```
Queues
==================

Aims:

* Give the students experience of specifying an interface (Queue)

* Have the students practice writing a couple of simple
  implementations (FifoQueue and LifoQueue)

* Get the students to think about how to write good test cases for
  these classes

* Introduce the concept of a priority queue

* Introduce the Comparator interface, and give the students experience
  implementing this interface

* Illustrate the concepts of default parameters and nullable types

* Investigate implementing multiple interfaces

Guide to breakdown of marks (out of 10):

- 1 mark for Queue interface and FIFO and LIFO implementations.

- 2 marks for writing solid tests for these classes

- 2 marks for writing and testing priority queues for
  naturally-ordered types

- 2 marks for enhancing priority queues to work with custom orderings

- 3 marks for correctly implementing the various classes needed to
  chain queues together

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



```
<?xml version="1.0" encoding="utf-8"?>
<checkstyle version="8.0">
</checkstyle>
```

```
1: <?xml version="1.0" encoding="utf-8"?>
2: <checkstyle version="8.0">
3: </checkstyle>
```



```
 1: package queues
 2:
 3: //
 4: // Students can put these where they wish, so this file will not be part of the
skeleton.
 5: //
 6:
 7: // List-based queues...
 8:
 9: abstract class ListBasedQueue<T> : Queue<T> {
10:     protected val elements = mutableListOf<T>()
11:
12:     override fun enqueue(item: T) {
13:         elements.add(item)
14:     }
15:
16:     override fun isEmpty(): Boolean = elements.isEmpty()
17:
18:     override fun size(): Int = elements.size
19: }
20:
21: class FifoQueueExtension<T> : ListBasedQueue<T>() {
22:     override fun dequeue(): T? = if (isEmpty()) null else elements.removeAt(0)
23:
24:     override fun peek(): T? = elements.firstOrNull()
25: }
26:
27: class LifoQueueExtension<T> : ListBasedQueue<T>() {
28:     override fun dequeue(): T? = if (isEmpty()) null else
elements.removeAt(elements.size - 1)
29:
30:     override fun peek(): T? = elements.lastOrNull()
31: }
32:
33: // Add time...
34:
35: class Clock {
36:     private var time = 0.0
37:
38:     fun currentTime(): Double = time
39:
40:     fun advanceTime(dt: Double) {
41:         time += dt
42:     }
43: }
44:
45: // This version delegates by queue...
46:
47: class MeasurableQueue<T>(private val queue: Queue<T>, private val clock: Clock) :
Queue<T> by queue {
48:     private var acc: Double = 0.0
49:     private var t: Double = 0.0
50:
51:     override fun enqueue(item: T) {
52:         acc += (clock.currentTime() - t) * queue.size()
53:         t = clock.currentTime()
54:         queue.enqueue(item)
55:     }
56:
57:     override fun dequeue(): T? {
58:         acc += (clock.currentTime() - t) * queue.size()
59:         t = clock.currentTime()
60:         return queue.dequeue()
61:     }
62:
63: // If you don’t have the ’by queue’ delegation above then you need to define these
as well.
64: //
```

```
65: //    override fun peek(): T? = queue.peek()
66: //
67: //    override fun isEmpty(): Boolean = queue.isEmpty()
68: //
69: //    override fun size(): Int = queue.size()
70:
71:     // The mean population is the accumulated area at time t, divided by t.
72:     fun meanPop(): Double {
73:         return acc / clock.currentTime()
74:     }
75: }
```



```
 1: package queues
 2:
 3: interface Acceptor<T> {
 4:     fun accept(item: T)
 5: }
 6:
 7: interface Forwarder {
 8:     fun forward()
 9: }
10:
11: class QueueNode<T>(private val queue: Queue<T>, private val successor: Acceptor<T>)
: Acceptor<T>, Forwarder {
12:     override fun accept(item: T) {
13:         queue.enqueue(item)
14:     }
15:
16:     override fun forward() {
17:         queue.dequeue()?.let { successor.accept(it) }
18:     }
19: }
20:
21: class Sink<T> : Acceptor<T> {
22:     private val accepted: MutableList<T> = mutableListOf()
23:
24:     override fun accept(item: T) {
25:         accepted.add(item)
26:     }
27:
28:     fun getAccepted(): List<T> = accepted
29: }
```

```kotlin
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
}
```



```kotlin
package queues

import org.junit.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertNull
import kotlin.test.assertTrue

class FifoQueueExtensionTests {
    @Test
    fun `queue implements FIFO for Int`() {
        val fifoQueue = FifoQueueExtension<Int>()

        fifoQueue.enqueue(1)
        fifoQueue.enqueue(3)
        fifoQueue.enqueue(2)
        assertEquals(1, fifoQueue.dequeue())
        assertEquals(3, fifoQueue.dequeue())
        assertEquals(2, fifoQueue.dequeue())
    }

    @Test
    fun `test peek`() {
        val fifoQueue = FifoQueueExtension<Char>()
        assertNull(fifoQueue.peek())
        fifoQueue.enqueue('A')
        assertEquals('A', fifoQueue.peek())
        fifoQueue.enqueue('B')
        assertEquals('A', fifoQueue.peek())
        fifoQueue.enqueue('C')
        assertEquals('A', fifoQueue.peek())
        fifoQueue.dequeue()
        assertEquals('B', fifoQueue.peek())
        fifoQueue.dequeue()
        assertEquals('C', fifoQueue.peek())
        fifoQueue.dequeue()
        assertNull(fifoQueue.peek())
    }

    @Test
    fun `test size`() {
        val fifoQueue = FifoQueueExtension<String>()
        assertEquals(0, fifoQueue.size())
        fifoQueue.enqueue("A")
        assertEquals(1, fifoQueue.size())
        fifoQueue.enqueue("B")
        assertEquals(2, fifoQueue.size())
        fifoQueue.enqueue("C")
        assertEquals(3, fifoQueue.size())
        fifoQueue.dequeue()
        assertEquals(2, fifoQueue.size())
        fifoQueue.dequeue()
        assertEquals(1, fifoQueue.size())
        fifoQueue.dequeue()
        assertEquals(0, fifoQueue.size())
    }

    @Test
    fun `test isEmpty`() {
        val fifoQueue = FifoQueueExtension<Int>()
        assertTrue(fifoQueue.isEmpty())
        fifoQueue.enqueue(0)
        assertFalse(fifoQueue.isEmpty())
        fifoQueue.enqueue(1)
        assertFalse(fifoQueue.isEmpty())
        fifoQueue.enqueue(2)
        assertFalse(fifoQueue.isEmpty())
        fifoQueue.dequeue()
```

```
69:         assertFalse(fifoQueue.isEmpty())
70:         fifoQueue.dequeue()
71:         assertFalse(fifoQueue.isEmpty())
72:         fifoQueue.dequeue()
73:         assertTrue(fifoQueue.isEmpty())
74:     }
75: }
76:
77: class LifoQueueExtensionTests {
78:     @Test
79:     fun `queue implements LIFO for Int`() {
80:         val lifoQueue = LifoQueueExtension<Int>()
81:
82:         lifoQueue.enqueue(1)
83:         lifoQueue.enqueue(3)
84:         lifoQueue.enqueue(2)
85:         assertEquals(2, lifoQueue.dequeue())
86:         assertEquals(3, lifoQueue.dequeue())
87:         assertEquals(1, lifoQueue.dequeue())
88:     }
89:
90:     @Test
91:     fun `test peek`() {
92:         val lifoQueue = LifoQueueExtension<Char>()
93:         assertNull(lifoQueue.peek())
94:         lifoQueue.enqueue('A')
95:         assertEquals('A', lifoQueue.peek())
96:         lifoQueue.enqueue('B')
97:         assertEquals('B', lifoQueue.peek())
98:         lifoQueue.enqueue('C')
99:         assertEquals('C', lifoQueue.peek())
100:        lifoQueue.dequeue()
101:        assertEquals('B', lifoQueue.peek())
102:        lifoQueue.dequeue()
103:        assertEquals('A', lifoQueue.peek())
104:        lifoQueue.dequeue()
105:        assertNull(lifoQueue.peek())
106:    }
107:
108:    @Test
109:    fun `test size`() {
110:        val lifoQueue = LifoQueueExtension<String>()
111:        assertEquals(0, lifoQueue.size())
112:        lifoQueue.enqueue("A")
113:        assertEquals(1, lifoQueue.size())
114:        lifoQueue.enqueue("B")
115:        assertEquals(2, lifoQueue.size())
116:        lifoQueue.enqueue("C")
117:        assertEquals(3, lifoQueue.size())
118:        lifoQueue.dequeue()
119:        assertEquals(2, lifoQueue.size())
120:        lifoQueue.dequeue()
121:        assertEquals(1, lifoQueue.size())
122:        lifoQueue.dequeue()
123:        assertEquals(0, lifoQueue.size())
124:    }
125:
126:    @Test
127:    fun `test isEmpty`() {
128:        val lifoQueue = LifoQueueExtension<Int>()
129:        assertTrue(lifoQueue.isEmpty())
130:        lifoQueue.enqueue(0)
131:        assertFalse(lifoQueue.isEmpty())
132:        lifoQueue.enqueue(1)
133:        assertFalse(lifoQueue.isEmpty())
134:        lifoQueue.enqueue(2)
135:        assertFalse(lifoQueue.isEmpty())
136:        lifoQueue.dequeue()
```



```
137:        assertFalse(lifoQueue.isEmpty())
138:        lifoQueue.dequeue()
139:        assertFalse(lifoQueue.isEmpty())
140:        lifoQueue.dequeue()
141:        assertTrue(lifoQueue.isEmpty())
142:    }
143: }
144:
145: class MeasurableFifoQueueTest {
146:     @Test
147:     fun `Measurable FIFO queue computes mean OK`() {
148:         val clock = Clock()
149:         val queue = MeasurableQueue<Int>(FifoQueue(), clock)
150:         val sink = Sink<Int>()
151:         val fifoNode = QueueNode(queue, sink)
152:
153:         fifoNode.accept(1)
154:         clock.advanceTime(10.0)
155:         fifoNode.accept(2)
156:         clock.advanceTime(5.0)
157:         fifoNode.accept(3)
158:         clock.advanceTime(10.0)
159:         fifoNode.forward()
160:         clock.advanceTime(10.0)
161:         fifoNode.forward()
162:         clock.advanceTime(2.0)
163:         fifoNode.forward()
164:
165:         assertEquals(queue.meanPop(), 72.0 / 37.0, 0.001)
166:     }
167: }
```

```
 1: package queues
 2:
 3: import kotlin.test.Test
 4: import kotlin.test.assertEquals
 5:
 6: class NetworkTests {
 7:     @Test
 8:     fun ‘FIFO queue and sink combine OK‘() {
 9:         val sink: Sink<Int> = Sink()
10:         val fifoNode = QueueNode(FifoQueue(), sink)
11:
12:         fifoNode.accept(1)
13:         fifoNode.accept(2)
14:         fifoNode.accept(3)
15:         fifoNode.forward()
16:         fifoNode.forward()
17:         fifoNode.forward()
18:
19:         assertEquals(listOf(1, 2, 3), sink.getAccepted())
20:     }
21:
22:     @Test
23:     fun ‘FIFO and LIFO queues combine OK‘() {
24:         val sink: Sink<Int> = Sink()
25:         val lifoNode = QueueNode(LifoQueue(), sink)
26:         val fifoNode = QueueNode(FifoQueue(), lifoNode)
27:
28:         fifoNode.accept(1)
29:         fifoNode.accept(2)
30:         fifoNode.forward()
31:         fifoNode.accept(3)
32:         fifoNode.forward()
33:         lifoNode.forward()
34:         fifoNode.forward()
35:         lifoNode.forward()
36:         lifoNode.forward()
37:
38:         assertEquals(listOf(2, 3, 1), sink.getAccepted())
39:     }
40:
41:     @Test
42:     fun ‘LIFO, Priority and FIFO queues combine OK‘() {
43:         val sink: Sink<Int> = Sink()
44:         val lifoNode = QueueNode(LifoQueue(), sink)
45:         val prQueueNode = QueueNode(PrQueue(), lifoNode)
46:         val fifoNode = QueueNode(FifoQueue(), prQueueNode)
47:
48:         fifoNode.accept(3)
49:         fifoNode.accept(1)
50:         fifoNode.forward()
51:         fifoNode.accept(2)
52:         fifoNode.accept(0)
53:         prQueueNode.forward()
54:         fifoNode.forward()
55:         fifoNode.forward()
56:         lifoNode.forward()
57:         fifoNode.forward()
58:         prQueueNode.forward()
59:         prQueueNode.forward()
60:         lifoNode.forward()
61:         lifoNode.forward()
62:         prQueueNode.forward()
63:         lifoNode.forward()
64:
65:         assertEquals(listOf(3, 1, 0, 2), sink.getAccepted())
66:     }
67: }
```



```
 1: package queues
 2:
 3: import kotlin.test.Test
 4: import kotlin.test.assertEquals
 5: import kotlin.test.assertFalse
 6: import kotlin.test.assertNull
 7: import kotlin.test.assertTrue
 8:
 9: class FifoQueueTests {
10:     @Test
11:     fun ‘queue implements FIFO for Int‘() {
12:         val fifoQueue = FifoQueue<Int>()
13:
14:         fifoQueue.enqueue(1)
15:         fifoQueue.enqueue(3)
16:         fifoQueue.enqueue(2)
17:         assertEquals(1, fifoQueue.dequeue())
18:         assertEquals(3, fifoQueue.dequeue())
19:         assertEquals(2, fifoQueue.dequeue())
20:     }
21:
22:     @Test
23:     fun ‘test peek‘() {
24:         val fifoQueue = FifoQueue<Char>()
25:         assertNull(fifoQueue.peek())
26:         fifoQueue.enqueue(’A’)
27:         assertEquals(’A’, fifoQueue.peek())
28:         fifoQueue.enqueue(’B’)
29:         assertEquals(’A’, fifoQueue.peek())
30:         fifoQueue.enqueue(’C’)
31:         assertEquals(’A’, fifoQueue.peek())
32:         fifoQueue.dequeue()
33:         assertEquals(’B’, fifoQueue.peek())
34:         fifoQueue.dequeue()
35:         assertEquals(’C’, fifoQueue.peek())
36:         fifoQueue.dequeue()
37:         assertNull(fifoQueue.peek())
38:     }
39:
40:     @Test
41:     fun ‘test size‘() {
42:         val fifoQueue = FifoQueue<String>()
43:         assertEquals(0, fifoQueue.size())
44:         fifoQueue.enqueue("A")
45:         assertEquals(1, fifoQueue.size())
46:         fifoQueue.enqueue("B")
47:         assertEquals(2, fifoQueue.size())
48:         fifoQueue.enqueue("C")
49:         assertEquals(3, fifoQueue.size())
50:         fifoQueue.dequeue()
51:         assertEquals(2, fifoQueue.size())
52:         fifoQueue.dequeue()
53:         assertEquals(1, fifoQueue.size())
54:         fifoQueue.dequeue()
55:         assertEquals(0, fifoQueue.size())
56:     }
57:
58:     @Test
59:     fun ‘test isEmpty‘() {
60:         val fifoQueue = FifoQueue<Int>()
61:         assertTrue(fifoQueue.isEmpty())
62:         fifoQueue.enqueue(0)
63:         assertFalse(fifoQueue.isEmpty())
64:         fifoQueue.enqueue(1)
65:         assertFalse(fifoQueue.isEmpty())
66:         fifoQueue.enqueue(2)
67:         assertFalse(fifoQueue.isEmpty())
68:         fifoQueue.dequeue()
```

```
        assertFalse(fifoQueue.isEmpty())
        fifoQueue.dequeue()
        assertFalse(fifoQueue.isEmpty())
        fifoQueue.dequeue()
        assertTrue(fifoQueue.isEmpty())
    }
}

class LifoQueueTests {
    @Test
    fun `queue implements LIFO for Int`() {
        val lifoQueue = LifoQueue<Int>()

        lifoQueue.enqueue(1)
        lifoQueue.enqueue(3)
        lifoQueue.enqueue(2)
        assertEquals(2, lifoQueue.dequeue())
        assertEquals(3, lifoQueue.dequeue())
        assertEquals(1, lifoQueue.dequeue())
    }

    @Test
    fun `test peek`() {
        val lifoQueue = LifoQueue<Char>()
        assertNull(lifoQueue.peek())
        lifoQueue.enqueue('A')
        assertEquals('A', lifoQueue.peek())
        lifoQueue.enqueue('B')
        assertEquals('B', lifoQueue.peek())
        lifoQueue.enqueue('C')
        assertEquals('C', lifoQueue.peek())
        lifoQueue.dequeue()
        assertEquals('B', lifoQueue.peek())
        lifoQueue.dequeue()
        assertEquals('A', lifoQueue.peek())
        lifoQueue.dequeue()
        assertNull(lifoQueue.peek())
    }

    @Test
    fun `test size`() {
        val lifoQueue = LifoQueue<String>()
        assertEquals(0, lifoQueue.size())
        lifoQueue.enqueue("A")
        assertEquals(1, lifoQueue.size())
        lifoQueue.enqueue("B")
        assertEquals(2, lifoQueue.size())
        lifoQueue.enqueue("C")
        assertEquals(3, lifoQueue.size())
        lifoQueue.dequeue()
        assertEquals(2, lifoQueue.size())
        lifoQueue.dequeue()
        assertEquals(1, lifoQueue.size())
        lifoQueue.dequeue()
        assertEquals(0, lifoQueue.size())
    }

    @Test
    fun `test isEmpty`() {
        val lifoQueue = LifoQueue<Int>()
        assertTrue(lifoQueue.isEmpty())
        lifoQueue.enqueue(0)
        assertFalse(lifoQueue.isEmpty())
        lifoQueue.enqueue(1)
        assertFalse(lifoQueue.isEmpty())
        lifoQueue.enqueue(2)
        assertFalse(lifoQueue.isEmpty())
        lifoQueue.dequeue()
```



```
        assertFalse(lifoQueue.isEmpty())
        lifoQueue.dequeue()
        assertFalse(lifoQueue.isEmpty())
        lifoQueue.dequeue()
        assertTrue(lifoQueue.isEmpty())
    }
}

class PrQueueTests {
    @Test
    fun `queue implements Priority for Int`() {
        val prQueue = PrQueue<Int>()

        prQueue.enqueue(1)
        prQueue.enqueue(3)
        prQueue.enqueue(2)
        assertEquals(1, prQueue.dequeue())
        assertEquals(2, prQueue.dequeue())
        assertEquals(3, prQueue.dequeue())
    }

    @Test
    fun `test peek`() {
        val prQueue = PrQueue<Char>()
        assertNull(prQueue.peek())
        prQueue.enqueue('B')
        assertEquals('B', prQueue.peek())
        prQueue.enqueue('C')
        assertEquals('B', prQueue.peek())
        prQueue.enqueue('A')
        assertEquals('A', prQueue.peek())
        prQueue.dequeue()
        assertEquals('B', prQueue.peek())
        prQueue.dequeue()
        assertEquals('C', prQueue.peek())
        prQueue.dequeue()
        assertNull(prQueue.peek())
    }

    @Test
    fun `test size`() {
        val prQueue = PrQueue<String>()
        assertEquals(0, prQueue.size())
        prQueue.enqueue("C")
        assertEquals(1, prQueue.size())
        prQueue.enqueue("B")
        assertEquals(2, prQueue.size())
        prQueue.enqueue("A")
        assertEquals(3, prQueue.size())
        prQueue.dequeue()
        assertEquals(2, prQueue.size())
        prQueue.dequeue()
        assertEquals(1, prQueue.size())
        prQueue.dequeue()
        assertEquals(0, prQueue.size())
    }

    @Test
    fun `test isEmpty`() {
        val prQueue = PrQueue<Int>()
        assertTrue(prQueue.isEmpty())
        prQueue.enqueue(0)
        assertFalse(prQueue.isEmpty())
        prQueue.enqueue(0)
        assertFalse(prQueue.isEmpty())
        prQueue.enqueue(0)
        assertFalse(prQueue.isEmpty())
        prQueue.dequeue()
```

```
205:         assertFalse(prQueue.isEmpty())
206:         prQueue.dequeue()
207:         assertFalse(prQueue.isEmpty())
208:         prQueue.dequeue()
209:         assertTrue(prQueue.isEmpty())
210:     }
211:
212:     @Test
213:     fun `queue implements Priority for Point`() {
214:         val prQueue = PrQueue<Point>(PointComparator())
215:
216:         prQueue.enqueue(Point(2, 2))
217:         prQueue.enqueue(Point(1, 1))
218:         prQueue.enqueue(Point(0, 1))
219:         prQueue.enqueue(Point(1, 2))
220:         prQueue.enqueue(Point(0, 2))
221:         prQueue.enqueue(Point(1, 0))
222:         prQueue.enqueue(Point(2, 0))
223:         prQueue.enqueue(Point(0, 0))
224:         prQueue.enqueue(Point(2, 1))
225:
226:         assertEquals(Point(0, 0), prQueue.dequeue())
227:         assertEquals(Point(0, 1), prQueue.dequeue())
228:         assertEquals(Point(0, 2), prQueue.dequeue())
229:         assertEquals(Point(1, 0), prQueue.dequeue())
230:         assertEquals(Point(1, 1), prQueue.dequeue())
231:         assertEquals(Point(1, 2), prQueue.dequeue())
232:         assertEquals(Point(2, 0), prQueue.dequeue())
233:         assertEquals(Point(2, 1), prQueue.dequeue())
234:         assertEquals(Point(2, 2), prQueue.dequeue())
235:     }
236: }
237:
238: data class Point(val coordX: Int, val coordY: Int)
239:
240: class PointComparator : Comparator<Point> {
241:     override fun compare(p0: Point, p1: Point): Int {
242:         val compareCoordX = p0.coordX.compareTo(p1.coordX)
243:         if (compareCoordX != 0) return compareCoordX
244:         return p0.coordY.compareTo(p1.coordY)
245:     }
246: }
```