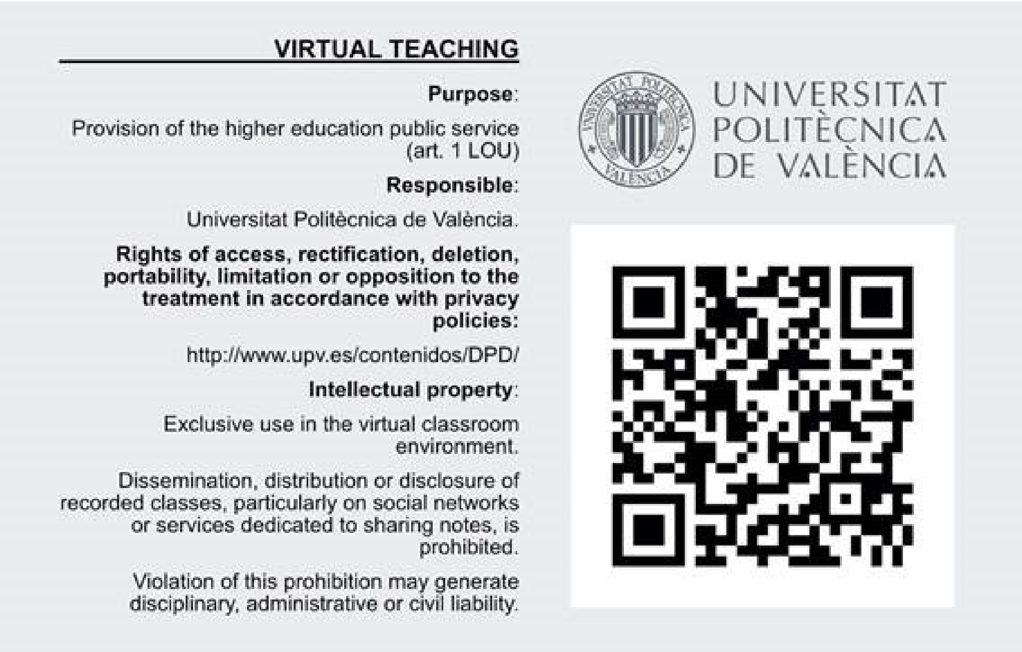
P7. BACKGROUND PROCESSING



VIRTUAL TEACHING

Purpose:

Provision of the higher education public service (art. 1 LOU)

Responsible:

Universitat Politècnica de València.

Rights of access, rectification, deletion, portability, limitation or opposition to the treatment in accordance with privacy policies:

http://www.upv.es/contenidos/DPD/

Intellectual property:

Exclusive use in the virtual classroom environment.

Dissemination, distribution or disclosure of recorded classes, particularly on social networks or services dedicated to sharing notes, is prohibited.

Violation of this prohibition may generate disciplinary, administrative or civil liability.

Interfaces Persona Computador

Depto. Sistemas Informáticos y Computación

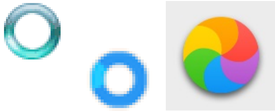
UPV

Outline

- Introduction
- Concurrence in JavaFX
- Interface Worker<V>
- ClassTask<V>
- Class Service<V>
- Class WorkerStateEvent
- Changing the mouse cursor
- Useful Tools
- References

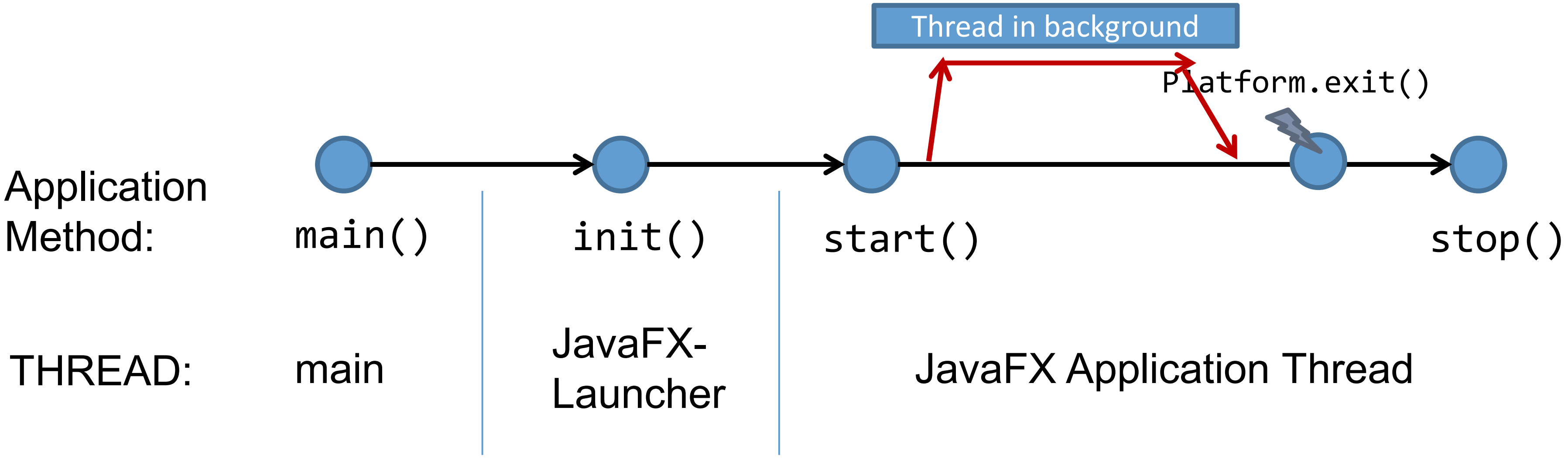
Introduction

- 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:



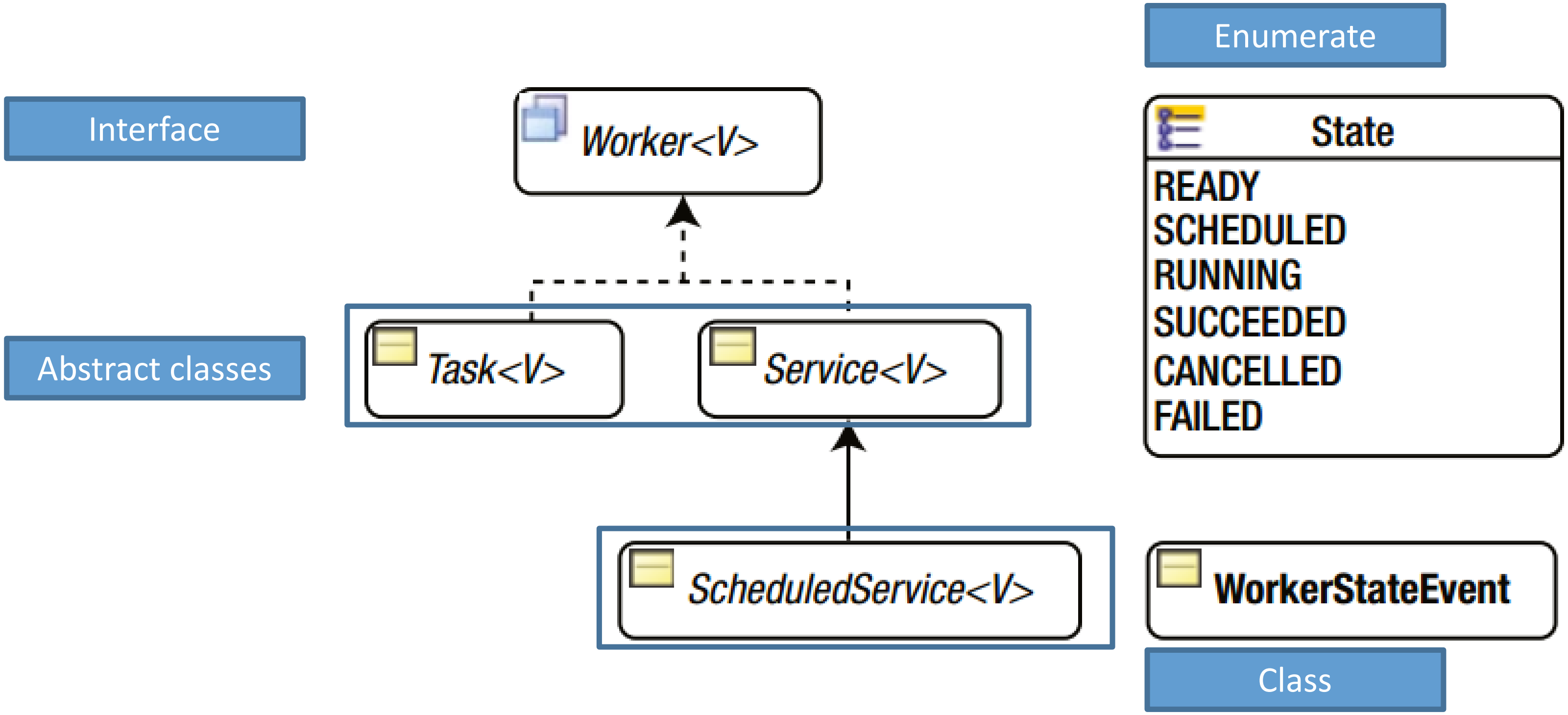
Introduction

- We try to solve the following problems:
  - Perform processing in the background while you can do something in the first thread. You use the `Worker` interface and the classes:
  - `Task`, `Service`, `ScheduledService`.

  - Update the interface with results from background threads, for example using:
    - `Platform.runLater`
    - `Task properties`

Concurrence in JavaFX

- The concurrence framework in JavaFX is based on the `java.util.concurrent` framework. It is focused to work with GUIs.



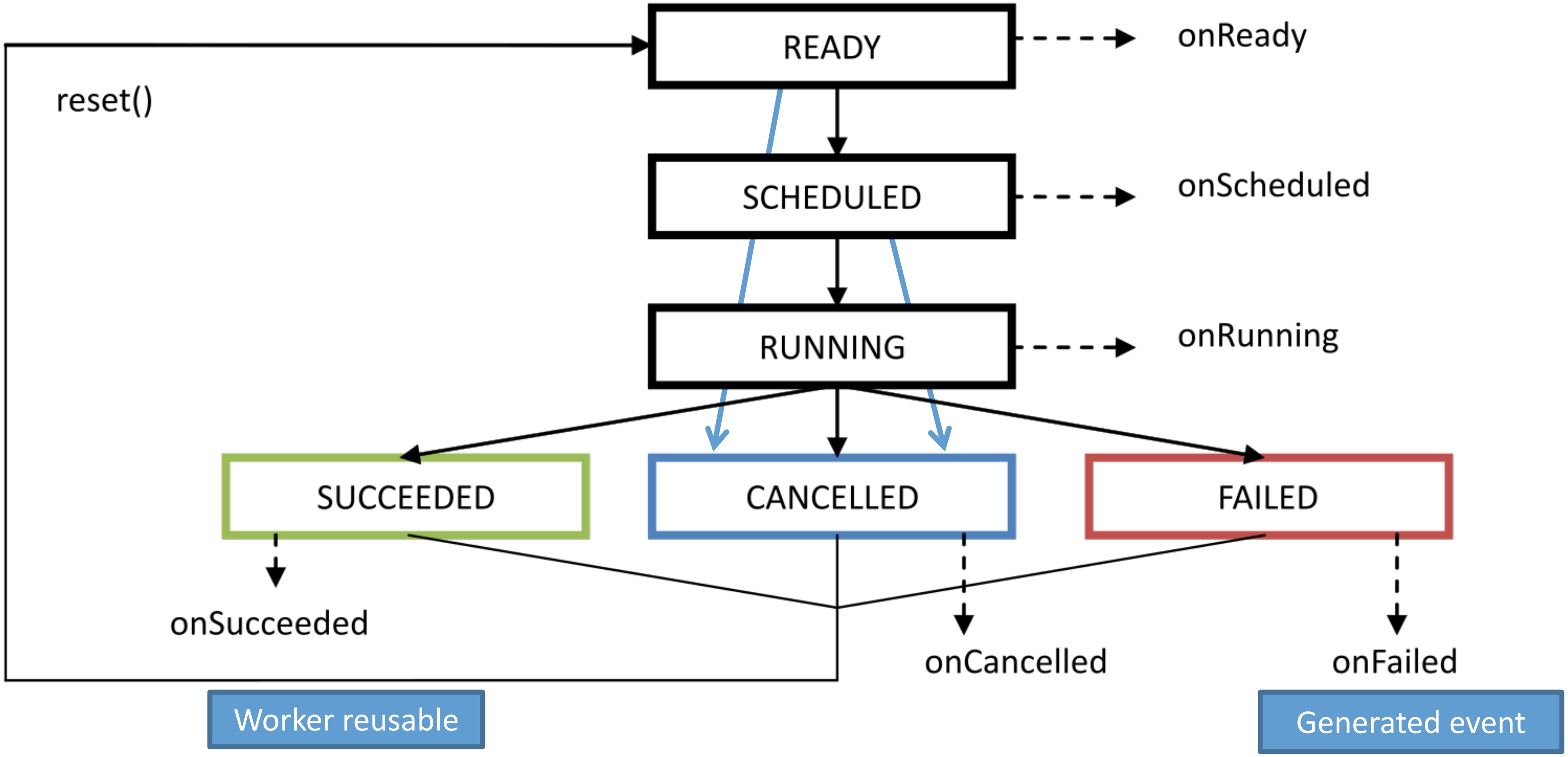
We will talk generically about tasks instead of specifying Worker, Task, Service, etc.

Concurrence in JavaFX

- A `Worker` instance represent a taks which which must be executed in one or more background threads.
- `Task, Service and ScheduledService` are abstract classes which implement the `Worker` interface.
- A `Task` instance represents a non-reusable task (one execution).
- A `Service` instance represents a reusable task (more than one execution).
- A `ScheduledService` is a task which can be planned and executed executed more than one time after a time interval (when it finalizes, it initiates again)
- Values of the enurmated type `State` represent the possible states of a `Worker` instance.
- A `WorkerStateEvent` instance represent an event which happens when the state of a Worker instance changes

Concurrence in JavaFX

- Task lifecycle



- The non-reusable ones don't have Ready status.

Interface Worker<V>

- It is a type of task than can be executed in one or more background threads.

- The V parameter is the type of the result returned by the Worker (Void, if it doesn’t return a value, with the first character in capital letter).

- The task state implements observable and it is publish to the main JavaFX thread, so it is possible to use it to modify the nodes of the scene (bindings, etc.).

Interface Worker<V>

- Some properties of the internal state:

```
title     : task name
message   : Mensaje detallado durante la ejecución, para feedback
running   : True if it is executing or it is Scheduled
state     : enumerated value (READY, SCHEDULED, RUNNING, CANCELLED, SUCCEEDED, FAILED)
Progress  : Ratio between workDone and totalWork
workDone  : quantity of work done
totalWork : quantity of work to do
value     : Value returned by the task when it ends in success.
            Null in another case (also if V is Void).
exception : Exception generated if the task fails
```

Value=-1 if they’re unknown

Use of Task<V>: definition

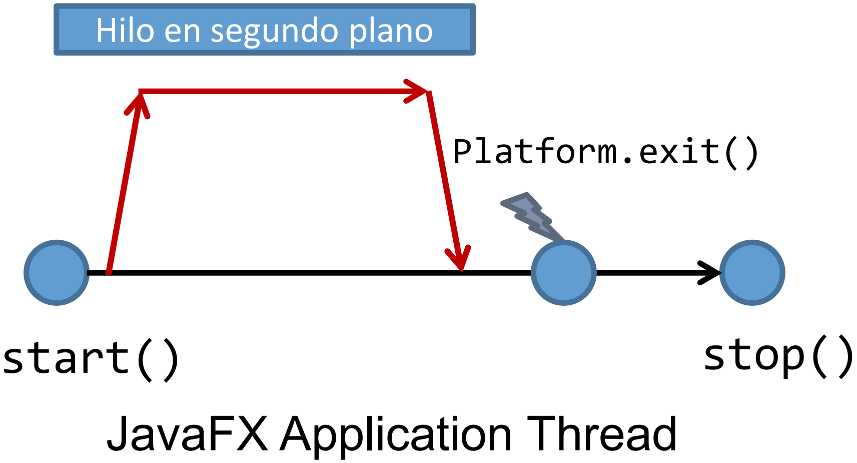
- It represents a task that is executed only once, if it ends, is cancelled, or fails, it cannot be restarted.
- To create a Task, we need to create a new class which extends from `Task<V>` and override the abstract method `call()`
- For example:

```
import javafx.collections.ObservableList;
import javafx.concurrent.Task;

//Returns a list with prime numbers
public class TaskFindPrimes extends Task<ObservableList<Long>>
{
    @Override
    protected ObservableList<Long> call() throws Exception {
         //my code to find prime numbers
    }

}
```

Use of Task<V>: update

- While the task is executing, we can update some of its properties using the following methods
- This properties can be observed (bindings, listeners..) from the main JavaFX Thread: messageProperty(), valueProperty(), progressProperty()…

```
protected void updateMessage(String message)
protected void updateProgress(double workDone, double totalWork)
protected void updateProgress(long workDone, long totalWork)
protected void updateTitle(String title)
protected void updateValue(V value)
```



Use of Task<V>: update

- We call the previous methods in the `call()` methods of the task.

- In the `call()` method, we can access to objects of the main GUI thread using:

  `Platform.runLater()  (explain after)`

- An easy way of modify GUI objects depending of the task work is binding the task properties to them. We have a property for each value: messageProperty(), valueProperty(), progressProperty(), workDoneProperty(), totalWorkProperty() , titleProperty(), runningProperty()..

Use of Task<V>: events

- Tasks generate different events when their state changes that we can handle:

```
onCancelled
onFailed
onRunning
onScheduled
onSucceeded
```

- For example, we can add a handler when a task finishes successfully:

```
Task<ObservableList<Long>> task = new MyTask()
task.setOnSucceeded(e -> { System.out.println(“Task successfully finished.")
});
```

Use of Task<V>: cancel

- We have two different methods to cancel a task:

```
public final boolean cancel()
public boolean cancel(boolean mayInterruptIfRunning)
```

- `cancel(): it` removes the task from the execution queue or ends its execution.
- `cancel(mayInterruptIfRunning):` it allows to control if the thread of the task can be interrumpt or not.
- Inside the task method `call():`
  - check is the task has been cancelled (`isCancelled()`).
  - If it is cancelled, call() must finish, on the contrary `cancel(true)` will not work properly

Use of Task<V>: example

- This task calculates the factorial of the number `calculateFactorial.`

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

Anonymous class

Use of Task<V>: start

- A task can be started in two ways:

1

```
// Program the task in a background thread
Thread backgroundThread = new Thread(task);
backgroundThread.setDaemon(true);
backgroundThread.start();
```

2

```
//Use an Executor Service to program the task:
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(task);
```

Use of Task<V>: waiting

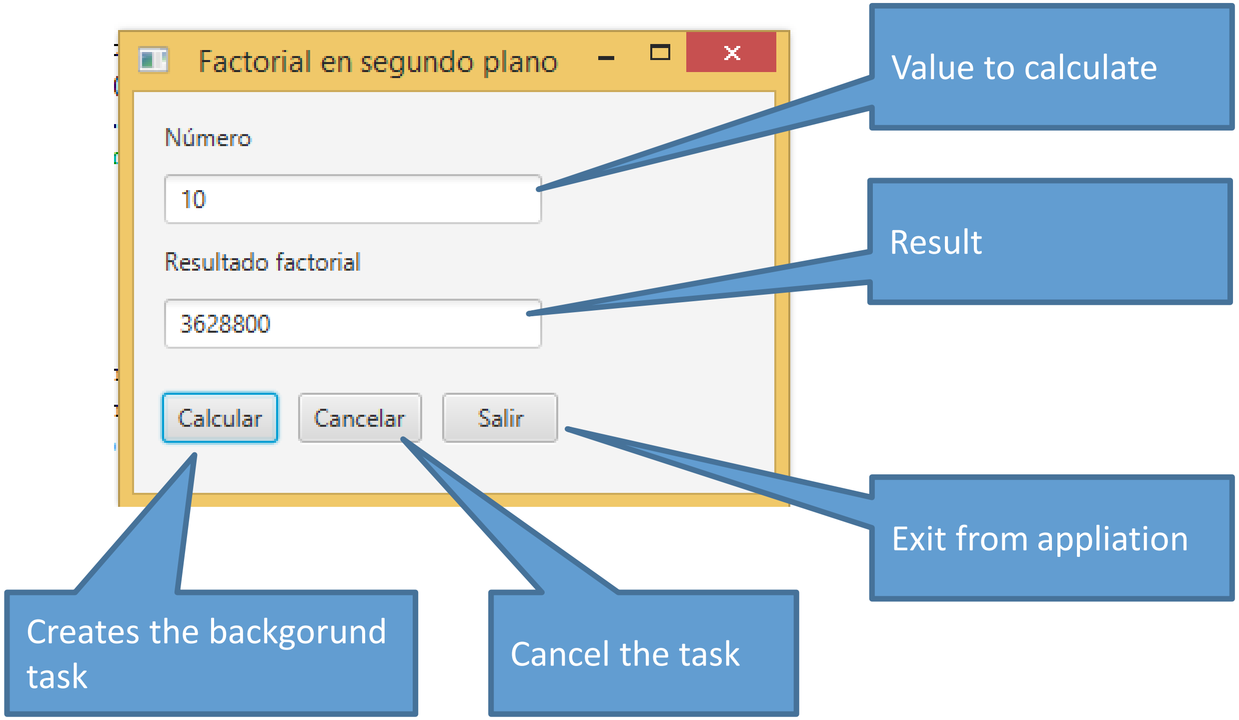
- Calling `Thread.sleep` the task is stopped during the milliseconds given as parameter
- If the task is cancel during a sleep, an `InterruptedException` is thrown. It is required to check the cancellation again.

```
class Factorial extends Task<Long> {
private Long calculateFactorial; //Factorial of this value
     @Override
     protected Long call() throws Exception {
            long f = 1;
            for (long i = 2; i <= calculaFactorial; i++) {
              if (isCancelled()) {
                break;
              }
              f = f * i;
              try { Thread.sleep(100); }
              catch (InterruptedException e) { if (isCancelled()) break; }
            }
            return f;
            }
    }
```

Internal class or in other file

Use of Task<V>: factorial example

- Example of calculation of the factorial as a background thread, defined from the application controller.



Factorial en segundo plano

Número

10

Resultado factorial

3628800

Value to calculate

Result

Exit from appliation

Creates the backgorund task

Cancel the task

Use of Task<V>: factorial example

- First, we define the Factorial class, which extends from Task<Long> (returns a long value)

```
// Internal class in the controller
    class Factorial extends Task<Long> {

        private Long calculaFactorial; //valor para el que se calcula el factorial
        public Factorial() { }
        public Factorial(Long valor) { calculaFactorial = valor; }

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

Internal class to the controller or in other file

Use of Task<V>: factorial example

- Adding the start and cancel button handlers

```
 public class FXMLDocumentController implements Initializable {

     private Factorial miTarea;
     @FXML
     private TextField numero; // input field
     @FXML
     private TextField factorialResultado; // output field
 ..
 }
```

Handler to calculate the factorial

```
@FXML void handleButtonCalcular(ActionEvent event) {
   Long factorial;
factorial = Long.parseLong(this.numero.getText());
   miTarea = new Factorial(factorial);
```

The value of the task is binded to the output field

```
   factorialResultado.textProperty().bind(Bindings.convert(miTarea.valueProperty()));
```

Creates and starts the background thread

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

Use of Task<V>: factorial example

- Cancel button handler

```
@FXML void handleCancelar(ActionEvent event) {
        miTarea.cancel();
 }
```

- we link the result text field of the scene with the task's valueProperty

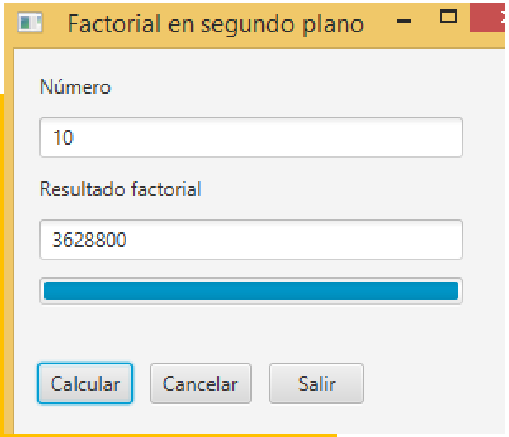
```
factorialResultado.textProperty().bind(Bindings.convert(miTarea.valueProperty()));
```

- How to know the task progress?
  - We add a call to the updateProgress method in the call() method to set the progress done
  - The progress bar value of the scene is bound to the progressProperty() of the task

Use of Task<V>: factorial example

- Update progress in Factorial class:

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



Factorial en segundo plano

Número

10

Resultado factorial

3628800

- Binding the progressProperty in the scene controller:

```
@FXML     void handleButtonCalcular(ActionEvent event) {
         Long factorial;
         factorial = Long.parseLong(this.numero.getText()); // no se comprueban errores de formato
         miTarea = new Factorial(factorial);
         factorialResultado.textProperty().bind(Bindings.convert(miTarea.valueProperty()));
         barraProgreso.progressProperty().bind(miTarea.progressProperty());
         Thread th = new Thread(miTarea);
         th.setDaemon(true);
         th.start();
    }
```

Use of Task<V>: factorial example

- `runningProperty:` it can be used to make visible or invisible some nodes of the scene. For example:
  - Hiding the result text field while the task is running
  - Disabling the “Calcular” button while the task is running

```
@FXML     void handleButtonCalcular(ActionEvent event) {
…
// Result field invisible
   factorialResultado.visibleProperty().bind(Bindings.not(miTarea.runningProperty()));
// Button disabled while the task is running
   calcular.disableProperty().bind(miTarea.runningProperty());
   Thread th = new Thread(miTarea);
   th.setDaemon(true);
   th.start();
```

Use of Task<V>: RunLater

- `Platform.runLater: it` is executed in the main `JavaFX` thread. It is useful to update the user interface from a background thread when the execution is not very long in time .

```
@Override protected Long call() throws Exception {
  long f = 1;
  for (long i = 2; i <= calculaFactorial; i++) {
    if (isCancelled()) { break; }
    f = f * i;
    updateProgress(i, calculaFactorial);
    updateValue(f);
    Platform.runLater(() -> {
         factorialResultado.setText(miTarea.valueProperty().get().toString());
            });
    try {
      Thread.sleep(100);
    }catch (InterruptedException e) {
              if (isCancelled())
                  break;
    }
  }
  return f;
 }
```

Factorial example:
Replacing the binding by runLater
(see slide 23)

Use of Task<V>: Summary

- Used to add the code that will run in a thread in the background:
  - Create a new class that extends `Task`
  - Override `call`, adding the needed code, returning the value if required (or null of Void). Inside this method:
    - Handle the scene graph is NOT allowed
    - We can use the task methods `updateProgress`, `updateValue`, `updateMessage` and `updateTitle` to inform the main JavaFX thread about the task execution progress
    - It is required to check if the task has been cancelled using `isCancelled()`, and finish the execution in such case
- Task Object can be executed only one time. We need to create and planning it each time we need to execute it.

Use of Service<V>

- Service<V> is an abstract class which implements the interface Worker<V> and wraps a Task<V>

- 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

- In the previous example, we can use a Service<Long> instead of the Task<Long>

Use of Service<V>

- As an internal class of the controller or in other file

```
class MiServicio extends Service<Long> {

        private Long factorial; // número para el factorial
        public MiServicio(Long numero){ factorial = numero; }
        @Override
        protected Task<Long> createTask() {
            return new Factorial(factorial);
        }

    }
```

- createTask() is called automatically each time the service is started or restarted

Use of Service<V>

- It has not got updateXXX methods, as they are related to their internal task.

- It has the same state events as tasks, an also one new: onReady

- Provides the same properties as tasks to access to the interal state of its internal task: messageProperty(), valueProperty(), progressProperty(), workDoneProperty(), totalWorkProperty() , titleProperty(), runningProperty()..

Use of Service<V>

- start(): initiates the execution of a service.

```
MyService myService= new MyService(factorial2);
myService.start();
```

- cancel() : cancels the service.
- reset(): restart the service properties to their initial values. If the service is in the state RUNNING or SCHEDULED, an exception is thrown.
- restart(): use this method to restart the service, as it calls to cancel(), reset() and start() in arranged way.

The `WorkerStateEvent` class

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

- Outside 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!");
  }
});
```

Executed in the main JavaFX thread

- Inside Task: using its helping methods

```
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());
```

Executed in the background thread

Changing the Cursor

- Other common action when launching a long task is to change the cursor to a wait cursor. We can do it inside the call method:

```
 class Factorial extends Task<Long> {
     private Long calculaFactorial; //valor para el que se calcula el factorial
     private Scene _scene;
     public Factorial() { }
     public Factorial(Long valor, Scene scene) {
            calculaFactorial = valor;
            _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;
    }
}
```

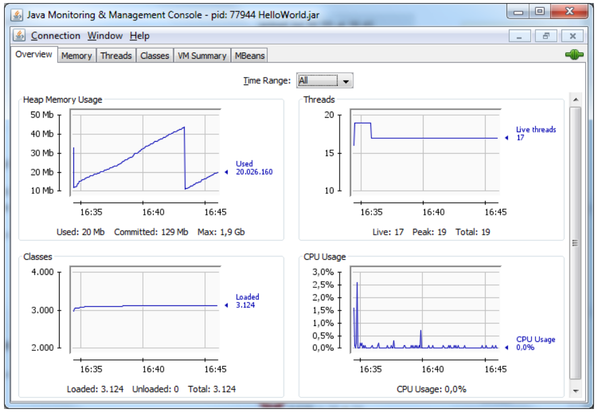
Changing the Cursor

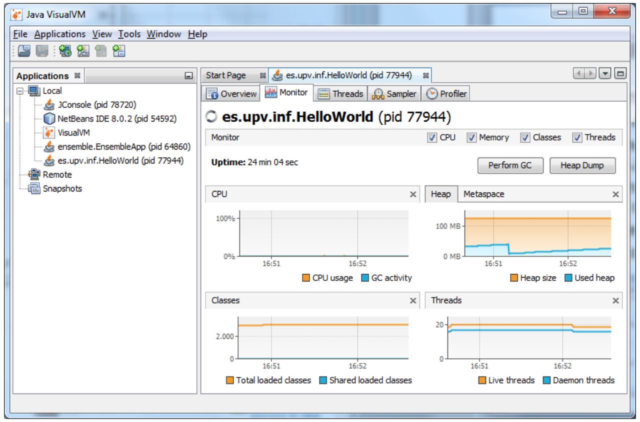
- Using convenient methods to add handlers to the different state events:

```
Task<Long> backgroundTask;
Long calculaFactorial = inputText.textProperty().getValueSafe();
backgroundTask = new Factorial(calculaFactorial);

//We change the cursor when the Task starts.
backgroundTask.setOnRunning((e)->{
            myButton.getScene().setCursor(Cursor.WAIT);
});

//We change the cursor when the Task ends succesfully.
backgroundTask.setOnSucceeded((e)->{
            myButton.getScene().setCursor(Cursor.DEFAULT);
 });
//We change the cursor when the Task ends with error.
backgroundTask.setOnFailed((e)->{
            myButton.getScene().setCursor(Cursor.DEFAULT);
});
//We change the cursor when the Task is cancelled.
backgroundTask.setOnCancelled((e)->{
            myButton.getScene().setCursor(Cursor.DEFAULT);
});
```

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





Bibliography

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