

# Advanced Programming

## 2nd Exercise: Concurrent and Asynchronous Abstractions

Programming Group  
guido.salvaneschi@unisg.ch

University of St.Gallen

### 1 Goals

In this exercise, you'll learn how you can effectively use different approaches to manage concurrency in your programs, including synchronous programming, callbacks, async-await, futures, and actors.

### 2 Concurrency in Kotlin

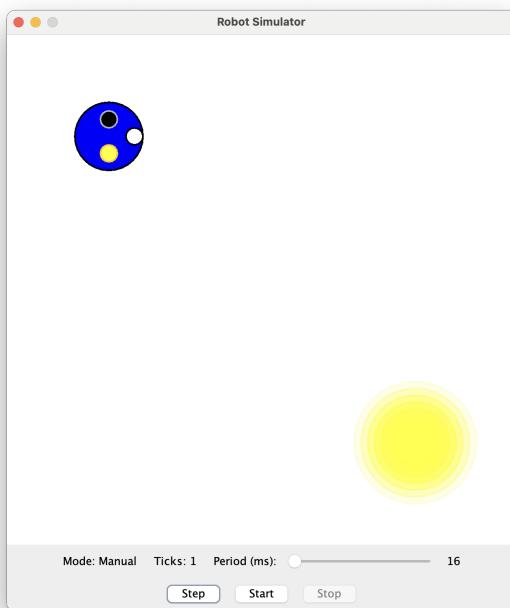
#### 2.1 Instructions

In the following exercises, you will experiment with a robot simulator and write concurrent tasks that the robots have to execute, using different concurrency paradigms. Here are some instructions that explains how to use the simulator and program the robots.

##### 2.1.1 Robot Simulator

Figure 1 shows how the simulator looks like.

Figure 1: The robot simulator.



On screen, you can see a white **canvas** containing a blue **robot** on the top left and a yellow **light source** on the bottom right. Inside the robot, there are the **front**, a **led**, and a **light sensor**. The front is represented by the white circle on the right and tells which direction the robot is facing (currently the right). The led is represented by the black circle on top and can be turned on or off with a specific color (currently off). The light sensor is represented by the yellow circle on the bottom and tells the closest light source to the robot (currently the yellow light).

On the canvas, coordinates are expressed in **pixels**, the top-left corner being  $(0, 0)$ , while the bottom-right corner has positive coordinates  $(width, height)$ . Directions are expressed in **degrees**,  $0^\circ$  being the right,  $90^\circ$  the top,  $180^\circ$  the left, and  $270^\circ$  the bottom. Clicking on the canvas will spawn a new light source or change the color of an existing one. Clicking repeatedly the same light source will instead remove it.

Below the canvas, there's the **control panel**, where you can read the amount of time passed in the simulation (**ticks**) and set the speed of the simulation (**period**). In the control panel, there are also three buttons. Clicking **Step** will advance the simulation by one tick. Clicking **Start** will create a timer that advances the simulation by one tick every period (currently every 16ms). Clicking **Stop** will stop that timer.

### 2.1.2 Programming Robots - Instructions

In the simulator, the actions performed by a Robot during a simulation are expressed by a **RobotBehavior**, which is simply a lambda that accepts a Robot as input. The actions are limited to the following sensors and actuators:

- **Light Sensor:** a sensor that detects the light closest to the robot. The robot can use it to detect the direction, distance, and color of the closest light:  
`robot.lightSensor.closestLight()`
- **Led:** a simple light. The robot can turn it on/off and change its color:  
`robot.led.switch(on=..., color=...)`
- **Motor:** an actuator that allows the robot to move forward, following its facing direction:  
`robot.motor.forward(distance=...)`
- **Spin Motor:** an actuator that allows the robot to rotate around itself, changing its facing direction:  
`robot.spinMotor.rotate(angle=...)`

Notably, all of these actions are *blocking* and require time in the simulation to pass in order to complete. Instead, the **robot body** allows to access the **position**, **direction**, **color**, and **size** of the robot *instantly* (e.g. `robot.body.position`). Additionally, the robot body allows to translate absolute coordinates into relative directions for the robot to follow (`robot.body.relativeDirection(destination=...)`), and viceversa (`robot.body.absolutePosition(direction=...)`).

#### Task 1: Tutorial

Unzip the archive `concurrency-in-kotlin.zip` and open it with IntelliJ IDEA. Let's look together at the file `src/main/kotlin/exercises/e0/Tutorial.kt`. Inside, the `Tutorial.behavior` showcases all possible interactions that a robot can perform.

You can run it by executing the `main` function and pressing the Start button to advance time in the simulation.

## 2.2 Synchronous Programming

Let's try writing your first behaviors. First, we will focus on simple synchronous behaviors, which blocks until completion and are consequently executed in sequence as they appear in the code. Open the file `src/main/kotlin/exercises/e1/Sync.kt`. You can run any of the following behaviors by modifying the behavior in the `main` function above and execute it.

#### Task 2: Switch On

Implement the `Sync.switchOn` behavior, which switches on the led on the robot.

#### Task 3: Blink Once

Implement the `Sync.blinkOnce` behavior, which switches on the led on the robot, waits for a bit, and finally switches it off.

#### Task 4: Reach The Light

Implement the `Sync.reachTheLight` behavior, which detects the light closest to the robot and moves towards it.

#### Task 5: Iterative Behaviors

Look at the `Sync.blinking` and `Sync.followingTheLight` behaviors. The former blinks the led forever. The second reaches the light closest to the robot forever (even if it changes position).

Most useful behaviors are iterative, as they need to adapt to changes in the environment.

#### Task 6: Combining Behaviors

Look at `Sync.brokenBlinkingAndFollowingTheLight`, which supposedly combines `Sync.blinking` and `Sync.followingTheLight`.

- What do you expect from running its code?
- Can you think of a solution for correctly combining the two behaviors in `Sync.blinkingAndFollowingTheLight`?

## 2.3 Callbacks

Expressing concurrent tasks with synchronous programming is not straightforward and requires the developer to explicitly program the interleaving between tasks. With time, asynchronous programming techniques have been introduced to reduce this burden on the developers, letting them focus more on the behaviors inside their program, rather than when they are executed.

One first approach to asynchronous programming is **callbacks**. A callback is simply some code (e.g. a function) that may be executed at a later time (e.g. after a certain amount of time, after a certain event...). Open the file `src/main/kotlin/exercises/e2/Callbacks.kt`.

#### Task 7: setInterval

You are given the implementation of `Callback.setTimeout`, which executes some callback after a certain amount of time (approximately). Implement `Callback.setInterval`, which executes some callback periodically with a given period.

#### Task 8: Periodic Behaviors

Using `setInterval`, implement the `Callback.blinking` and `Callback.followingTheLight` behaviors. You make take inspiration from the corresponding synchronous behaviors.

#### Task 9: Callback Hell

The `Callback.blinkingSOS` behavior blinks the led of the robot in a specific pattern, resembling a SOS message (that is three short signals, followed by three long ones, followed by three short ones). Its implementation is an example of typical *callback hell*, where expressing the causal dependencies between behaviors make the code harder to read. Can you think of a solution to make the code more readable?

## 2.4 Async-Await and Futures

Other approaches for asynchronous programming provide different solutions to the problem of callback hell. Among these, **async-await** offers a way to promptly switch between synchronous and asynchronous programming, while **futures** offer a way to chain and compose asynchronous tasks.

#### Task 10: Async-Await

Open the file `src/main/kotlin/exercises/e3/AsyncAwait.kt`. Inside, the `async` function **defer** the execution of a callback to a (un)certain point later in time. Calling it returns a `Deferred<T>`, which represent a deferred execution producing a value of type `T` in Kotlin. The `Deferred` completes when the callback is executed. It can be **awaited** (`await`) or even **cancelled** (`cancel`) prematurely. Let's look at how the `AsyncAwait.blinkingSOS` is implemented in *async-await style*. It is much easier to read, understand, and refactor.

#### Task 11: Futures

Open the file `src/main/kotlin/exercises/e3/Future.kt`. Inside, the `setTimeout` behaves similarly to `Callback.setTimeout`, except that it returns a `CompletableFuture<T>`, which is the standard future type in Java. The future completes when the callback is executed and allows to append additional tasks to execute when the future is completed (`thenCompose`).

Let's look at how the `Future.blinkingSOS` is implemented using futures. It is quite similar to the *async-await style* in terms of readability.

## 2.5 Actors

The **actor model** was born to help with reasoning about complex scalable distributed systems. In the actor model, the unit of concurrency is the **actor**, which encapsulate its own **local state** and **behavior**. The local state can only be modified by the owner actor. The behavior describes how the actor reacts to incoming messages from other actors. The system as a whole evolves in terms of **message exchanges** between actors.

#### Task 12: Actors

Open the file `src/main/kotlin/exercises/e4/Actors.kt`. Let's have a look at the scenario together.