```kotlin
package journeyplan

class Route(val segments: List<Segment>) {

  override fun toString(): String {
    val result = StringBuilder()

    result.append("${segments.first().from} to ${segments.last().to} - ${duration()} minutes, ${numChanges()} changes\n")

    var current = segments.first()
    result.append(" - ${current.from} to ")

    for (seg in segments) {
      if (seg.line != current.line) {
        result.append("${seg.from} by ${current.line}\n")
        result.append(" - ${seg.from} to ")
        current = seg
      }
    }

    result.append("${segments.last().to} by ${current.line}")

    return result.toString()
  }

  fun duration(): Int = segments.sumOf(Segment::minutes)

  fun numChanges(): Int = changes().size

  fun allInterchangesOpen(): Boolean = changes().all(Station::canInterchange)

  private fun changes(): List<Station> =
    segments.zipWithNext().filter { (l1, l2) -> l1.line != l2.line }.map { (l1, _) -> l1.to }
}

class SubwayMap(private val segments: List<Segment>) {

  private val nowhereToGo = listOf(Route(emptyList()))

  fun routesFrom(
    startingPoint: Station,
    destination: Station,
    optimisingFor: (Route) -> Int = Route::duration
  ): List<Route> =
    routesFrom(startingPoint, destination, alreadyVisited = setOf(startingPoint))
      .filter(Route::allInterchangesOpen)
      .sortedBy(optimisingFor)

  private fun routesFrom(startingPoint: Station, destination: Station, alreadyVisited: Set<Station>): List<Route> {

    if (startingPoint == destination) {
      return nowhereToGo
    }

    val nextSteps = segments.filter { seg -> seg.from == startingPoint }
      .filterNot { seg -> seg.to in alreadyVisited }
      .filterNot { seg -> seg.line.isSuspended() }

    return nextSteps.flatMap { seg ->
      routesFrom(seg.to, destination, alreadyVisited + startingPoint)
```

```kotlin
        .map { r -> Route(listOf(seg) + r.segments) }
    }
  }
}
```

```kotlin
package journeyplan

class Station(private val name: String) {

  private var open = true

  fun canInterchange(): Boolean {
    return open
  }

  fun open() {
    open = true
  }

  fun close() {
    open = false
  }

  override fun toString(): String {
    return name
  }
}

class Line(private val name: String) {

  private var suspended: Boolean = false

  override fun toString(): String {
    return "$name Line"
  }

  fun suspend() {
    this.suspended = true
  }

  fun resume() {
    suspended = false
  }

  fun isSuspended(): Boolean {
    return suspended
  }
}

class Segment(val from: Station, val to: Station, val line: Line, val minutes: Int)
```

```kotlin
package journeyplan

import org.junit.Assert.assertEquals
import org.junit.Assert.assertNotNull
import org.junit.Assert.assertNull
import org.junit.Assert.assertTrue
import org.junit.Test

class ExtensionsTest {

  /** Uncomment the code in this file if you do the extensions **/

  val piccadillyLine = Line("Piccadilly")
  val victoriaLine = Line("Victoria")
  val districtLine = Line("District")

  val southKensington = Station("South Kensington")
  val knightsbridge = Station("Knightsbridge")
  val hydeParkCorner = Station("Hyde Park Corner")
  val greenPark = Station("Green Park")
  val oxfordCircus = Station("Oxford Circus")
  val victoria = Station("Victoria")
  val sloaneSquare = Station("Sloane Square")

  fun londonUnderground(): SubwayMap = SubwayMap(
    listOf(
      Segment(southKensington, knightsbridge, piccadillyLine, 3),
      Segment(knightsbridge, hydeParkCorner, piccadillyLine, 4),
      Segment(hydeParkCorner, greenPark, piccadillyLine, 2),
      Segment(greenPark, oxfordCircus, victoriaLine, 1),
      Segment(greenPark, victoria, victoriaLine, 1),
      Segment(victoria, greenPark, victoriaLine, 1),
      Segment(victoria, sloaneSquare, districtLine, 6),
      Segment(sloaneSquare, southKensington, districtLine, 3),
      Segment(southKensington, sloaneSquare, districtLine, 6),
      Segment(sloaneSquare, victoria, districtLine, 6)
    )
  )

  val map = londonUnderground()

  @Test
  fun ‘can find multiple routes between stations‘() {

    val routes = map.routesFrom(southKensington, victoria)
    assertEquals(2, routes.size)

    assertTrue(routes[0].segments.all { s -> s.line in setOf(piccadillyLine,
victoriaLine) })
    assertTrue(routes[1].segments.all { s -> s.line == districtLine })
  }

  @Test
  fun ‘can optimise for number of changes‘() {

    val routes = map.routesFrom(southKensington, victoria, optimisingFor =
Route::numChanges)
    assertEquals(2, routes.size)

    assertEquals(0, routes[0].numChanges())
    assertEquals(1, routes[1].numChanges())
  }

  @Test
  fun `can optimise for duration`() {

    val routes = map.routesFrom(southKensington, victoria, optimisingFor = Route::duration)
    assertEquals(2, routes.size)

    assertEquals(10, routes[0].duration())
    assertEquals(12, routes[1].duration())
  }

  @Test
  fun `does not offer routes with suspended lines`() {

    var routes = map.routesFrom(southKensington, victoria)

    assertEquals(2, routes.size)
    assertTrue(routes[0].segments.all { s -> s.line in setOf(piccadillyLine, victoriaLine) })
    assertTrue(routes[1].segments.all { s -> s.line == districtLine })

    districtLine.suspend()

    routes = map.routesFrom(southKensington, victoria)

    assertEquals(1, routes.size)
    assertTrue(routes[0].segments.none { s -> s.line == districtLine })

    districtLine.resume()

    routes = map.routesFrom(southKensington, victoria)

    assertEquals(2, routes.size)
    assertTrue(routes[0].segments.all { s -> s.line in setOf(piccadillyLine, victoriaLine) })
    assertTrue(routes[1].segments.all { s -> s.line == districtLine })
  }

  @Test
  fun `avoids interchange at closed stations`() {

    var routes = map.routesFrom(southKensington, oxfordCircus)
    assertEquals(2, routes.size)

    victoria.close()

    routes = map.routesFrom(southKensington, oxfordCircus)

    assertEquals(1, routes.size)
    assertDoesNotGoVia(victoria, routes[0])
  }

  @Test
  fun `does not avoid closed stations if interchange not required`() {

    var routes = map.routesFrom(southKensington, oxfordCircus)
    assertEquals(2, routes.size)

    sloaneSquare.close()

    routes = map.routesFrom(southKensington, oxfordCircus)
    assertEquals(2, routes.size)
    println(routes)
```

```

    assertGoesVia(sloaneSquare, routes[1])
  }

  fun assertGoesVia(station: Station, route: Route) {
    assertNotNull(findIn(route, station))
  }

  fun assertDoesNotGoVia(station: Station, route: Route) {
    assertNull(findIn(route, station))
  }

  fun findIn(route: Route, station: Station) = route.segments.find { s -> s.to == station }
}
```

```kotlin
package journeyplan

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

class RoutePlannerTest {

  val northernLine = Line("Northern")
  val victoriaLine = Line("Victoria")
  val centralLine = Line("Central")

  val highgate = Station("Highgate")
  val archway = Station("Archway")
  val tufnellPark = Station("Tufnell Park")
  val kentishTown = Station("Kentish Town")
  val camden = Station("Camden Town")
  val euston = Station("Euston")
  val warrenStreet = Station("Warren Street")
  val oxfordCircus = Station("Oxford Circus")
  val bondStreet = Station("Bond Street")

  val tufnellParkToHighgate =
    Route(
      listOf(
        Segment(tufnellPark, archway, northernLine, 3),
        Segment(archway, highgate, northernLine, 3)
      )
    )

  val highgateToOxfordCircus =
    Route(
      listOf(
        Segment(highgate, archway, northernLine, 3),
        Segment(archway, kentishTown, northernLine, 3),
        Segment(kentishTown, camden, northernLine, 3),
        Segment(camden, euston, northernLine, 3),
        Segment(euston, warrenStreet, victoriaLine, 3),
        Segment(warrenStreet, oxfordCircus, victoriaLine, 3)
      )
    )

  val camdenToBondStreet =
    Route(
      listOf(
        Segment(camden, euston, northernLine, 3),
        Segment(euston, warrenStreet, victoriaLine, 3),
        Segment(warrenStreet, oxfordCircus, victoriaLine, 3),
        Segment(oxfordCircus, bondStreet, centralLine, 2)
      )
    )

  @Test
  fun `can calculate number of changes`() {
    assertEquals(0, tufnellParkToHighgate.numChanges())
    assertEquals(1, highgateToOxfordCircus.numChanges())
    assertEquals(2, camdenToBondStreet.numChanges())
  }

  @Test
  fun `can calculate total duration`() {
    assertEquals(6, tufnellParkToHighgate.duration())
    assertEquals(18, highgateToOxfordCircus.duration())
    assertEquals(11, camdenToBondStreet.duration())
```

```kotlin
  }

  @Test
  fun `toString omits calling points`() {
    assertEquals(
      """
                Tufnell Park to Highgate - 6 minutes, 0 changes
                 - Tufnell Park to Highgate by Northern Line
      """.trimIndent(),
      tufnellParkToHighgate.toString()
    )
  }

  @Test
  fun `toString shows changes`() {
    assertEquals(
      """
                Highgate to Oxford Circus - 18 minutes, 1 changes
                 - Highgate to Euston by Northern Line
                 - Euston to Oxford Circus by Victoria Line
      """.trimIndent(),
      highgateToOxfordCircus.toString()
    )
  }
}
```

```kotlin
package journeyplan

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

class TravelModelTest {

  @Test
  fun ‘printing stations shows their names‘() {
    assertEquals("South Kensington", Station("South Kensington").toString())
    assertEquals("Knightsbridge", Station("Knightsbridge").toString())
  }

  @Test
  fun ‘printing lines shows their names‘() {
    assertEquals("District Line", Line("District").toString())
    assertEquals("Circle Line", Line("Circle").toString())
  }
}
```