15. Cooperative coroutines

Description

In this chapter, we introduce *coroutines*, which represent activities that may be suspended and later resumed at the point of suspension. Suspension may take place at several points during the activity.

In its simplest form a coroutine is like a method object where execution may be temporarily suspended and later resumed.

In its most general form, it is like a parallel object in the sense that a coroutine object may execute methods and thus constitute an invocation stack (stack) as described in section . The method activation currently being executed may suspend the whole coroutine object including the stack. The difference is that two or more parallel objects may execute in parallel whereas at most one coroutine object at a time may be executed. This has the advantage that two or more coroutine objects cannot simultaneously access the same data as is the case for parallel objects.

The scheduling of coroutine objects is called *cooperative scheduling* since the programmer has to be explicit about when/where a coroutine object suspends its execution and about which coroutine object to execute/resume next.

In a later chapter, we introduce *preemptive coroutines,* where execution may be preempted. These coroutine objects are even closer to process objects. The coroutines described in this chapter is often called *cooperative coroutines.*

Cooperative coroutines can be used for a number of useful algorithms, which we will give examples of later in this chapter.

A coroutine object may suspend execution by executing a `suspend` statement:

`aCoroutineObj``.suspend`

where *`aCoroutineObj`* must be an expression that evaluates to a reference to the coroutine object that executes the statement.

Execution may later be resumed by execution of a `call` statement:

`aCoroutineObj``.call`

where *`aCoroutineObj`* must be an expression that evaluates to a reference to a coroutine object.

A sketch of using `suspend` and `call` is shown in the following example:

```
      coroutineSketch: obj
         class CoroutineEx:
            ...
L2:         this(CoroutineEx).suspend
L5:         stmt
            ...
         aCoroutineObj: ref CoroutineEx
L1:      aCoroutineObj := CoroutineEx
L3:      ...
L4:      aCoroutineObj.call
         ...
```

We have inserted labels `L1`–`L5` before some of statements in the above code to be able to refer to these in the text below.

The object `coroutineSketch` has the attributes:

- Class `CoroutineEx` describes objects that include a `suspend` statement.
- A variable reference `aCoroutineObj` of type `CoroutineEx`.
- An instance of `CoroutineEx` is generated and assigned to `aCoroutineEx` at the label `L1`.
- As part of the generation of this object, its items are executed.

- When execution of the `CoroutineEx` object arrives at L2, its execution is suspended.
- Control then returns to `coroutineSketch`, which is the *invoker* of `CoroutineEx`, and execution continues at L3.
- At L4, execution of `aCoroutineObj` is resumed by execution of the `call` statement `aCoroutineObj.call`.
- This implies that execution of `aCoroutineObj` is resumed at `stmt` L5

In the next section, we will show concrete examples of using coroutines.