Kotlin Queues

COMP40009 - Computing Practical 1

22nd – 26th January 2024

Aims

- To explore interfaces and basic generics in Kotlin
- To implement polymorphic queues with various queueing disciplines
- To implement a simple linear network of queues

Introduction

Queues are ubiquitous in real-world systems. Their principal job is to organise entities (people, vehicles, jobs etc.) that are waiting to access a resource (post office counter, road section, CPU etc.).

Entities in a queue can either said to be waiting, or receiving service. Once an entity has finished receiving service, it exits the queue. In a queueing-theoretic sense, queues are more general than what we traditionally think of as a queue, and also encompass stacks or systems with entities that are selected based on priority.

The *queueing discipline* determines the order in which waiting entities are allocated to the resource, with some common ones being:

- First-In-First-Out (FIFO), or First-Come-First-Served (FCFS). This is how we typically think of queues in common parlance (e.g. queueing in a shop).
- Last-In-First-Out (LIFO). This is less common, but features in systems that involve stacking, e.g. storage areas in container ports.
- Priority-based. This is common in computer systems and networks, where the job/thread/packet etc. with the highest priority is served next. In this exercise priorities will be derived from an ordering on the values of a given data type, with lower values representing *higher* priorities.

Complex systems may involve multiple queueing *nodes* where the entities move from one node to another according to some routing scheme, which may be deterministic, state-dependent or probabilistic. Interconnected nodes may form a *queueing network*. An interesting problem in Computer Science and Operations Research is the simulation of such networks, which can be used to explore and optimise real-world systems.

In this exercise we will use Kotlin's interfaces, together with predefined collection classes to implement simulations of queues with the above queueing disciplines. The second part of the exercise will develop some simple Kotlin classes for defining linear networks of queueing nodes.

Getting started

As per the previous exercises, get the skeleton files from GitLab. You can clone your repository with the following command (remember to replace the *username* with your own username).

`git clone https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinqueues_username.git`

What to do: Part 1

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

Implementing FIFO and LIFO queues

The methods that are common to all queue types can be encapsulated in a Kotlin interface.

Create a file, `Queues.kt`, starting with a package declaration to make it part of the `queues` package.

In this file, define a generic interface, `Queue<T>` (where `T` represents an arbitrary type that can be defined when a queue is instantiated), with the following abstract method declarations:

- `enqueue`, which should add an item of type `T` to a queue.
- `peek`, which should return the next item in the queue, as determined by the queueing discipline, without removing it. If `peek` is called on an empty queue, `null` should be returned.
- `dequeue`, which should remove and return the next item in the queue, as determined by the queueing discipline. If `dequeue` is called on an empty queue, `null` should be returned.
- `isEmpty`, which should return `true` if the queue is empty and `false` otherwise.
- `size`, which should return the number of items in the queue.

Polymorphism is where we handle values of different types in a uniform way. This interface describes a *polymorphic* queue because it can work with elements of *any* given type `T`. This kind of polymorphism is known as *parametric polymorphism*.

In the same file, define classes `FifoQueue<T>` and `LifoQueue<T>`. These should implement the `Queue<T>` interface, and use the FIFO and LIFO queueing disciplines described above, respectively.

You can use any of Kotlin's existing classes to implement the container for the queued items, although the suggestion is to use `MutableList`s for these queue types.

Note: You may notice that the `enqueue`, `isEmpty` and `size` methods are implemented identically in the FIFO and LIFO queues. It is possible to avoid this duplication (see the extensions) but you can leave it like that for now.

Testing your FIFO and LIFO queues

The `QueueTests.kt` test file contains test classes `FifoQueueTests` and `LifoQueueTests`. Initially, each contains a single commented-out test that checks a very basic property of the corresponding queue implementation. Uncomment these simple tests and check that they pass. Then, add numerous additional tests that thoroughly test the behaviour of these two queue implementations.

Writing comprehensive tests is a key part of this lab, so be sure to take it seriously and think carefully about the tests that you write.

Make sure that your tests feature queues that use data types other than just `Int`. Make sure your tests exercise all of the methods of the `Queue` interface. If you are using IntelliJ, you can use the "run tests with coverage" option to see which lines of code your tests are hitting. Use

this feedback as the basis for writing additional tests that increase coverage, aiming for 100% statement coverage.

Try to avoid *redundant* tests: each test should check a specific property of your implementations, and should avoid any "clutter" that is not related to checking that property.

Give your tests descriptive names so that their intent is clear.

Implementing a priority queue for naturally-ordered types

Now, back in `Queues.kt` write a class `PrQueue<T>` that implements the `Queue<T>` interface using the priority-based discipline defined above.

To achieve this, your class should use a Java `PriorityQueue<T>` behind the scenes (from the `java.util` package). A Java priority queue works out-of-the-box for any type `T` that has a *natural ordering*.

Various basic types such as `Int`, `Char` and `String` have a natural ordering. For instance, for the `Int` type, 1 is ordered before 2, 2 before 3, and so on. For the `Char` type, digit characters ('0', '1', '2', etc.) are ordered before uppercase letters ('A', 'B', 'C', etc.), which are in turn ordered before lower case letters ('a', 'b', 'c', etc.). The `Char` ordering provides the basis for ordering strings, which are compared character-by-character. If a string s is a strict prefix of a string t then s is ordered before t.

More generally, any type `T` that implements the `Comparable<T>` interface has a natural ordering.

Java's `PriorityQueue` prioritises values that are lower in the natural ordering.

For now, you can simply assume that the type `T` with respect to which your `PrQueue<T>` class is polymorphic has a natural ordering.

Testing your priority queue for naturally ordered types

Look at the `PrQueueTests` class in the `QueueTests.kt` file. It contains two tests that are commented out, and a note to add more tests.

For now, uncomment the first, basic test and check that it passes, and then, as above, write comprehensive additional tests that thoroughly test your `PrQueue<T>` implementation various built-in types that have a natural ordering.

Again, be comprehensive but avoid redundant tests, and give your tests descriptive names.

Enhancing priority queues to work with custom orderings

Uncomment out the other test that was provided in the `PrQueueTests` class – `'queue implements Priority for Point'`.

This test defines a priority queue of `Point` objects, where `Point` is a simple data class defined lower down in the file.

If you run this test you should find that it fails with an error along the lines of:

```
class queues.Point cannot be cast to class java.lang.Comparable
```

This is because `Point` does *not* have a natural ordering.

Your task now is to enhance your `PrQueue<T>` class so that it can prioritise with respect to a given *custom* ordering on elements of type `T`. For example, for the `Point` class we might want to order points *lexicographically* so that they are first compared by their x coordinates, and only when their x coordinates match are they then compared by their y coordinates.

Look at the Kotlin `Comparator<T>` interface, which is actually just an alias for the Java `Comparator<T>` interface about which you can find details online. It defines a single method, `compare`, that takes two elements of type `T` and returns:

- a negative integer if the first element should be deemed less than the second element;
- zero if the elements should be deemed equal;
- a positive integer if the first element should be deemed greater than the second element.

This allows a custom ordering to be defined on any type `T`.

Kotlin's `PriorityQueue<T>` class can optionally be constructed with a `Comparator<T>` object, i.e. an object that implements the `Comparator<T>` interface and thus provides a custom ordering on elements of type `T`. If provided, this ordering will be used for prioritisation. If the type `T` already has a natural ordering *and* a custom ordering is provided via a comparator, the custom ordering will be used. If `null` is passed as the optional comparator argument, then `T`'s natural ordering (if it exists) will be used, just as if no comparator argument had been provided to the `PriorityQueue<T>` constructor.

Adapt your `PrQueue<T>` class so that it takes an *optional* parameter: a possibly-null `Comparator<T>` object. Use this given comparator, if provided, to equip the Java `PriorityQueue<T>` that backs your `PrQueue<T>` with a custom ordering.

In the `QueueTests.kt` file, write a `PointComparator` class that implements the `Comparator<T>` interface for `T=Point`. Implement the `compare` method in `PointComparator` so that it compares points first by their x coordinates and then by their y coordinates if their x coordinates match.

Adapt the `'queue implements Priority for Point'` test so that it creates a `PrQueue<Point>` object using a `PointComparator` instance.

If you have done things correctly, you should now find that the test passes.

What to do: Part 2

Chaining queues

We can form a rudimentary (linear) network of queues by chaining together instances of the above queue classes. The idea is for a node in such a network to be able to *accept* items (of type `T`) to be enqueued and `forward` items from the node's queue to the next node in the chain, by dequeuing an item and asking the next node in the chain to accept it. We will see how to chain together numerous queues that potentially use multiple different queueing disciplines between them. Because all of the queue classes implement the common `Queue<T>` interface, the queueing network classes that you will write can treat them uniformly. This is another example of *polymorphism*, which is called *subtype polymorphism*: each of `FifoQueue<T>`, `LifoQueue<T>` and `PrQueue<T>` is a *subtype* of the more general type `Queue<T>`.

In a new file, `Network.kt`, define interfaces `Acceptor<T>` and `Forwarder` to capture the notion of a node in the network accepting an item (via an `accept` method), and forwarding an item to the next node in the chain (via a `forward` method).

Write a class `QueueNode<T>` that implements both interfaces. To declare that your Kotlin class implements multiple interfaces, simply list all the interfaces that it implements after the "implements" colon between the closing bracket of the primary constructor and the opening curly brace of the class body. A class that implements multiple interfaces must provide implementations of the union of the abstract methods and properties that the interfaces declare.

The `QueueNode<T>` constructor should take the queue (of type `Queue<T>`) and successor node (of type `Acceptor<T>`) as parameters. The `accept` method should enqueue a given item in the queue

and `forward` should dequeue an item from the queue and pass the dequeued item to the successor node's `accept` method. If there is nothing available to be dequeued then `forward` should have no effect (it is a "no-op").

For testing purposes it is useful to define a terminal node `Sink<T>` that implements `Acceptor<T>` but not `Forwarder`. Its role is simply to record the items that it has accepted in a list. A `Sink` should contain an additional method, `getAccepted()`, returning the items that have been accepted.

With all this in place you should be able to build a chain "back to front", starting with a `Sink` and chaining `QueueNode`s from right to left in succession.[1]

The `NetworkTests.kt` file contains a number of commented-out test cases illustrating how linear queueing networks should work.

Examining these tests may help you to understand the requirements of the `QueueNode<T>`, `Acceptor<T>` and `Forwarder` classes and interfaces described above, and you should refine and debug your solution until these tests pass.

Finally, write some additional tests to try out further combinations of queues in queueing networks, to thoroughly test that your solution is working.

Extensions

These parts are optional. If you have the above working you might like to try one or more of these.

Subclassing queues

If you implement both FIFO and LIFO queues using `MutableList`s, or similar, then you may notice that the `enqueue`, `isEmpty` and `size` methods are implemented identically. Duplicate code is something we usually wish to avoid, so try fixing it by defining an *abstract* subclass of `Queue` that implements the common code. The FIFO and LIFO queues will then become subclasses of your new abstract class. Abstract classes are a standard feature of object-oriented languages and will be covered in the course in due time. For now, feel free to read about them online or in a Kotlin or Java textbook.

Although parts of your priority queue class may *look* similar to parts of your FIFO and LIFO queue classes, think about why it is not possible to avoid this perceived duplication.

Measurable queues

Queueing systems are much more interesting when there is a notion of time. Define a class `Clock` that keeps track of *virtual* time in the form of a `Double`. A method `advanceTime` should be used to advance the clock by a specified amount of time.

Now add a new class `MeasurableQueue` – another implementation of `Queue` – whose properties include a `Queue` (one of the above, for example) and a `Clock`. It should essentially work as a wrapper for the given `Queue`, but with additional code in the `enqueue` and `dequeue` functions for measuring the average (mean) population of the queue, measured over time. To do that, accumulate the area under the time-dependent population graph (a step function):

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

[1] Note that this only works because our networks are linear. If we allowed cycles then would have to first define the nodes and then add their interconnections separately.

where T_n is the time spent with population n and $N_{\max}$ is the maximum observed population. Note that you need to update the accumulator each time the queue population changes. When you are done, the mean population is given by A/t, where t is the time on the clock at the end. For example, given `val fifoNode = QueueNode(fifoQueue, sink)` with `fifoQueue` and `sink` suitable defined, the sequence:

```
fifoNode.accept(1)
clock.advanceTime(10.0)
fifoNode.accept(2)
clock.advanceTime(5.0)
fifoNode.accept(3)
clock.advanceTime(10.0)
fifoNode.forward()
clock.advanceTime(10.0)
fifoNode.forward()
clock.advanceTime(2.0)
fifoNode.forward()
```

should yield a mean population given by `fifoQueue.meanPop()` $= 72/37 = 1.9459$ to 4 d.p., as the queue spent 12 time units with population 1, 15 with population 2 and 10 with population 3.

Delegation

You may notice that your `MeasurableQueue` has to "delegate" the implementations of `peek`, `isEmpty` and `size` to the encapsulated queue. A neat trick in Kotlin is to implement `Queue<T>` with *delegation* using the '`by`' keyword, e.g. using `... : Queue<T> by queue`. If you do this you can delete the definitions of the above functions. Try it out.

Note: your repository containing the skeleton for this lab can be found at *`https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinqueues_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.
```