```
Simulation
==================

Aims:

* Give the students experience working with abstract classes

* Give the students experience working with inner classes

* Give the students experience working with random number generation

* Expose the students to an interesting application area

Guide to breakdown of marks (out of 10):

- 2 marks for their solutions to "Writing some simple simulations"

- 3 marks for their solutions to "Generating inter-event times by
  random sampling"

- 5 marks for their solutions to "Simulating an M/M/1 queue"

This is just a guide - please use your judgement when deciding how to
score the exercise.
```



```
package queues

import java.util.PriorityQueue

interface Queue<T> {
    fun enqueue(item: T)

    fun peek(): T?

    fun dequeue(): T?

    fun isEmpty(): Boolean

    fun size(): Int
}

// Note the repeated code in peek, isEmpty and size (fixed in an extension)...

class FifoQueue<T> : Queue<T> {
    private val elements: MutableList<T> = mutableListOf()

    override fun enqueue(item: T) {
        elements.add(item)
    }

    override fun peek(): T? = elements.firstOrNull()

    override fun dequeue(): T? = if (isEmpty()) null else elements.removeAt(0)

    override fun isEmpty(): Boolean = elements.isEmpty()

    override fun size(): Int = elements.size
}

class LifoQueue<T> : Queue<T> {
    private val elements: MutableList<T> = mutableListOf()

    override fun enqueue(item: T) {
        elements.add(item)
    }

    override fun peek(): T? = elements.lastOrNull()

    override fun dequeue(): T? = if (isEmpty()) null else
elements.removeAt(elements.size - 1)

    override fun isEmpty(): Boolean = elements.isEmpty()

    override fun size(): Int = elements.size
}

class PrQueue<T>(comparator: Comparator<T>? = null) : Queue<T> {
    private val elements: PriorityQueue<T> = PriorityQueue(comparator)

    override fun enqueue(item: T) {
        elements.add(item)
    }

    override fun peek(): T? = elements.peek()

    override fun dequeue(): T? = elements.poll()

    override fun isEmpty(): Boolean = elements.isEmpty()

    override fun size(): Int = elements.size
}
```

```
    1: package simulation
    2:
    3: import java.io.PrintStream
    4:
    5: class BetterTickSimulator(private val printStream: PrintStream, private val
stopTime: Double) : Simulator() {
    6:     inner class TickEvent : Event {
    7:         override fun invoke() {
    8:             printStream.println("Tick at " + currentTime())
    9:             schedule(TickEvent(), 1.0)
   10:         }
   11:     }
   12:
   13:     override fun shouldTerminate() = currentTime() >= stopTime
   14: }
   15:
   16: fun main() {
   17:     val betterTickSimulator = BetterTickSimulator(System.out, 10.0)
   18:     betterTickSimulator.schedule(betterTickSimulator.TickEvent(), 0.5)
   19:     betterTickSimulator.execute()
   20: }
```



```
    1: package simulation
    2:
    3: interface Clock {
    4:     fun currentTime(): Double
    5: }
```

```
1: package simulation
2:
3: interface Event {
4:     fun invoke()
5: }
```



```
 1: package simulation
 2:
 3: import java.util.Random
 4:
 5: class ExponentialTimeDelay private constructor(
 6:     private val rate: Double,
 7:     private val rand: () -> Double,
 8: ) : TimeDelay {
 9:     constructor(rate: Double) : this(rate, { Math.random() })
10:
11:     constructor(rate: Double, generator: Random) : this(rate, {
generator.nextDouble() })
12:
13:     override fun next(): Double {
14:         return -Math.log(rand()) / rate
15:     }
16: }
```

```
    1: package simulation
    2:
    3: import queues.FifoQueue
    4: import queues.Queue
    5:
    6: // The events can sit outside the Simulator, but they will  then need to be
    7: // given an EventScheduler and some wrapper class instance for the state variables.
    8: class MU1Queue(val lambda: Double, val a: Double, val b: Double, val stopTime:
Double) : Simulator() {
    9:     private var queueLength: Int = 0
   10:     private var acc: Double = 0.0
   11:     private var t: Double = 0.0
   12:
   13:     // These are unparameterised, so can be predefined.
   14:     private val arrival = JobArrival()
   15:     private val completion = JobCompletion()
   16:
   17:     override fun shouldTerminate() = currentTime() >= stopTime
   18:
   19:     inner class JobArrival : Event {
   20:         override fun invoke() {
   21:             acc += (currentTime() - t) * queueLength
   22:             t = currentTime()
   23:             queueLength++
   24:             if (queueLength == 1) {
   25:                 schedule(JobCompletion(), UniformTimeDelay(a, b).next())
   26:             }
   27:             schedule(arrival, ExponentialTimeDelay(lambda).next())
   28:         }
   29:     }
   30:
   31:     inner class JobCompletion() : Event {
   32:         override fun invoke() {
   33:             acc += (currentTime() - t) * queueLength
   34:             t = currentTime()
   35:             queueLength--
   36:             if (queueLength > 0) {
   37:                 schedule(completion, UniformTimeDelay(a, b).next())
   38:             }
   39:         }
   40:     }
   41:
   42:     fun runSim(): Double {
   43:         schedule(JobArrival(), 0.0)
   44:         execute()
   45:         var meanQueueLength: Double = acc / currentTime()
   46:         return meanQueueLength
   47:     }
   48: }
   49:
   50: class MM1WithQueue(val lambda: Double, val mu: Double, val stopTime: Double) :
Simulator() {
   51:     private var queue = MeasurableQueue(FifoQueue<Job>(), this)
   52:     private var id = 0
   53:
   54:     // These are unparameterised, so can be predefined.
   55:     private val arrival = JobArrival()
   56:     private val completion = JobCompletion()
   57:
   58:     override fun shouldTerminate() = currentTime() >= stopTime
   59:
   60:     inner class JobArrival : Event {
   61:         override fun invoke() {
   62:             queue.enqueue(Job(id++, currentTime()))
   63:             if (queue.size() == 1) {
   64:                 schedule(JobCompletion(), ExponentialTimeDelay(mu).next())
   65:             }
   66:             schedule(arrival, ExponentialTimeDelay(lambda).next())
```



```
   67:         }
   68:     }
   69:
   70:     inner class JobCompletion() : Event {
   71:         override fun invoke() {
   72:             queue.dequeue()
   73:             if (queue.size() > 0) {
   74:                 schedule(completion, ExponentialTimeDelay(mu).next())
   75:             }
   76:         }
   77:     }
   78:
   79:     fun runSim(): Double {
   80:         schedule(JobArrival(), 0.0)
   81:         execute()
   82:         return queue.meanQueueLength()
   83:     }
   84: }
   85:
   86: fun main() {
   87:     val meanQueueLength = MU1Queue(1.0, 0.0, 1.0, 1000000.0).runSim()
   88:     print("Mean queue length = $meanQueueLength")
   89: }
   90:
   91: class MeasurableQueue<T>(
   92:     private val queue: Queue<T>,
   93:     private val clock: Clock,
   94: ) : Queue<T> by queue {
   95:     private var acc: Double = 0.0
   96:     private var t: Double = 0.0
   97:
   98:     override fun enqueue(item: T) {
   99:         acc += (clock.currentTime() - t) * queue.size()
  100:         t = clock.currentTime()
  101:         queue.enqueue(item)
  102:     }
  103:
  104:     override fun dequeue(): T? {
  105:         acc += (clock.currentTime() - t) * queue.size()
  106:         t = clock.currentTime()
  107:         return queue.dequeue()
  108:     }
  109:
  110:     // The mean queue length is the accumulated area at time t, divided by t.
  111:     fun meanQueueLength(): Double {
  112:         return acc / clock.currentTime()
  113:     }
  114: }
```

```
 1: package simulation
 2:
 3: import java.util.*
 4:
 5: // The events can sit outside the Simulator, but they will then need to be
 6: // given an EventScheduler and some wrapper class instance for the state variables.
 7: //
 8: // This version uses a random number seed to enable reproducible simulations, hence
 9: // the extra parameter to Exp. If you want non-determinism just omit the second
10: // parameter to Exp or write Exp(???).next(). You then won't need the seed.
11: class MM1Queue(
12:     lambda: Double,
13:     mu: Double,
14:     seed: Long,
15:     val stopTime: Double,
16: ) : Simulator() {
17:     private var queueLength: Int = 0
18:     private var acc: Double = 0.0
19:     private var t: Double = 0.0
20:     private val rand = Random(seed)
21:     private val interArrivalTimeSampler = ExponentialTimeDelay(lambda, rand)
22:     private val serviceTimeSampler = ExponentialTimeDelay(mu, rand)
23:
24:     // These are unparameterised, so can be predefined.
25:     private val arrival = JobArrival()
26:     private val completion = JobCompletion()
27:
28:     override fun shouldTerminate() = currentTime() >= stopTime
29:
30:     inner class JobArrival : Event {
31:         override fun invoke() {
32:             acc += (currentTime() - t) * queueLength
33:             t = currentTime()
34:             queueLength++
35:             if (queueLength == 1) {
36:                 schedule(JobCompletion(), serviceTimeSampler.next())
37:             }
38:             schedule(arrival, interArrivalTimeSampler.next())
39:         }
40:     }
41:
42:     inner class JobCompletion : Event {
43:         override fun invoke() {
44:             acc += (currentTime() - t) * queueLength
45:             t = currentTime()
46:             queueLength--
47:             if (queueLength > 0) {
48:                 schedule(completion, serviceTimeSampler.next())
49:             }
50:         }
51:     }
52:
53:     fun runSim(): Double {
54:         schedule(JobArrival(), 0.0)
55:         execute()
56:         val meanQueueLength: Double = acc / currentTime()
57:         return meanQueueLength
58:     }
59: }
60:
61: fun main() {
62:     val meanQueueLength = MM1Queue(1.0, 2.0, 12345, 100000.0).runSim()
63:     print("Mean $meanQueueLength")
64: }
```



```
 1: package simulation
 2:
 3: import queues.Queue
 4:
 5: // Could add priorities etc. as for the queues exercise, but this will do.
 6: class Job(private val id: Int, private val arrivalTime: Double) {
 7:     override fun toString() = id.toString()
 8:
 9:     fun getArrivalTime() = arrivalTime
10: }
11:
12: interface Acceptor {
13:     fun accept(job: Job)
14: }
15:
16: // Forwards jobs to anything that can accept them - the base class for a queueing
17: // network.
18: // Unlike the queues exercise, forwarding is done internally, so the forward
19: // function is not exposed (no Forwarder interface).
20: open class ForwardingNode() {
21:     private var successors: Array<Acceptor> = arrayOf()
22:     private var probs: Array<Double> = arrayOf(1.0)
23:
24:     // Forwards jobs probabilistically using the inverse transform method.
25:     // You can use the Alias Method, but this will do fine.
26:     fun forward(job: Job) {
27:         var r = Math.random()
28:         var i = 0
29:         var p = probs.get(0)
30:         while (r > p) {
31:             i++
32:             p += probs[i]
33:         }
34:         successors[i].accept(job)
35:     }
36:
37:     // Nodes can link to a single successor...
38:     fun linkTo(node: Acceptor) {
39:         successors = arrayOf(node)
40:         probs = arrayOf(1.0)
41:     }
42:
43:     // ...or multiple successors, selected probabilistically.
44:     // Pre: The probs sum to 1.
45:     fun linkTo(successors: Array<Acceptor>, probs: Array<Double>) {
46:         this.successors = successors
47:         this.probs = probs
48:     }
49: }
50:
51: // Accepts jobs and forwards them when they've been served. Accepted jobs are
queued until
52: // they're next in line, according to the queueing discipline.
53: class QNode(
54:     val queue: Queue<Job>,
55:     val delay: TimeDelay,
56:     val scheduler: Scheduler,
57: ) : Acceptor, ForwardingNode() {
58:
59:     override fun accept(job: Job) {
60:         if (queue.isEmpty()) {
61:             scheduler.schedule(JobCompletion(), delay.next())
62:         }
63:         queue.enqueue(job)
64:     }
65:
66:     inner class JobCompletion() : Event {
67:         override fun invoke() {
```

```
            // Note: Mustn't forward until after the isEmpty check, as there
            // may be a cycle.
            val job = queue.dequeue()
            if (!queue.isEmpty()) {
                scheduler.schedule(JobCompletion(), delay.next())
            }
            if (job != null) {
                forward(job)
            }
        }
    }
}

// Forwards jobs with specified inter-arrival time (iat) distribution (only).
class Source(private val iatSampler: TimeDelay, private val simulator: Simulator) :
ForwardingNode() {
    private var id = 0
    init {
        simulator.schedule(EndDelay(), iatSampler.next())
    }

    inner class EndDelay() : Event {
        override fun invoke() {
            forward(Job(id++, simulator.currentTime()))
            simulator.schedule(EndDelay(), iatSampler.next())
        }
    }
}

// Accepts jobs, but doesn't forward them.
class Sink(private val clock: Clock) : Acceptor {
    private var totalTime: Double = 0.0
    private var n: Int = 0

    override fun accept(job: Job) {
        n++
        totalTime += clock.currentTime() - job.getArrivalTime()
    }

    fun meanResponseTime(): Double {
        return totalTime / n
    }
}
```



```
package simulation

import java.io.PrintStream
import java.util.Random

class RandomTickSimulator private constructor(
    private val printStream: PrintStream,
    private val stopTime: Double,
    private val timeDelay: UniformTimeDelay,
) : Simulator() {

    constructor(
        printStream: PrintStream,
        stopTime: Double,
        interval: Pair<Double, Double>,
    ) : this(printStream, stopTime, UniformTimeDelay(interval.first,
interval.second))

    constructor(
        printStream: PrintStream,
        stopTime: Double,
        interval: Pair<Double, Double>,
        generator: Random,
    ) : this(printStream, stopTime, UniformTimeDelay(interval.first,
interval.second, generator))

    inner class TickEvent : Event {
        override fun invoke() {
            printStream.println("Tick at " + currentTime())
            schedule(TickEvent(), timeDelay.next())
        }
    }

    override fun shouldTerminate() = currentTime() >= stopTime
}

fun main() {
    val randomTickSimulator = RandomTickSimulator(System.out, 10.0, Pair(1.0, 2.0))
    randomTickSimulator.schedule(randomTickSimulator.TickEvent(), 0.5)
    randomTickSimulator.execute()
}
```

```
    1: package simulation
    2:
    3: // An event/time pair that will be placed in the event queue (priority queue). The
ordering
    4: // is based on the event invocation time.
    5: class ScheduledEvent(
    6:     val event: Event,
    7:     val time: Double,
    8: ) : Comparable<ScheduledEvent> {
    9:     override fun compareTo(other: ScheduledEvent): Int =
   10:         time.compareTo(other.time)
   11: }
```



```
    1: package simulation
    2:
    3: interface Scheduler {
    4:     fun schedule(event: Event, dt: Double)
    5: }
```

```
    1: package simulation
    2:
    3: import java.util.PriorityQueue
    4:
    5: // This implements Clock, but we could instead pass a Clock to a constructor.
    6: abstract class Simulator : Clock, Scheduler {
    7:     private val eventQueue: PriorityQueue<ScheduledEvent> =
PriorityQueue<ScheduledEvent>()
    8:     private var currentTime: Double = 0.0
    9:
   10:     override fun currentTime(): Double = currentTime
   11:
   12:     // dt is the time between current time and the time the event will be invoked.
   13:     // The ScheduledEvent contains the absolute time.
   14:     override fun schedule(event: Event, dt: Double) {
   15:         eventQueue.add(ScheduledEvent(event, currentTime + dt))
   16:     }
   17:
   18:     // If shouldTerminate depends on the time, we terminate as soon as that time is
reached.
   19:     fun execute() {
   20:         while (!eventQueue.isEmpty()) {
   21:             val nextEvent: ScheduledEvent = eventQueue.poll()
   22:             currentTime = nextEvent.time
   23:             if (shouldTerminate()) {
   24:                 break
   25:             }
   26:             nextEvent.event.invoke()
   27:         }
   28:     }
   29:
   30:     // shouldTerminate may depend on the current state and/or time.
   31:     abstract fun shouldTerminate(): Boolean
   32: }
```



```
    1: package simulation
    2:
    3: import java.io.PrintStream
    4:
    5: class TickEvent(private val printStream: PrintStream, private val simulator:
Simulator) : Event {
    6:     override fun invoke() {
    7:         printStream.println("Tick at " + simulator.currentTime())
    8:         simulator.schedule(TickEvent(printStream, simulator), 1.0)
    9:     }
   10: }
   11:
   12: class TickSimulator(private val stopTime: Double) : Simulator() {
   13:     override fun shouldTerminate() = currentTime() >= stopTime
   14: }
   15:
   16: fun main() {
   17:     val tickSimulator: Simulator = TickSimulator(10.0)
   18:     tickSimulator.schedule(TickEvent(System.out, tickSimulator), 0.5)
   19:     tickSimulator.execute()
   20: }
```

```
1: package simulation
2:
3: interface TimeDelay {
4:     fun next(): Double
5: }
```



```
 1: package simulation
 2:
 3: import java.io.PrintStream
 4:
 5: class ToyEvent(private val printStream: PrintStream) : Event {
 6:     override fun invoke() {
 7:         printStream.println("A toy event occurred.")
 8:     }
 9: }
10:
11: class ToySimulator() : Simulator() {
12:     override fun shouldTerminate(): Boolean = false
13: }
14:
15: fun main() {
16:     val toyScheduler: Simulator = ToySimulator()
17:     for (i in 1..10) {
18:         toyScheduler.schedule(ToyEvent(System.out), i.toDouble())
19:     }
20:     toyScheduler.execute()
21: }
```

```
package simulation

import java.util.Random

class UniformTimeDelay private constructor(
    private val a: Double,
    private val b: Double,
    private val rand: () -> Double,
) : TimeDelay {

    constructor(a: Double, b: Double) : this(a, b, { Math.random() })

    constructor(a: Double, b: Double, generator: Random) : this(a, b, {
generator.nextDouble() })
    override fun next(): Double {
        return rand() * (b - a) + a
    }
}
```



```
package simulation

import java.io.ByteArrayOutputStream
import java.io.PrintStream
import kotlin.test.Test
import kotlin.test.assertEquals

class BetterTickSimulatorTest {

    @Test
    fun `test better tick simulator`() {
        val outputStream = ByteArrayOutputStream()
        val printStream = PrintStream(outputStream)
        val betterTickSimulator = BetterTickSimulator(printStream, 10.0)
        betterTickSimulator.schedule(betterTickSimulator.TickEvent(), 0.5)
        betterTickSimulator.execute()

        assertEquals(
            """
                Tick at 0.5
                Tick at 1.5
                Tick at 2.5
                Tick at 3.5
                Tick at 4.5
                Tick at 5.5
                Tick at 6.5
                Tick at 7.5
                Tick at 8.5
                Tick at 9.5

            """.trimIndent(),
            outputStream.toString(),
        )
    }
}
```

```
 1: package simulation
 2:
 3: import org.junit.Test
 4: import queues.FifoQueue
 5: import kotlin.test.assertEquals
 6:
 7: class Branch(val stopTime: Double) : Simulator() {
 8:
 9:     override fun shouldTerminate() = currentTime() >= stopTime
10:
11:     fun runSim(): Double {
12:         val q1 = MeasurableQueue<Job>(FifoQueue(), this)
13:         val q2 = MeasurableQueue<Job>(FifoQueue(), this)
14:         val q3 = MeasurableQueue<Job>(FifoQueue(), this)
15:
16:         val source = Source(ExponentialTimeDelay(0.5), this)
17:         val qn1 = QNode(q1, ExponentialTimeDelay(1.0), this)
18:         val qn2 = QNode(q2, ExponentialTimeDelay(0.5), this)
19:         val qn3 = QNode(q3, ExponentialTimeDelay(1.0 / 3.0), this)
20:         val sink = Sink(this)
21:
22:         source.linkTo(qn1)
23:         qn1.linkTo(arrayOf(qn2, qn3), arrayOf(0.5, 0.5))
24:         qn2.linkTo(sink)
25:         qn3.linkTo(sink)
26:
27:         execute()
28:
29:         return (sink.meanResponseTime())
30:     }
31: }
32:
33: class BranchTest {
34:     @Test
35:     fun `matches mm1 queueing theory`() {
36:         assertEquals(Branch(10000000.0).runSim(), 10.0, 0.1)
37:     }
38: }
```



```
 1: package simulation
 2:
 3: import org.junit.Test
 4: import queues.FifoQueue
 5: import kotlin.test.assertEquals
 6:
 7: class CyclicQueue(val stopTime: Double) : Simulator() {
 8:
 9:     override fun shouldTerminate() = currentTime() >= stopTime
10:
11:     fun runSim(): Double {
12:         val q1 = MeasurableQueue<Job>(FifoQueue(), this)
13:
14:         val source = Source(ExponentialTimeDelay(0.5), this)
15:         val qn1 = QNode(q1, ExponentialTimeDelay(1.0), this)
16:         val sink = Sink(this)
17:
18:         source.linkTo(qn1)
19:         qn1.linkTo(arrayOf(qn1, sink), arrayOf(1.0 / 3.0, 2.0 / 3.0))
20:
21:         execute()
22:
23:         val meanQueueLength = q1.meanQueueLength()
24:         println("Mean queue length = $meanQueueLength")
25:         return (sink.meanResponseTime())
26:     }
27: }
28:
29: class CycleTest {
30:     @Test
31:     fun `matches mm1 queueing theory`() {
32:         assertEquals(CyclicQueue(10000000.0).runSim(), 6.0, 0.1)
33:     }
34: }
```

```kotlin
package simulation

import java.io.ByteArrayOutputStream
import java.io.PrintStream
import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertTrue

class RandomTickSimulatorTest {
    @Test
    fun `test random tick simulator`() {
        // Repeat the simulation several times
        for (repeat in 0..20) {
            val outputStream = ByteArrayOutputStream()
            val printStream = PrintStream(outputStream)
            val randomTickSimulator = RandomTickSimulator(printStream, 10.0,
Pair(1.0, 2.0))
            randomTickSimulator.schedule(randomTickSimulator.TickEvent(), 0.5)
            randomTickSimulator.execute()

            // We cannot know exactly what the simulation should produce,
            // but we can check that the first time is 0.5, that the times
            // then increase by at least 1.0 and at most 2.0, and that no
            // time exceeds 10.0.
            val components = outputStream.toString().split("\n")
            val regex = """Tick at (\d+\.\d+)""".toRegex()
            var foundLastLine = false
            var lastTime: Double? = null
            for (component in components) {
                assertFalse(foundLastLine)
                if (component == "") {
                    foundLastLine = true
                } else {
                    assertTrue(regex.containsMatchIn(component))
                    val matchResult = regex.find(component)
                    val (matchString) = matchResult!!.destructured
                    val time = matchString.toDouble()
                    assertTrue(time <= 10.0)
                    if (lastTime == null) {
                        assertEquals(0.5, time)
                    } else {
                        assertTrue(time >= lastTime + 1.0)
                        assertTrue(time <= lastTime + 2.0)
                    }
                    lastTime = time
                }
            }
        }
    }
}
```



```kotlin
package simulation

import org.junit.Test
import kotlin.test.assertEquals

class SSQTest {
    @Test
    fun `matches mm1 queueing theory`() {
        // assertEquals(MM1WithQueue(4.0, 5.0, 1000000.0).runSim(), 4.0, 0.01)
        assertEquals(MM1Queue(4.0, 5.0, 12345, 1000000.0).runSim(), 4.0, 0.01)
    }
}
```

```kotlin
package simulation

import java.io.ByteArrayOutputStream
import java.io.PrintStream
import kotlin.test.Test
import kotlin.test.assertEquals

class TickSimulatorTest {

    @Test
    fun `test tick simulator`() {
        val outputStream = ByteArrayOutputStream()
        val printStream = PrintStream(outputStream)

        val tickSimulator: Simulator = TickSimulator(10.0)
        tickSimulator.schedule(TickEvent(printStream, tickSimulator), 0.5)
        tickSimulator.execute()

        assertEquals(
            """
                Tick at 0.5
                Tick at 1.5
                Tick at 2.5
                Tick at 3.5
                Tick at 4.5
                Tick at 5.5
                Tick at 6.5
                Tick at 7.5
                Tick at 8.5
                Tick at 9.5

            """.trimIndent(),
            outputStream.toString(),
        )
    }
}
```



```kotlin
package simulation

import java.io.ByteArrayOutputStream
import java.io.PrintStream
import kotlin.test.Test
import kotlin.test.assertEquals

class ToySimulatorTest {

    @Test
    fun `test toy simulator`() {
        val outputStream = ByteArrayOutputStream()
        val printStream = PrintStream(outputStream)

        val toySimulator: Simulator = ToySimulator()
        for (i in 1..10) {
            toySimulator.schedule(ToyEvent(printStream), i.toDouble())
        }
        toySimulator.execute()

        assertEquals(
            """
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.
                A toy event occurred.

            """.trimIndent(),
            outputStream.toString(),
        )
    }
}
```