Test Cases

Input Validation:

1. **MapPlanner class**

StreetEntry constructor

- The input streetId should not be null or empty.
- The input from and to points should not be null.
- The fromNode and toNode should not be null.

getNode

- The input position should not be null.
- The key generated by concatenating the X and Y coordinates should not be null or empty.
- The node should not be null.

depotLocation

- The input depot should not be null.
- The street identified by depot's streetId should exist in the streets map.

addStreet

- The input streetId should not be null or empty.
- The input start and end points should not be null.
- The fromNode and toNode of StreetEntry should not be null.
- The street identified by the streetId should not already exist in the streets map.
- The start and end points of the street should not be the same.

getStreetId

- The input streets should not be null or empty.
- The input current node should not be null.

getFurtherestNode

- Before calling the getStreetByStreetId() method, validate that currentLocation.getStreetId() is not null or empty.
- Before accessing the position field of a node, validate that it is not null.
- Before adding a NodeEntry to the priority queue, validate that the node is not null.
- Before calling the getStreetId() method, validate that the streets parameter is not null or empty.

routeNoLeftTurn

- Check that the destination parameter is not null before using it.
- Check that the graph is not null before accessing it.

- Check that the graph.get("depot") call returns a non-null value before using it as the starting node for Dijkstra's algorithm.
- Check that each StreetEntry object in the graph has a valid from and to node that exist in the graph before accessing their positions.
- Check that the distance and previous maps are not null before accessing or updating them.
- Check that the queue is not null before adding or removing elements from it.
- Check that the getTurnDirection() method is called with valid StreetEntry, prevNode, and currentNode parameters.

2. SubRoute class

subrouteStart

- No input validation is needed for this method.

subrouteEnd

- No input validation is needed for this method.

extractRoute

- No input validation is needed for this method.

3. Route class

addStreet

- addStreet method should accept non-empty string values for both streetId and turnDirection parameters.
- addStreet method should throw an exception if either streetId or turnDirection parameter is null or an empty string.

getStreets

- getStreets method should return the list of streets added using the addStreet method.

reverseStreets

- No input validation required for this method.

appendTurn

- appendTurn method should accept non-null values for both turn and streetTurnedOnto parameters.
- appendTurn method should throw an exception if either turn or streetTurnedOnto parameter is null.

turnOnto

- turnOnto method should accept a positive integer value for legNumber parameter.
- turnOnto method should throw an exception if legNumber is less than 1 or greater than the number of locations in the locations list.

turnDirection

- turnDirection method should accept a positive integer value for legNumber parameter.
- turnDirection method should throw an exception if legNumber is less than or equal to 0 or greater than the number of locations in the locations list.

legs

- No input validation required for this method.

length

- No input validation required for this method.

getLoops

- No input validation required for this method.

4. Point Class

Point constructor

- Check that the x and y parameters are not null.
- Check that the x and y parameters are integers.

distanceTo

- Check that the to parameter is not null.

turnType

- Check that the turnAt and turnTo parameters are not null.
- Check that the degreeTolerance parameter is greater than 0.

5. Node class

addStreet

- The streetId, from, and to parameters should be validated to ensure that they are not null.
- The from and to points should also be validated.
- The streetId should be validated to ensure that it is unique for the node, as adding a duplicate street may cause errors later on.

Boundary test cases:

1. **MapPlanner class**

StreetEntry constructor

- streetId is null or empty
- from and to are null
- from is null
- to is null
- nodes is null
- from is a point not in the map
- to is a point not in the map

getNode

- position is null
- position is a point not in the map

MapPlanner constructor

- degrees is less than 0

depotLocation

- depot is null
- streetId in depot is null or empty
- currentLocation is not set

addStreet

- streetId is null or empty
- start and end are null
- start is null
- end is null
- fromNode is null
- toNode is null

getFurtherestNode

- distances is null
- distances is empty

getStreetId

- streets is null
- current is null

furthestStreet

- When the nodes are empty.
- When the current location is null.
- When the current location is not on any street.
- When the current location is on a street.
- When there are multiple furthest streets from the current location.

routeNoLeftTurn

- When the graph is empty.
- When the startNode or destination is null.
- When the startNode or destination is not in the graph.
- When there is no path from the startNode to the destination.
- When there is a path from the startNode to the destination, it should return the correct route with no left turns.
- When there are multiple paths from the startNode to the destination.

2. SubRoute Class

Constructor

- Route is null
- startLeg is negative
- endLeg is negative
- startLeg is greater than endLeg
- startLeg is greater than the last leg of Route
- endLeg is greater than the last leg of Route

subrouteStart

- When the subroute starts at the first leg of Route
- When the subroute starts at some other leg of Route

subrouteEnd

- When the subroute ends at the last leg of Route
- When the subroute ends at some other leg of Route

extractRoute

- When the subroute has only one leg
- When the subroute has multiple legs
- When startLeg and endLeg are the same
- When startLeg is the first leg of Route
- When endLeg is the last leg of Route
- When Route has loops

3. **Route class**

addStreet

- Empty streetId and valid turnDirection.
- Valid streetId and empty turnDirection.
- Empty parameters.
- Null streetId and valid turnDirection.
- Valid streetId and null turnDirection.
- Null parameters.
- streetId and turnDirection of length 1.
- streetId and turnDirection of maximum length.
- Multiple valid addStreet() calls.

getStreets

- Empty streets list.
- Non-empty streets list.

reverseStreets

- Empty streets list.
- List of size 1.
- List of size 2.
- List of size greater than 2.
- Multiple reverseStreets() calls.

appendTurn

- Valid TurnDirection and streetTurnedOnto parameters.
- Null TurnDirection and valid streetTurnedOnto parameter.
- Valid TurnDirection and null streetTurnedOnto parameter.
- Null parameters.
- Empty locations list.
- Locations list containing one location.
- Locations list containing multiple locations.
- Multiple appendTurn() calls.

turnOnto

- LegNumber = 1.
- LegNumber = legs() (last leg).
- LegNumber greater than legs().
- LegNumber less than 1.
- Empty locations list.
- Multiple turnOnto() calls.

turnDirection

- LegNumber = 1.
- LegNumber = legs() (last leg).
- LegNumber greater than legs().
- Test with legNumber less than 1.
- Empty locations list.
- Locations list containing only one location.
- Locations list containing multiple locations.
- Multiple turnDirection() calls.

legs

- Empty locations list.
- Locations list containing one location.
- Locations list containing multiple locations.
- Multiple legs() calls.

length

- Empty points list.
- Points list containing one point.
- Points list containing multiple points.
- Multiple length() calls.

List<SubRoute> loops

- Empty locations list.
- Locations list containing one location.
- Locations list containing multiple locations without any loops.
- Locations list containing one loop.
- Locations list containing multiple loops.
- Nested loop.