Kotlin Simulation

COMP40009 - Computing Practical 1

5th–9th February 2024

Aims

- To explore inheritance and inner classes in Kotlin
- To design and implement a Kotlin framework for discrete-event simulation
- To simulate simple queueing systems and use results from queueing theory to validate those simulations

Introduction

Discrete-event simulation (DES) is a way of modelling the evolution of systems over time. It is a very general and powerful technique that is used extensively to study behavioural and performance aspects of real-world systems, such as manufacturing systems, computer networks and systems and transportation systems. You may have the opportunity to study some of these applications of DES later in your degree.

DES is based on the *scheduling* of *events* in time order. An event can be modelled as an object – an instance of an `Event` class with an invocation function `invoke()` that when called will typically:

- Update the *state* of the simulation, represented by one or more properties.
- Schedule new events, to be invoked at specified times in the future.

At the heart of every DES is a data structure that stores the sequence of events yet to be invoked, in time order. You can think of this a bit like a diary where we record events (meetings, dinner parties, fly fishing trips, etc.) so that we don't forget them. If, for example, we attend a meeting that we previously recorded in the diary we might end that meeting by agreeing a time for the next one. In that case we'll add the new meeting to our diary at the specified date in the future. Event scheduling works similarly. In this exercise we will refer to the "diary" as the *event list*.

The "execution" of a simulation involves traversing the sequence of events in time order, invoking each event in turn. The simulation terminates after some condition has been met, e.g. a certain amount of simulated time has elapsed, a particular simulation state has been reached, or a counter has tripped, e.g. one that records the number of times a particular event has been invoked. The particular termination condition will depend on the application (the simulation *model*).

Note that all time in a DES is *virtual*: the simulated passing of time has nothing to do with the passing of real time as the simulation executes. Note also that the time units are arbitrary, so we can think of each unit being a millisecond, second, hour etc. In this exercise, times will be `Doubles`.

Getting started

As per the previous exercises, use `git` to clone the repository with the skeleton files for this exercise, via

```
git clone git@gitlab.doc.ic.ac.uk:lab2324_spring/kotlinsimulation_username.git
```

Defining an abstract class for simulation

The source files that you create should be under `src/main/kotlin/simulation`. If you wish to create additional test files, you can create them under `src/test/kotlin/simulation`.

In this part you are free to choose how to split your classes and interfaces between Kotlin files. The only constraint is that—as explained below—you will implement various main functions, and there cannot be more than one main function per file.

It will be useful to separate out the notions of (virtual) *time*, *event* and *scheduler*, so define an interface for each: `Clock`, `Event` and `Scheduler`. The services these interfaces provide access to should be as follows:

- a class that implements `Clock` should yield the current time;
- it should be possible to invoke an `Event`;
- a `Scheduler` should support scheduling an event to be invoked after a certain period of time.

Now define a class `Simulator` containing the following functions:

- `currentTime(): Double`, which returns the current virtual time.
- `schedule(event: Event, dt: Double)`, which adds an (event, time) pair to the time-ordered event list. The `dt` parameter should be the time between the current time and the time the event is scheduled to be invoked, i.e. you must remember to add the current time to `dt` before adding to the event list.
- `execute()`, which should process the (event, time) pairs in the event list in order. Processing an event involves updating the simulator's current virtual time with the invocation time of the event, then calling the event's `invoke()` function. Note that time advances in discrete steps – the rationale is that no state changes happen in between events, so the time between event occurrences is not of interest. The execution should terminate when either the event list becomes empty or the function `shouldTerminate` (see below) returns `true`. You should check for termination *after* updating the simulator's virtual time in response to an event, but *before* said event is invoked, as termination may depend on a particular time being reached.
- `shouldTerminate(): Boolean`, which tells the executor (above) when to stop. The definition of the function will depend on the specific simulation model, i.e. `Simulator` instance, so the function and `Simulator` class itself should be declared abstract, having `shouldTerminate` as an abstract method.

The event list can be implemented in various ways, e.g. using the `PriorityQueue` class in Java's `java.util` library. You'll need to set up the (event, time) pairs above as instances of a class that you can also define in the same file. To equip this class with a time-based natural ordering it should implement the `Comparable` interface, i.e. overriding `compareTo`. This allows the priority queue to maintain the pairs in time order.

The suggestion is that `Simulator` implements both the clock and scheduler interfaces that you have defined, but you may prefer to experiment with separately-defined objects for managing time and event scheduling.

Writing some simple simulations

A toy simulation

We will now write a very simple simulation where an event writes a message to a print stream, and a simulation involves executing a given number of toy events.

The toy event class that you write will make reference to the `PrintStream` class from the `java.io` package. The `PrintStream` class provides a method `println` that allows a line of textual data to be written to the `PrintStream`. The Kotlin `println` function that you are familiar with simply invokes the `println` method of a special `PrintStream` object reference called `System.out`, which corresponds to the standard output of your program.

Write a class, `ToyEvent`, that has access to a `PrintStream` object. When invoked, a `ToyEvent` should print the message "A toy event occurred." to its `PrintStream`.

Write a class, `ToySimulator`, that extends the `Simulator` abstract class. A toy simulation should terminate if and only if the event list has been exhausted—there are no further conditions on termination. Your implementation of the abstract method `shouldTerminate` from `Simulator` should reflect this.

Write a `main` method in the same file as your `ToySimulator` class that schedules 10 `ToyEvent`s, with each event taking `System.out` as its `PrintStream`. Running your `main` method should lead to the "A toy event occurred." message being printed 10 times.

Test your solution so far by un-commenting and running the test case in `ToySimulation.kt` (the body of which should look very similar to the `main` method you have just written).

Simulating ticks

Your next task is to write a simple simulation that makes basic use of time.

Write a `TickEvent` class that has access to a `PrintStream` and a `Simulator`. (The `Simulator` that a `TickEvent` has access to will in practice be the simulator with which the event is associated.)

When invoked, a `TickEvent` should first print a message of the form "Tick at x", where x is the simulator's current time. Then, the `TickEvent` should schedule *another* `TickEvent` to be simulated one time unit in the future.

Write a class, `TickSimulator`, for simulating tick events. Without any further conditions, launching a simulation of `TickEvent`s will lead to an infinite number of events, because every `TickEvent` leads to another one being scheduled. Therefore, a `TickSimulator` should have a *stopping time* property, and be designed so that the simulation terminates when the simulator's current time reaches or exceeds this stopping time.

Write a `main` method in the same file as your `TickSimulator` class that schedules a single `TickEvent` at time 0.5, and has a stopping time of 10.0. Each `TickEvent` should take `System.out` as its `PrintStream`. Running your `main` method should lead to messages stating that ticks occurred at times 0.5, 1.5, ..., 9.5.

Test your solution so far by un-commenting and running the test case in `TickSimulation.kt` (the body of which should look very similar to the `main` method you have just written).

Representing ticks using an inner class

The simulation of ticks that you have just implemented has some undesirable properties. First, every time we create `TickEvent` we need to explicitly pass in a print stream object reference and a simulation object reference, which is rather cumbersome. Secondly, if working with multiple simulations in the same program, we might create two `TickEvent`s that are supposed to be associated

with the same simulation, but accidentally pass different simulation objects to them. Thirdly, and similarly, we would probably like all the events associated with a given `TickSimulator` to write to the same print stream. At present, there is nothing stopping us giving different print streams to different events associated with the same `TickSimulator`.

To overcome these problems, write a new class, `BetterTickSimulator`, in which a `TickEvent` is represented as an *inner* class. A `BetterTickSimulator` should have a `PrintStream` property. Instances of the inner `TickEvent` class should have no properties of their own. Instead, they should make use their implicit reference to their enclosing `BetterTickSimulator` object.

Write a `main` function to demonstrate your `BetterTickSimulator` in action, just as you did for `TickSimulation`: the output of this `main` function should be the same as before. Test your solution so far by un-commenting and running the test case in `BetterTickSimulation.kt` (the body of which should look very similar to the `main` method you have just written).

Generating inter-event times by random sampling

When one event schedules another, what should be the inter-event time, i.e. the `dt` above? In the simple example of simulating ticks, we used the value 1.0 as `dt`. However, in the real world times are often "random" in some sense, e.g. the times between customer arrivals at a post office are certainly not deterministic.

Sampling from uniform and exponential distributions

To model random inter-event times it is useful to implement *samplers* for various probability distributions. A sampler uses *pseudo random numbers*, each typically between 0 and 1, and transforms them into samples from a particular distribution. For example, to generate a *uniformly* distributed sample between an arbitrary a and b we simply multiply a "$U(0,1)$" random sample (a value sampled uniformly-at-random from the interval $[0,1]$) by $b-a$ and add a. Note that the mean of this distribution is $(b+a)/2$, so the long-run average of the samples generated should be the same, plus or minus some (small) variation due to the nature of random processes.

To generate samples from an *exponential* distribution with mean m, the sampling algorithm requires you to generate a $U(0,1)$ sample, u say, and return $-log_e(u) \times m$.[1] Note that the parameter of the exponential distribution is, by convention, the reciprocal of the mean, and often referred to as the *rate parameter* – more on this later. Thus, the sampling algorithm is conventionally written $-log_e(u)/r$, where $r = 1/m$ is the rate parameter. It does not matter which you choose, as long as you are consistent.

To put samplers into practice, write an interface called `TimeDelay` with a method that returns the next sample from a distribution. Write two different classes implementing this interface, one that samples uniformly from an interval $[a,b]$ that is supplied on construction, and another that samples from an exponential distribution with a rate that is supplied on construction.

In both of your classes, it should be possible to construct the sampler by passing in an object of type `java.util.Random`. The `nextDouble()` method of this object should then be used a source of random numbers. Alternatively, construction can omit a `java.util.Random`, in which case `Math.random()` should be used as a source of random numbers. There are various ways you can achieve this; one is to declare a private primary constructor that accepts a function of type `() -> Double`, and then two secondary constructors that invoke this primary constructor, passing an appropriate function reference.

A challenge associated with code that uses randomisation is that it can be challenging to test. Think about how you could meaningfully test your sampler classes, and if you have good ideas,

[1] For this lab exercise, you do not need to fully understand details of exponential distributions, but if you are interested in the details take a look at `https://en.wikipedia.org/wiki/Exponential_distribution`.

write some unit tests for them.

Feel free to research other distributions and corresponding samplers for them as classes that implement `TimeDelay` (also see the extensions below).

A random tick simulation

Write a new class, `RandomTickSimulator`, that should be similar to your `BetterTickSimulator` class, except that rather than scheduling future events after a fixed time interval, their time delay should be sampled uniformly at random.

In addition to the parameters that a `BetterTickSimulator` takes, a `RandomTickSimulator` should take a `Pair<Double, Double>` parameter specifying the interval that should be sampled from. The interval should be well-formed: both components should be non-negative, and the lower bound of the interval should be less than the upper bound.

A `BetterTickSimulator` should optionally take a `java.util.Random` object, providing a source of randomness to the uniform sampler that underpins the random simulation.

Write a `main` method in the file where you defined `BetterTickSimulator`, and launch a simulation with stopping time 10.0, an initial tick at time 0.5, and with ticks scheduled at random intervals between 1.0 and 2.0.

The output of your simulation will be something like this (but the values you see, and potentially the number of lines of output, will be different):

```
Tick at 0.5
Tick at 2.1665851025374
Tick at 3.3121867663485274
Tick at 4.474556673041025
Tick at 5.913838468403039
Tick at 7.275526878736515
Tick at 8.355925107281294
Tick at 9.941801911213624
```

Use the test in `RandomTickSimulatorTest.kt` to test your solution. Have a look at this test as an example of a best-effort way to test code that uses randomisation.

Simulating an M/M/1 queue

The remainder of this exercise is fairly challenging, as a are the extensions. You may want to consider working in pairs or with small groups of your peers. If you do this, you must still submit your own code for the solution.

One of the simplest queueing systems is a single-server queue, where "jobs" arrive from the outside world, are placed in a queue and then served one at a time in some order determined by the queueing discipline. For convenience, it is assumed that the job first in line at the queue is being served. If both the times between consecutive arrivals and the service times are exponentially distributed, and if the queueing discipline is First-In-First-Out (FIFO, also known as First-Come-First-Served) then the queue is called an M/M/1 queue: the 'M's denote "Markovian",[2] which is synonymous with "Exponential", and the '1' indicates the number of servers. The first 'M' specifies the inter-arrival time distribution and the second the service time distribution.

Your task is now to write a simulator class for M/M/1 queues. To simulate an M/M/1 queue you need two event types: job `Arrival` and service `Completion`. We could represent the queue as a `Queue` object, as per the Kotlin Queues lab exercise (see the extension below), but for now we can

[2] After Andrey Markov (https://en.wikipedia.org/wiki/Andrey_Markov).

just keep track of the number of jobs in the queue (the queue length), as an `Int`. This should be a property of the M/M/1 simulator class.

The `Arrival` event should increase the queue length by 1 and schedule the next arrival. Note that arrivals should only be scheduled on the arrival event rather than scheduling them all at once, which is inefficient. The inter-arrival time should be a sample from an exponential distribution. If the job is the only job in the queue after arrival, then it will enter service immediately. In this case, you should schedule a `Completion` event for s time units in the future, where s is sampled from an exponential distribution with a mean of the average service time.

The `Completion` event should decrease the queue length by 1. If there is at least one job left in the queue after the event has finished, its service should begin immediately. Hence, in this case it should also schedule a new completion event, if needed, similar to the above.

The means of the two exponential distributions should be parameters of the constructor, alongside the termination time. As with the tick simulations from earlier, the termination time should be used to define the `shouldTerminate` method.

As with the tick simulations, if you define the events as inner classes then the `Simulator` superclass functions, constructor arguments and queue length variable will be in scope throughout.

In order to test your simulation, add code to the M/M/1 simulator class that measures the average queue length (this was one of the extensions for the Kotlin Queue's lab). As there, do this by accumulating the area under the time-dependent queue length graph (a step function):

$$A = \sum_{n=0}^{N_{\max}} n \times T_n$$

where T_n is the time spent with queue length n and $N_{\max}$ is the maximum observed queue length. Note that you need to update the accumulator each time the queue length changes, i.e. in the `Arrival` and `Completion` events. When you're done, the mean queue length is given by A/t.

Add a `runSim` function in the class for your M/M/1 simulator. It should schedule the first `Arrival` event and then call the `execute()` function, and finally return the mean queue length.

Write a main function in the same file as your M/M/1 simulation that demonstrates a simulation in action.

Testing the M/M/1 simulation using queueing theory

In queueing theory it's common to refer to arrival and service *rates*, rather than working with means. The arrival rate is simply the reciprocal of the mean inter-arrival time, e.g. 10 jobs per second $\equiv$ a mean inter-arrival time of 1/10; similarly with service rates/means. If the arrival rate is λ and the service rate is $\mu > \lambda$ then a *beautiful* result from queueing theory is that the probability that there are n jobs in the queue is equal to $\rho^n(1-\rho)$ where $\rho = \lambda/\mu$. From this it's easy to show that the mean "long run" queue length is given by $\rho/(1-\rho)$. (See the Appendix if you are interested in the details that underpin this.)

So, if the mean inter-arrival time is 0.25 ($\lambda = 4$) and the mean service time is 0.2 ($\mu = 0.2$) the the mean long-run queue length is 4. You can test results such as this using `assertEquals` from `org.junit.Test` using the three-parameter variant, the third parameter being an `absoluteTolerance`, e.g. 0.01; the test will pass if the simulation yields a mean of 4 ± 0.01. You need to simulate for long enough for the measure to converge reliably, so experiment with increasing termination times. Interestingly, a correct implementation will only pass tests such as this with a particular probability, so beware! See also the extensions below.

To evaluate your solution, un-comment and try out the `SSQTest.kt`.

Extensions

Measurable queues

Instead of modelling the M/M/1 queue length as an integer, use a `FifoQueue` from the Kotlin Queues lab exercise. Even better, use a `Measurable` FIFO queue (see the Kotlin Queues extension), then you can delete the measurement code from the simulation, as it will be done automatically behind the scenes.

Deterministic random numbers

When testing simulations it's sometimes useful to use *repeatable* pseudo-random number sequences, which work by priming the generator with a "seed" value. If you give it the same seed it will produce the same pseudo-random numbers. The class `Random` in `java.util` works this way – the constructor takes the seed as a parameter of type `Long`.

A nice way to allow for both deterministic and non-deterministic distribution samplers is to overload the constructors of each sampler class: given the distribution parameter(s) and a `Random` sampler it should produce repeatable samples by call `Random.nextDouble()`; without the sampler it should use `Math.random()`. You can test this by adding a seed to your M/M/1 simulator and using the extra constructor to build your inter-arrival time and service time samplers.

Other samplers

Experiment with different distribution samplers. For non-exponential distributions, queueing theory delivers closed-form results in only very restricted cases. For example, in a single-server queueing system with exponentially-distributed inter-arrival times (arrival rate λ), but arbitrary service times, the mean queue length is given by

$$\lambda M_1 + \frac{\lambda^2 M_2}{2(1-\rho)}$$

where M_1 is the mean service time – the *first moment* of the service time distribution – and M_2 is the *second moment*. The second moment is equal to the expected value of the square of the random variable, $\mathbf{E}[X^2]$. This is equivalent to variance plus the square of the mean, $Var(X) + M_1^2$. For example, if you replace the exponential service time distribution sampler with a U(a,b) sampler you have $M_1 = (a+b)/2$ and $M_2 = (b^3 - a^3)/(3(b-a))$. You could research different (continuous) distributions and try them out.

Queueing networks

A *queueing network* consists of a set of interconnected queueing *nodes* which, in general may contain branches and cycles. Using the above apparatus it's easy to set up `Arrival` and `Completion` events that are parameterised by something like a node identifier. However, when a job arrives, or completes at a given node, you have to know where to send it. In other words each event has to know the topology of the network. Also, when moving jobs around the network they need to know whether or not to schedule completion events at the receiving node, based on the state of the queue at that node. It's all rather low level and messy.

A better solution is to design a set of classes tailored to the problem of defining queueing networks. For this extension, start with:

```
interface Acceptor {
    fun accept(job: Job)
}
```

which captures the notion of something, e.g. a node, being able to accept a job. The `Job` class can be based on the Kotlin Queues exercise if you wish, or you can start afresh. You'll see shortly that it's useful to know the arrival time of a job, so add that as a property.

To encode the ability to *forward* a job from one node to another define a class, with a name such as `ForwardingNode`, which will forward a job probabilistically to one of possibly many successors. Unlike the simple linear networks we explored in the Kotlin Queues lab, a queueing network may contain cycles, so you can't define the nodes and their interconnections in "one go". The idea instead is to define the nodes first and then link them separately by calling a function such as

```
fun linkTo(successors: Array<Acceptor>, probs: Array<Double>) {
   ...
}
```

where the `probs` sum to 1. If there is only one successor you might simplify things a little with an overloaded version such as `fun linkTo(node: Acceptor)`. These functions should be members of `ForwardingNode`. You now just need a forwarding function `fun forward(job: Job)` that uses the probabilities above to pick a successor node and pass the job to its `accept` function.

Now define three types of node:

1. `Source`, which repeatedly generates jobs, each with a unique id, and forwards them to a successor probabilistically[3] Thus, it's a `ForwardingNode` but not an `Acceptor`. A source works by implementing a renewal process, so it will need access to a scheduler and will need to define an inner `Event` class to model the job arrivals. You'll need an initialiser to schedule the first arrival event.

2. `Sink`, which accepts jobs, but doesn't forward them (it's *not* a `ForwardingNode`). Because each job carries its arrival time in the network you can compute the time each job spent in the network and hence the mean network *response time* – the average of the jobs' individual response times. Hence, you'll need to add a method like `fun meanResponseTime(): Double`.

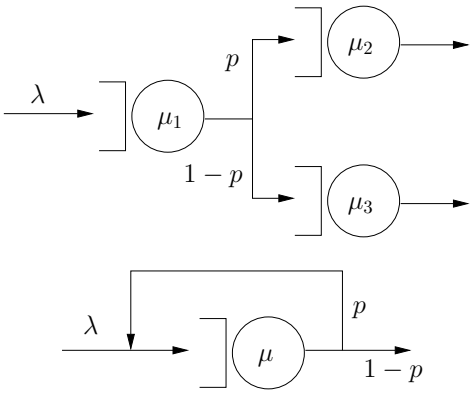
3. `QNode`, which both accepts and forwards jobs. It needs three properties, e.g. constructor arguments: a `Queue<Job>`, a `TimeDelay`, which is a sampler for job service times, and a `Simulation`, as it need to schedule events. The `accept` function should schedule a job completion if the queue is empty and enqueue the incoming job. The completion event should mirror that of the `Completion` event above: enqueue the job and schedule the completion event for the next job in the queue, if there is one. Finally, it should `forward` the dequeued job.

You can test your extension with the following queueing networks. In each case the queues (the open rectangles) are FIFO queues, the inter-arrival times must be exponentially distributed with rate parameter λ and the service times must be exponentially distributed with rate parameter shown in the corresponding circle, e.g. μ_1 for the leftmost server in the first example. In the skeleton, you may find BranchTest and CycleTest to be helpful guiding tests for this extension.

Hint: Don't confuse mean times with rates!

For the first, if you choose $p = 0.5, \lambda = 0.5, \mu_1 = 1, \mu_2 = 0.5, \mu_3 = 1/3$ then the mean response time should be 10.0. For the second, if you choose $p = 1/3, \lambda = 0.5, \mu = 1$ then the mean response time should be 6. Feel free to try more elaborate networks, but you'll need to know how to compute their (true) mean response times.

[3]This is random sampling of a discrete probability distribution and the naive algorithm is straightforward: pick a U(0,1) random number, u, and return the smallest i for which $u \geq \sum_{j=0}^{i} p_j$. There will always be such an i if the probabilities sum to 1.



Appendix

If you are interested it goes as follows: $\sum_{n=0}^{\infty} n\rho^n(1-\rho) = (1-\rho)\sum_{n=0}^{\infty} \rho\frac{d}{d\rho}\rho^n = \rho(1-\rho)\frac{d}{d\rho}\frac{1}{1-\rho} = \frac{\rho}{1-\rho}$.

Note: your repository containing the skeleton for this lab can be found at *https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinsimulation_username.* ***As always, you should use LabTS to test and submit your code.***

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```