# Kotlin Journey Planning

COMP40009 - Computing Practical 1

21st – 25th November 2022

## Aims

- To explore the basics of object-oriented programming.
- To provide experience designing and implementing your own classes in Kotlin.
- To implement a route-finding algorithm.

## Introduction

Cities all around the world have complex transport networks, and often provide route planning apps to help passengers find the best route from A to B. Route planning is also a common feature provided by apps like Google Maps or CityMapper. There are many different algorithms that can be used to plan routes intelligently, and you will look at these in courses later in your degree. In this lab exercise we'll look at how we can model a transport network, and implement a simple planner to find routes from place to place.

Transport networks are suprisingly complex when you get into the details, and if you want to plan a route that involves a tram, two trains, a bus and a boat journey, you may have to integrate data from many different systems from many different companies. There has been a lot of work in the past coming up with standards to model these things and allow different companies to exchange data, for example the European TransModel standard[1], or the TransXChange model for UK bus routes[2]. Things get even more complicated when you start to incorporate real-time data about where buses and trains actually are, and whether they are running on time[3].

One of the main things that these standards do is to give us a common vocabulary that we can use to talk about public transport, and when we write applications, we can define data-types that map to these to model the domain in a consistent way. We won't use the full TransModel for this exercise – that would be overkill – but we will create a simple model in the same style.

### Stations and Lines

We will model a subway or metro network for a city. In London we have the London Underground or "tube" network, and we will give examples based on that network, but this model should work for any city if you provide the relevant data.

Consider the network as a collection of *stations*, each of which may be on one or more *lines*, so for example South Kensington station is on the District Line, and it is also on the Piccadilly Line. Stations are linked by line *segments*. We will consider the route map to be made up from a collection of segments, each joining two stations via particular line. Each segment has a certain

[1] `http://www.transmodel-cen.eu/`
[2] `https://www.gov.uk/government/collections/transxchange`
[3] `http://www.transmodel-cen.eu/standards/siri/`

average time that it takes for a train to traverse that link. From these building blocks, we can build up a route map that will allow us to compute routes from one station to another.

# Getting started

As per the previous exercises, use `git` to clone the repository with the skeleton files for this exercise using the following command (remember to replace the *`username`* with your own username).

`git clone https://gitlab.doc.ic.ac.uk/lab2223_autumn/kotlinjourneyplanner_username.git`

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[4]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

`7z x skel.7z -pC20900BA`

(*n.b., C20900BA is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named "skel", because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.

Open the code in IntelliJ to work on it. You should do most of your work in the files `TravelModel.kt` and `RoutePlanner.kt` under the `src/main/kotlin/journeyplan` directory.

# What to do: Part 1

## Modelling the Domain

In object-oriented programming we design types that model the problem domain so that we can create and manipulate objects in our programs that represent objects in the real world. To do this we create *classes* that define new types. In the file `TravelModel.kt`, define three classes:

- `Station` which just has a property holding the station's name (e.g. "South Kensington").
- `Line` which also just has a property holding the line's name (e.g. "Piccadilly").
- `Segment` which has properties for two `Station`s, the `Line` which links them, and an integer number of minutes representing the average time for a train to travel along this segment.

If you try to print a `Station` or `Line` (using `println()`), you'll see that by default they don't print in a very meaningful way. Provide an implementation of the `toString()` function for both of these classes so that objects print as "South Kensington" or "Piccadilly Line" etc.

## Setting Up the Map

In the file `RoutePlanner.kt` define a class `SubwayMap`. This should be initialised (via the constructor) with a list of segments. In our model, segments are one-way, meaning that if you want to link two stations in both directions, you need two segments (think of this like two physical tracks, so trains can pass each other, and going in one direction may take longer than the other).

Imagine setting up the tube map from the picture on the following page. To specify the full thing would take a large number of lines of code, so we won't do that here. To test your journey planner, try setting up a small network of a few tube stations (or stations on a transport network you know well) connected together by appropriate line segments.

To separate setting up the map from the rest of the code, write a function `londonUnderground()` that builds a `SubwayMap` with your chosen stations and segments, and returns it.

[4] `https://www.7-zip.org/download.html`

Also in the file `RoutePlanner.kt` define a class `Route` with a constructor that takes a list of segments. A `Route` then will represent a list of segments to traverse, in order, to get from a particular origin to a particular destination.

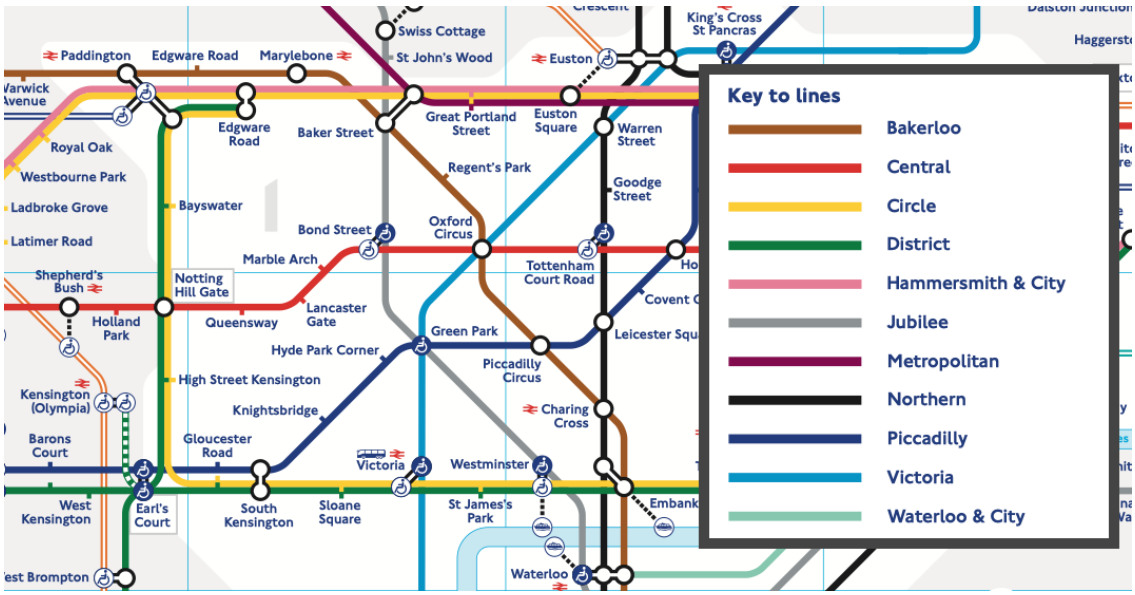


Figure 1: Part of the London Underground map – you can find the full map online.

# What to do: Part 2

## Route Finding

Given our map, we would like to find possible routes to get from one station to another. One way that we could do this would be to implement the following algorithm:

1. Find all of the segments we can follow directly from the origin station.
2. Exclude any segments that would cause us to revisit stations that we've already been to (to start with, just the origin).
3. For each of these segments, find all the routes from the end of that segment to our target destination. Each of these routes will be a route from the origin to the destination if we simply prepend the first segment.
4. To find these sub-routes, recursively apply the same algorithm.
5. If the origin and the destination are the same, we don't need to travel anywhere.

Define a method on the class `SubwayMap` with the following signature:

```
fun routesFrom(origin: Station, destination: Station): List<Route>
```

Implement this method according to the algorithm described above. Some tips:

- You may find it useful to define a (private) helper method to help with recursion.
- If the origin and destination are the same, rather than thinking that there are no routes, perhaps think that there is a route but it requires traversing no segments.

- Think back to Haskell and have a look at all the higher-order functions that are available to you as methods of the `List` type.

## Displaying Routes

If you write a `main()` function you can try out your route planner with some code like:

```
val map = londonUnderground()

println(map.routesFrom(southKensington, oxfordCircus))
```

But if you do this, again you will see that the routes don't print in a very meaningful way. Implement a `toString()` function for `Route`.

To start with, print each segment. So for a route from South Kensington to Green Park via Victoria, it might print as:

```
South Kensington to Green Park
 - South Kensington to Sloane Square by District Line
 - Sloane Square to Victoria by District Line
 - Victoria to Green Park by Victoria Line
```

Adding the string "\n" will insert a new line.

When you have this working, improve the implementation to summarise the details of segments where you stay on the same train, so the above example would instead be printed as:

```
South Kensington to Green Park
 - South Kensington to Victoria by District Line
 - Victoria to Green Park by Victoria Line
```

## Testing

As in the previous lab, you will find some tests under the `src/test/kotlin/journeyplan` directory. You can uncomment these as you work through the exercise to test your code, but note that this test suite is not complete - we have just given you few tests to start you off.

**Add more tests covering different cases to check that your code works correctly.**

## Calculating Route Statistics

For a given route, we likely want to know how long it would take to go that way, or how many times we need to change train, in case we are carrying heavy bags.

Add a method to `Route` to return the total duration for a particular route, i.e. the time to traverse all of the segments.

Add another method to return the total number of changes for a particular route, i.e. where you have to change from one line to another to continue the journey.

Update the `toString()` function to show these statistics when you print a route, e.g.:

```
South Kensington to Green Park - 13 minutes, 1 changes
 - South Kensington to Victoria by District Line
 - Victoria to Green Park by Victoria Line
```

When you calculate the list of potential routes, sort them by overall duration (shortest first) before returning them from `routesFrom()`.

# Extensions

These parts are optional. If you have got the first two parts working and want to explore more, try some of these extensions.

## Traveller Preferences

Travellers may not always favour the quickest route. They might want to minimise changes, or avoid particular stations.

Add a third parameter to the `routesFrom()` method on `SubwayMap`. This parameter should be a *function* from `Route` to integer, that gives us a characteristic by which to sort the list of routes (e.g. by duration).

Enhance your `routesFrom()` method to sort the returned routes according to this characteristic. You may find it convenient to provide a *default value*[5] for this parameter in the argument list, so that if you want to minimise duration (which is probably the most common case) then you don't need to pass anything.

You could also try adding another optional parameter with a list of specific stations to avoid. Or you could enhance the model to include data like whether there is step-free access, or whether lines run at particular times, and take these into account when determining the best route to suggest.

## Line Suspensions

Sometimes there are problems with the trains or the tracks and lines may be suspended for a time. Add a method `suspend()` to `Line` which puts a line into a suspended state. When a line is suspended, the route planner should not suggest any route that uses that line. Add another method `resume()` which puts the line back into normal service.

## Interchange Closures

Sometimes there are problems at individual stations (e.g. over-crowding or a fire alert) so trains can't stop there. This means that travellers can't change lines at this station. Add a method `close()` to `Station` that puts the station into a closed state. When a station is closed, no routes should be suggested that require an interchange at that station – but travelling through the station on the same line is still ok. Add a complementary method `open()` to re-open a station after closure.

[5] `https://kotlinlang.org/docs/reference/functions.html#default-arguments`

***Note: your repository containing the skeleton for this lab can be found at*** *https://gitlab.doc.ic.ac.uk/lab2223_autumn/kotlinjourneyplanner_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.
```