI updates to it
  - (this isn't unique to JavaFX – most GUI toolkits work this way)
- why rule 2? – the JavaFX scene graph is not thread safe
  - it has no locking
    - the UI thread may be reading or changing it at any moment (layout, painting, handling events) so another thread's change is a race condition
  - once displayed, only the JavaFX Application Thread can safely access the scene graph

Fixing the Prime Counter



Background Tasks

- Task<V> is JavaFX's class for one-off background work whose result updates the UI
  - V is the result type
  - Task implements Runnable, so it can be handed to a thread to run – call() runs on the background thread

```
public class PrimeCountTask extends Task<Integer> {

  private int limit_;

  public PrimeCountTask ( int limit ) {
    limit_ = limit;
  }

  @Override
  protected Integer call () {         // runs on the background thread
    int count = 0;
    for ( int n = 2 ; n < limit_ ; n++ ) {
      if ( PrimeCounter.isPrime(n) ) {
        count++;
      }
    }
    return count;                     // the result
  }
}
```

Background Tasks

- the count button's handler creates the task and starts a thread to run it
- setOnSucceeded registers a handler to process the results when call() has returned
  - getValue() accesses the value returned by call()
- setOnFailed registers a handler to process the results if call() throws an exception
  - getException() accesses the exception thrown

```
countButton_.setOnAction(e -> startCount());

private void startCount () {
  int limit = readLimit();            // read the input; -1 if invalid
  if ( limit < 0 ) {
    return;
  }

  PrimeCountTask task = new PrimeCountTask(limit);
  task.setOnSucceeded(e -> showCount(limit,task.getValue()));
  task.setOnFailed(e -> resultLabel_.setText("Counting failed: " +
                                        task.getException().getMessage()));

  Thread thread = new Thread(task,"prime-count");
  thread.setDaemon(true);
  thread.start();
}
```

startCount() returns (almost) immediately

setOnSucceeded and setOnFailed define event handlers – which will be run on the JavaFX thread

making the thread a daemon thread means that closing the window ends the program even if a count is still running

Background Tasks

- call() must not access anything in the scene graph, even just to read
  - pass in what the work needs
  - results go out through the return value

```
@Override
protected Integer call () {
  int limit = Integer.parseInt(limitField_.getText());  // NO: reads the UI,
                                                        // which the user may
                                                        // be editing
  int count = 0;
  for ( int n = 2 ; n < limit ; n++ ) {
    if ( PrimeCounter.isPrime(n) ) {
      count++;
      resultLabel_.setText("" + count);     // NO: changes the UI from the
                                            // background thread
    }
  }
  return count;
}
```


Reporting Progress

- there should be feedback so the user knows something is happening

duration of task	type of feedback
very fast (< 1s)	nothing – the result just appears
short (1-10s)	show that something is happening
medium and long (over 10s)	show progress and give a way out (cancel)

amount of work	type of progress indicator
known – you can can how much is done (and how much is left)	determinate progress bar and/or a message in words ("4,000,000 of 20,000,000 checked")
unknown	indeterminate progress bar or spinner and a message saying what is happening ("waiting for the server...")

Reporting Progress for the Prime Counter

- in the task: report progress as it goes

```
@Override
protected Integer call () {
  int count = 0;
  for ( int n = 2 ; n < limit_ ; n++ ) {
    if ( n % 10000 == 0 ) {                 // every 10,000 numbers
      updateProgress(n,limit_);             // n out of limit_ done
      updateMessage("Counting... " + n + " of " + limit_ + " checked");
    }
    if ( PrimeCounter.isPrime(n) ) {
      count++;
    }
  }
  return count;
}
```

Reporting Progress for the Prime Counter

- in the application: show the progress
  - only allow one count at a time

```
// in startCount(), before starting the thread
countButton_.setDisable(true);                          // one count at a time
countProgress_.progressProperty().bind(task.progressProperty()); // a ProgressBar
statusLabel_.textProperty().bind(task.messageProperty());        // a Label
task.setOnSucceeded(e -> {
  endCount();
  showCount(limit,task.getValue());
});
task.setOnFailed(e -> {
  endCount();
  statusLabel_.setText("Counting failed: " + task.getException().getMessage());
});

private void endCount () {                      // called by every ending handler
  countProgress_.progressProperty().unbind();
  countProgress_.setProgress(0);
  statusLabel_.textProperty().unbind();
  countButton_.setDisable(false);
}
```

- updateProgress and updateMessage are allowed in call()
  - the task passes the values to the JavaFX for you
- be sure to re-enable the count button in *every* ending handler

Binding the UI to the Task

```
// the bar follows the task's progress
countProgress_.progressProperty().bind(task.progressProperty());

// the label follows the task's message
statusLabel_.textProperty().bind(task.messageProperty());

...
countProgress_.progressProperty().unbind();                    // when the task ends
statusLabel_.textProperty().unbind();
```

- binding links two properties
  - a.bind(b) means that from now on, a always has b's value – whenever b changes, a is updated automatically
  - read it as "the bar's progress follows the task's progress"
- progressProperty() is the property itself
  - getProgress() and setProgress(…) are its value
- the task changes its progress and message on the JavaFX thread (via updateProgress and updateMessage) so the bound bar and label change on the JavaFX thread too
- while bound, a value can't be changed by hand
  - must unbind() when the task ends – the value keeps whatever value it last had but can then be set again

Canceling a Task

```
@Override
protected Integer call () {
  int count = 0;
  for ( int n = 2 ; n < limit_ ; n++ ) {
    if ( n % 10000 == 0 ) {
      if ( isCancelled() ) {
        return null;                        // stop early; the value is ignored
      }
      updateProgress(n,limit_);
      updateMessage("Counting... " + n + " of " + limit_ + " checked");
    }
    if ( PrimeCounter.isPrime(n) ) {
      count++;
    }
  }
  return count;
}


cancelButton_.setOnAction(e -> task_.cancel());        // on the FX thread
task.setOnCancelled(e -> statusLabel_.setText("Cancelled"));   // runs on the FX thread
```

- the task must check whether it has been canceled
  - it is not stopped automatically – cancel() only marks the task as canceled
- after cancel(), nothing call() returns reaches the UI

Stale Results

- if the user doing things while the task is running, the results of the task may not longer be valid when the task finishes
- three strategies
  - disable the inputs while the task runs
    - stale results are impossible
    - simple but blocks the user – they have to wait for the result (or cancel)
  - label the result
    - stale results are shown but marked for what they are for
    - the user can still edit freely
    - can be confusing – results aren't current, though the user can tell that
  - cancel and restart
    - each edit cancels the operation and starts a new one for the new input
    - stale results are never shown
    - most responsive to the user but the most work to implement

Cancel and Restart

- to cancel and restart when the input changes, add a listener for the field's text
  - the listener runs on the JavaFX thread, so it needs to be short but it can access scene graph elements

```
limitField_.textProperty().addListener((obs,oldText,newText) -> restartCount());

private void restartCount () {
  if ( task_ != null && task_.isRunning() ) {
    task_.cancel();                    // its result would be for the old limit
    startCount(true);                  // quiet: no message for a half-typed limit
  }
}
```

startCount normally displays a message if the limit isn't valid – but the listener calling restartCount responds to every keystroke, so the invalid input may be because the user is in the middle of typing

Cancel and Restart

- also must guard against a replaced task affecting the UI
  - the setOnCancelled handler always runs for a canceled task
  - the replaced task may finish on its own (success or failure) before the cancellation takes effect
    - cancel() only flags the replaced task as canceled – it may not end right away

```
task.setOnSucceeded(e -> {
  if ( task == task_ ) {                    // still the current task?
    endCount();
    showCount(limit,task.getValue());
  }
});
// do the same check in setOnFailed and setOnCancelled
```

- without the guard –
  - endCount() unbinds and re-enables the controls that should be disabled and bound to the replacing task
  - showCount shows stale results

Other Threads

- not all background threads are Tasks
  - e.g. a thread reading from a server
  - e.g. a timer thread

`Platform.runLater(…)`

- use `Platform.runLater(r)` to allow another thread to run code on the application thread
  - `r` is an object of type `Runnable`
    - can also use a lambda expression
  - schedules `r`'s `run()` method to be called by the JavaFX thread sometime in the (near) future
    - adds the `Runnable` to the JavaFX thread's event queue
  - returns immediately (does *not* block until `r` is run)
  - task should complete quickly to avoid tying up application thread
  - programmer still needs to be aware of / properly handle shared variables and objects

(lambda expression version)

```
while (running) {
    Platform.runLater( () -> redraw() );
    try {
        Thread.sleep(2000);  // Wait two seconds between redraws.
    }
    catch (InterruptedException e) {
    }
}
```

Example

- a chat client's reading thread

```
for ( ; ; ) {
  String line = in.readLine();        // blocks until a message arrives
  if ( line == null ) {
    break;                            // the server closed the connection
  }
  // change the model on the FX thread
  Platform.runLater(() -> chatLog_.addMessage(line));
}
```


Periodic Updates: Timers – JavaFX

- tasks are run by handlers on the JavaFX thread
  - each action must be quick
- can directly touch the UI
- `Timeline` with a `KeyFrame` (JavaFX)
  - you specify what happens at which times (key frames)
  - use for things that happen at set intervals (e.g. a clock, a countdown, a blinking cursor) or for smooth changes of a property from one value to another
- `AnimationTimer` (JavaFX)
  - runs once per frame (about 60 times per second)
  - use for continuous motion with your own logic (animation)
- which to choose?
  - use `Timeline` if you can say when things happen
  - use `AnimationTimer` if you need to decide in each frame what happens

`Timeline`

- things that happen at given times
  - you specify what should happen at which times (key frames)

```
Timeline clock = new Timeline(new KeyFrame(Duration.seconds(1),
                              e -> showElapsedTime()));
clock.setCycleCount(Animation.INDEFINITE);
clock.play();
```


`Timeline`

- smooth changes of a property from one value to another
  - you specify the start and end values (JavaFX fills in the frames in between)

```
Timeline fade = new Timeline(new KeyFrame(Duration.seconds(2),
                             new KeyValue(node.opacityProperty(),0)));
fade.play();   // fades the node out over 2 seconds
```

- there are ready-made versions for common cases – e.g. `FadeTransition`, `TranslateTransition`
  - each animates one kind of change on a node over a set `Duration`

```
// fade a message out over 2 seconds, then hide it
FadeTransition fade = new FadeTransition(Duration.seconds(2),messageLabel_);
fade.setFromValue(1.0);
fade.setToValue(0.0);
fade.setOnFinished(e -> messageLabel_.setVisible(false));
fade.play();
```

Timeline

- built-in controls: `play()`, `pause()`, `stop()`, the number of repeats, running in reverse

```
Timeline pulse = new Timeline(new KeyFrame(Duration.seconds(0.5),
                              new KeyValue(highlight_.scaleXProperty(),1.2)));
pulse.setCycleCount(Animation.INDEFINITE);  // repeat forever (or a set number:
                                            //   setCycleCount(4))
pulse.setAutoReverse(true);  // every other cycle runs backwards: grow, shrink,
                             //  grow, …
                             // setRate(-1) to run the whole animation backwards

pulse.play();     // start, or continue after pause()
pulse.pause();    // stop where it is; play() continues from there
pulse.stop();     // stop and go back to the start; play() starts over
```

- resources
  - Creating Transitions and Timeline Animation in JavaFX
    - https://docs.oracle.com/javafx/2/animations/basics.htm
  - JavaFX Animation Example
    - https://jenkov.com/tutorials/javafx/animation.html
  - Timeline API (includes an example)
    - https://openjfx.io/javadoc/21/javafx.graphics/javafx/animation/Timeline.html

AnimationTimer

- `handle(now)` is called once per frame (about 60 times per second)
  - for smooth change, base the amount of change on elapsed time
    - frame rate isn't guaranteed (and frames can be skipped in a busy system)
    - now is the current time in milliseconds, save the time of the last frame yourself

```
AnimationTimer mover = new AnimationTimer() {
  private long last_ = 0;                     // when last frame ran

  @Override
  public void handle ( long now ) {
    if ( last_ != 0 ) {
      double seconds = (now - last_) / 1e9;   // time since the last frame
      ball_.setCenterX(ball_.getCenterX() + speed_ * seconds);
    }
    last_ = now;
  }
};
mover.start();   // stop() to stop it
```

the example shows creating a timer using an anonymous class – can also create a named subclass of `AnimationTimer`

- `start()` to start or restart after a pause, `stop()` to pause

Periodic Updates: Timers – Non-JavaFX

- task is run on its own thread (not the UI thread)
  - can do slow work
- can only touch the UI through `Platform.runLater`
- `java.util.Timer`
  - use for non-UI periodic work e.g. autosave
- your own thread with a `sleep` loop
  - use for non-UI periodic work when you need full control – the delay can change from round to round, the loop can keep state between rounds, the loop can stop for any reason, …
- which to choose?
  - use `Timer` for a task that runs once after a delay ("warn in 5 minutes") or on a fixed schedule (e.g. every 10 seconds)
  - use your own thread with a sleep loop with a delay that changes (e.g. wait longer after each failure or wait until a specific time) or for a long-running activity that isn't naturally "the same small step repeated"

Timer

- define a `TimerTask` with the desired `run` method
- create a `Timer`
  - each `Timer` has its own background thread
- call `schedule(task,delay,period)` or `scheduleAtFixedRate(task,delay,period)` to run
  - `schedule` is for fixed delay – it waits a set time after each run ends
  - `scheduleAtFixedRate` is for regular start times – it shortens the delay before the next run to catch up if a previous run was late
  - multiple tasks sharing a `Timer` run one at a time – a slow task delays others
  - `Timer` is thread safe – multiple threads can share and schedule tasks on the same `Timer` object (and thus timer thread)
- call `cancel` to terminate timer thread to ensure a delay in garbage collection doesn't prevent application termination
  - timer thread isn't a daemon thread unless created with `new Timer(true)`
- an uncaught exception in `run()` ends the timer's thread – every task scheduled on it stops



Timer

the example shows creating a timer using an anonymous class – can also create a named subclass of `TimerTask`

```
Timer poller = new Timer(true);   // daemon: won't keep the program running

poller.schedule(new TimerTask() {
  @Override
  public void run () {                        // runs on the timer's thread
    int count = server_.countNewMessages();   // slow: waits for the network
    Platform.runLater(() -> unreadLabel_.setText(count + " new messages"));
  }
},0,10_000);                    // start now, then every 10s after each check

...
poller.cancel();                              // stop checking
```