P7. BACKGROUND PROCESSING

Interfaces Persona Computador

Depto. Sistemas Informáticos y Computación

UPV

Outline

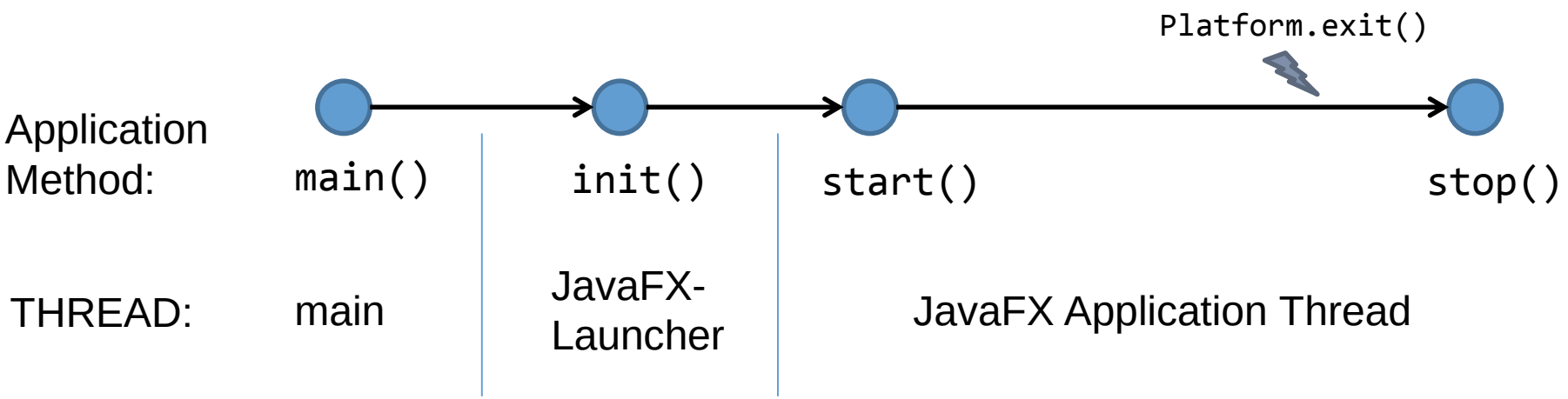
- Introduction
- Threads and Nodes
- Task Status
- The Task class
- The WorkerStateEvent class
- The Service class
- Changing the mouse cursor
- Useful Tools
- Exercise
- References

Introduction

- If you have tried to execute a computationally heavy task in a JavaFX event handler (for example, opening a big file or downloading a file from the Internet), you will probably have experienced how the interface *freezes*
  - Event handlers should not perform heavy tasks
- The proper method of executing tasks that could require some time to complete is:
  - Let the user know the task duration (for example, with a progress bar or, at least, with a wait cursor)
  - Launch the task in a separated thread
  - When the task ends, update the scene view

Threads in JavaFX

- Most of the time, JavaFX applications are executed in the JavaFX Application Thread, but there are other threads:



Threads and Nodes

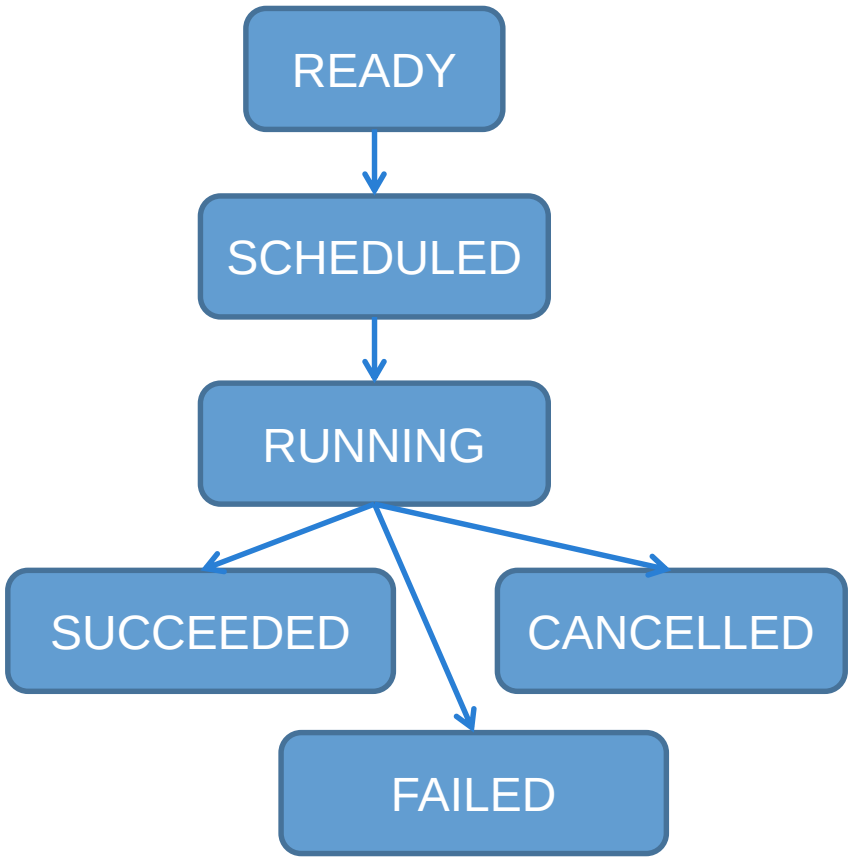
- JavaFX offers several classes for creating working threads and for synchronizing them with the GUI
  - In the package `javafx.concurrent`, that is based on and is compatible with the standard `java.util.concurrent`
    - `Worker` (interface): specifies the API that any task to be executed in a working thread and that will communicate with the JavaFX thread should exhibit
    - `Task` (class): contains the logic that will be executed in the working thread, and methods for communicating with the JavaFX thread
    - `Service` (class): it is in charge of executing tasks
    - `WorkerStateEvent` (class): represents an event that occurs when a task changes its status

Threads and Nodes

- Nodes can be built and modified in any thread, as long they are not in use in the scene of a visible window
  - The scene graph of a visible window can be modified exclusively by code running in the JavaFX Application Thread

Task Status

- The following diagram shows the life cycle of a task:



Main properties of Task (`ReadOnly<T>Property`):

- `Double totalWork, workDone`
- `Double progress (-1, 0..1)`
- `Boolean running`
- `Object<Worker.State> state`
- `Object<V> value`
- `Throwable exception`
- `String message, title`

La `Task` class

- We will use this class to implement the code to be executed in a working thread:
  - Create a new class that inherits from Task
  - Override the `call` method with the code to execute. It will return a result. Inside of this method:
    - The scene graph CANNOT be manipulated
    - It is possible to invoke to the methods: `updateProgress`, `updateMessage` and `updateTitle` for letting JavaFX know the status of the execution
    - Check regularly whether the task has been cancelled (`isCancelled()`) and, if so, end the execution immediately
- The Task objects can not be reused (a new one must be executed each time)

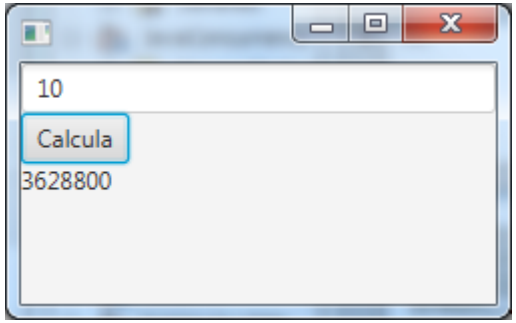
La Task class

- Example:

```
import javafx.concurrent.Task;
Task<Long> task = new Task<Long>() {
  @Override
  protected Long call() throws Exception {
    long f = 1;
    for (long i = 2; i <= computeFactorial; i++) {
      if (isCancelled()) {
        break;
      }
      f = f * i;
    }
    return f;
  }
};
```

```
@Override public void start(Stage primaryStage) {
  TextField num = new TextField();
  Label res = new Label();
  Button btn = new Button("Calcula");
  btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override public void handle(ActionEvent event) {
      final long computeFactorial = Long.parseLong(num.getText());
      // Insert here the code from the previous slide
      res.textProperty().bind(Bindings.convert(task.valueProperty()));
      Thread th = new Thread(task);
      th.setDaemon(true);
      th.start();
    }
  });
  VBox root = new VBox();
  root.getChildren().addAll(num, btn, res);
  Scene scene = new Scene(root, 300, 250);
  primaryStage.setScene(scene);
  primaryStage.show();
}
```

Launching the task



Blocking calls inside the task

- If the Task invokes to `Thread.sleep` or any other blocking method and the user cancel the task, an `InterruptedException` will be thrown. We should check the cancellation again:

```
Task<Long> task = new Task<Long>() {
  @Override protected Long call() throws Exception {
    long f = 1;
    for (long i = 2; i <= computeFactorial; i++) {
      if (isCancelled()) {
        break;
      }
      f = f * i;
      try { Thread.sleep(100); }
      catch (InterruptedException e) { if (isCancelled()) break; }
    }
    return f;
  }
};
```

The `Task` class

- The `runningProperty` property can be used to hide or show elements in the interface while the task is in execution:

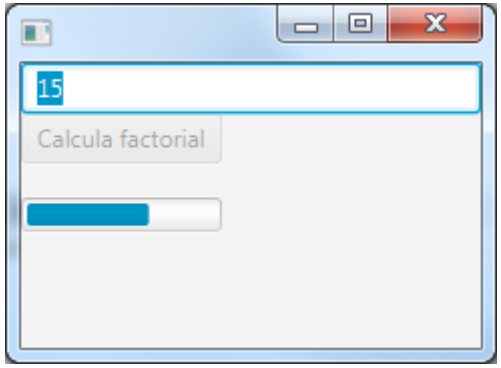
```
Label res;
Button btn;
// The label will show the result
res.textProperty().bind(Bindings.convert(task.valueProperty()));
// But it will not be visible while the task is running
res.visibleProperty().bind(Bindings.not(task.runningProperty()));
// Moreover, the button will be disabled while there is a running task
btn.disableProperty().bind(task.runningProperty());
```

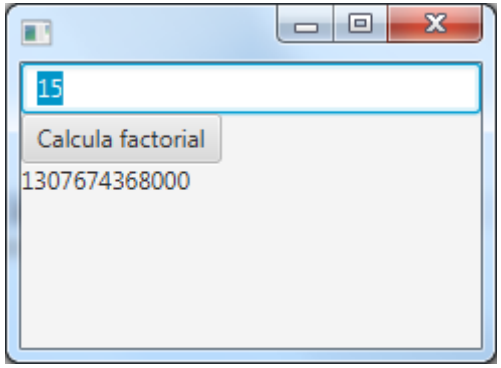
Showing Progress

- In any task that takes time (more than 1 or 2 s), it is advisable to keep the user updated about what the system is doing:

```
ProgressBar bar = new ProgressBar(0.0);

Task<Long> task = new Task<Long>() {
@Override protected Long call() throws Exception {
  long f = 1;
  for (long i = 2; i <= computeFactorial; i++) {
    […]
    updateProgress(i, calculaFactorial);
  }
  return f;
}
bar.progressProperty().bind(task.progressProperty());
bar.visibleProperty().bind(task.runningProperty());
[…]
root.getChildren().addAll(num, btn, res, bar);
```





The `WorkerStateEvent` class

- In each status change, the class that implements `Worker` generates a different event. How to use them:

- Outside of Task:

```
Label status = new Label();
task.setOnRunning(new
 EventHandler<WorkerStateEvent>() {
  @Override
  public void handle(WorkerStateEvent event) {
    status.setText("Computing...");
  }
});
task.setOnSucceeded(new
 EventHandler<WorkerStateEvent>() {
  @Override
  public void handle(WorkerStateEvent event) {
   status.setText("Done!");
  }
});
```

- Usando los métodos de ayuda de Task

```
Task<Long> task = new Task<Long>() {
@Override protected Long call()
[…]
@Override protected void running() {
  super.running();
  updateMessage("Computing...");
}
@Override protected void succeeded() {
  super.succeeded();
  updateMessage("Done!");
}
}
status.textProperty()
    .bind(task.messageProperty());
```

Executing Code in the JavaFX Thread

- Sometimes it is necessary to modify the status of a JavaFX application from a working thread. It is not possible to do it directly, but we can use the method:
  - `javafx.application.Platform.runLater(Runnable runnable)`
- The method will execute the `runnable` in the JavaFX Application Thread at some point in the future
- For example:

```
Platform.runLater(new Runnable() {
  @Override public void run() {
    customer.setFirstName(rs.getString("FirstName"));
    // etc
  }
});
```

The `Service` class

- The `Service` class implements also the `Worker` interface
- One difference with `Task` is that a `Service` object can be reused (executed, stopped, re-executed, etc.)
  - Although internally, `Service` is creating a new `Task` each time
- The `Service` class has a higher level than `Task`, and is in charge of creating the Threads using Executors
- The `ScheduledService` class is in charge of relaunching a task after it finishes, for implementing repetitive tasks
- More information in the JavaFX documentation

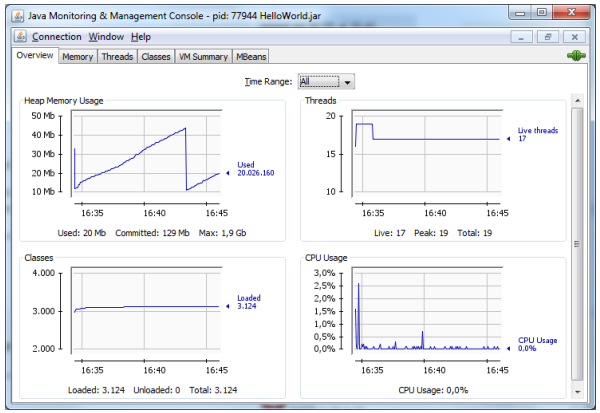
Changing the Cursor

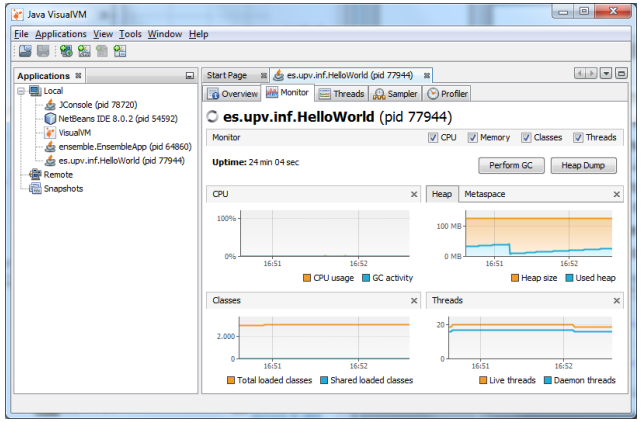
- Other common action when launching a length task is to change the cursor to a wait cursor:

```
final Scene _scene = scene;
@Override
protected Long call() throws Exception {
  Platform.runLater(new Runnable() {
    @Override public void run() {
      _scene.setCursor(Cursor.WAIT);
  }});
  long f = 1;
  for (long i = 2; i <= computeFactorial; i++) {
    if (isCancelled()) {
      break;
    }
    f = f * i;
  }
 Platform.runLater(new Runnable() {
    @Override public void run() {
      _scene.setCursor(Cursor.DEFAULT);
  }});
  return f;
}
```

Useful Tools

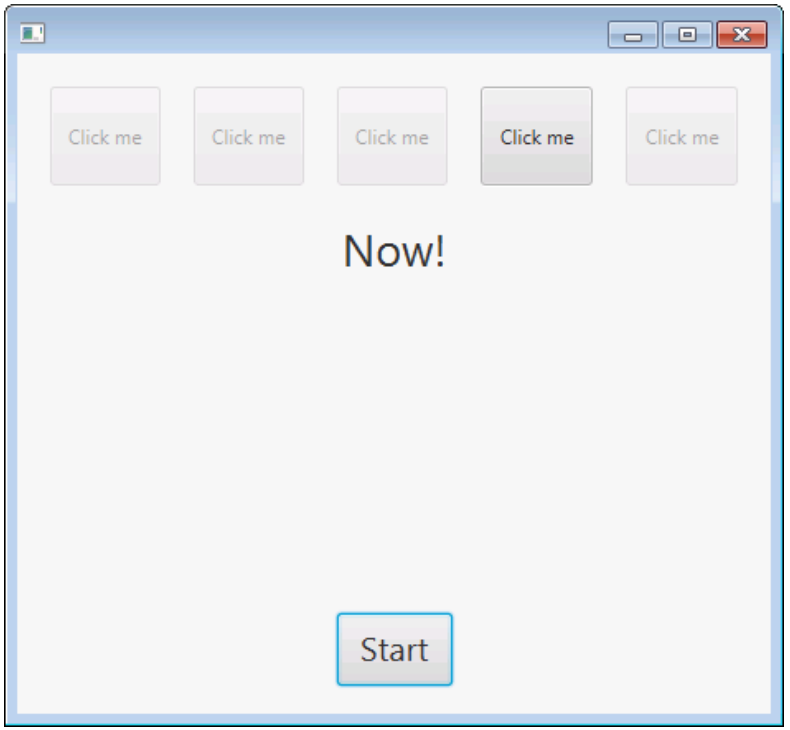
- The following tools in the JDK can be helpful for studying the status of a Java application
  - `jconsole`: shows in real time information about running Java applications
  - `jps`: shows in the console the list of running Java applications, with their id
  - `jstack`: shows the execution stack of a Java application
  - `jvisualvm`: like `jconsole`, but with more options





Exercise

- A game for measuring the user's response time has been implemented
- Initially, the buttons at the top are disabled. After clicking on the *Start* button, the game waits a random period of time between 1 and 6 seconds, and enables a button randomly
- The game will measure the time it takes to the user to click the button.
- The programmer has written all the code in the *Java Application Thread*, and therefore it does not work. Fix it.



References

- https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Task.html
- https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm