Informatics Large Practical

Coursework 2

s1807827

11 December 2020

Table of Contents

Software Architecture

The purpose of this report is to discuss the implementation of an application designed to program an autonomous drone which will collect readings from air quality sensors distributed around an urban geographical area as part of a research project to analyze urban air quality. This section will primarily focus on the architecture of said application, and how the various constituent components of the application interface with one another to enable the drone to fly through the geographical area, thus collecting readings from the aforementioned air quality sensors. During its flight, the drone is also programmed to avoid flying through four buildings deemed as “No-Fly Zones”.

The structure of the application is a series of Java classes that interact with one another to enable the drone to fly and function. The entry point to the application is the main **App** class. In its main method, command-line arguments are saved into variables, which determine: the day on which the drone should fly, the drone’s starting position, and connectivity details for the web server containing the sensors’ and No-Fly Zones’ locations. The drone is then instantiated, its flightpath is generated, and it then proceeds to fly around the geographical area taking readings from the sensors while dodging the No-Fly Zones, before returning close to its starting point. This class contains the main drone algorithm, which will be further elaborated upon in the drone algorithm section of this report. This class is also the central point where all other classes’ objects and methods interact with one another, enabling much of the logic to be encapsulated away from view in other classes, while seamlessly integrating together to remove swaths of complexity.

As the primary function of this application is to enable the drone to fly around the geographical area, the **Drone** class controls its behavior and attributes. Various instance variables model the state of the drone, such as - the direction in which it’s facing, a detailed list of all the moves it has made, a list of all the sensors it has taken readings from, and more. The class also contains a series of methods that alter the drone’s state, such as - the ability for the drone to move in a given direction, the ability to take the reading from a sensor, the ability to determine if it will enter a No-Fly Zone and collide with a building, and more. In addition to that, there are several methods that enable the user to derive information about the drone state, as per the instance variables aforementioned.

The **MoveData** class is ancillary to the drone class, in that its only purpose is to keep a detailed record of all of the moves the drone has performed - start and end locations, the direction the drone travelled in, the sensor that has been read, and more. In each drone move, a new **MoveData** object is instantiated, and added to an ArrayList of **MoveData** objects held on board the drone’s flight computer. The class contains a number of methods to derive the constituent components of each move. Once the drone has finished its flight, this list of moves is then extracted and output to a text file in a format that is readable by human beings.

While the **Drone** class models the state and behavior of the drone, the **FlightPlan** class determines the permutation in which to visit the sensors to minimize the distance travelled. Its main instance variable is the permutation of the sensors, although there are others which enable it to find the optimal path. Apart from the constructor, there are two public methods. The first is **tourValue()**, which returns the total distance travelled over the given flightplan. While it can be queried by the user, its main purpose is to be used to help determine the optimal flightpath. The other public method is **tuneFlightPlan()**, which is supported by an array of private methods that together calculate the optimal flight path.

Another ancillary class is the **Coordinate** class, which models geographic locations. Although the Earth is an oblate spheroid, it is sufficient for the purposes of this task to model it as a flat sheet, with locations encoded in a pair of (longitude, latitude) values. Getter methods enable the user to get the longitude or latitude of any given coordinate, and a static getter method enables one to get the distance between any two arbitrary coordinates, based on Euclidean distance.

The **Sensor** class models the state of the sensors. There are three instance methods that contain information on the sensor’s reading, battery level, and location. The location is encoded as a What3Words string. What3Words is a company that has segmented the Earth into an array of 3mx3m squares, each of which is codified by a unique address denoted by three words. Each sensor exists at the center of one such square. Since the state of a sensor is fixed on any given day, there are no methods that can alter its state. However, there are getter methods that correspond to each instance variable that fetch its data for usage elsewhere.

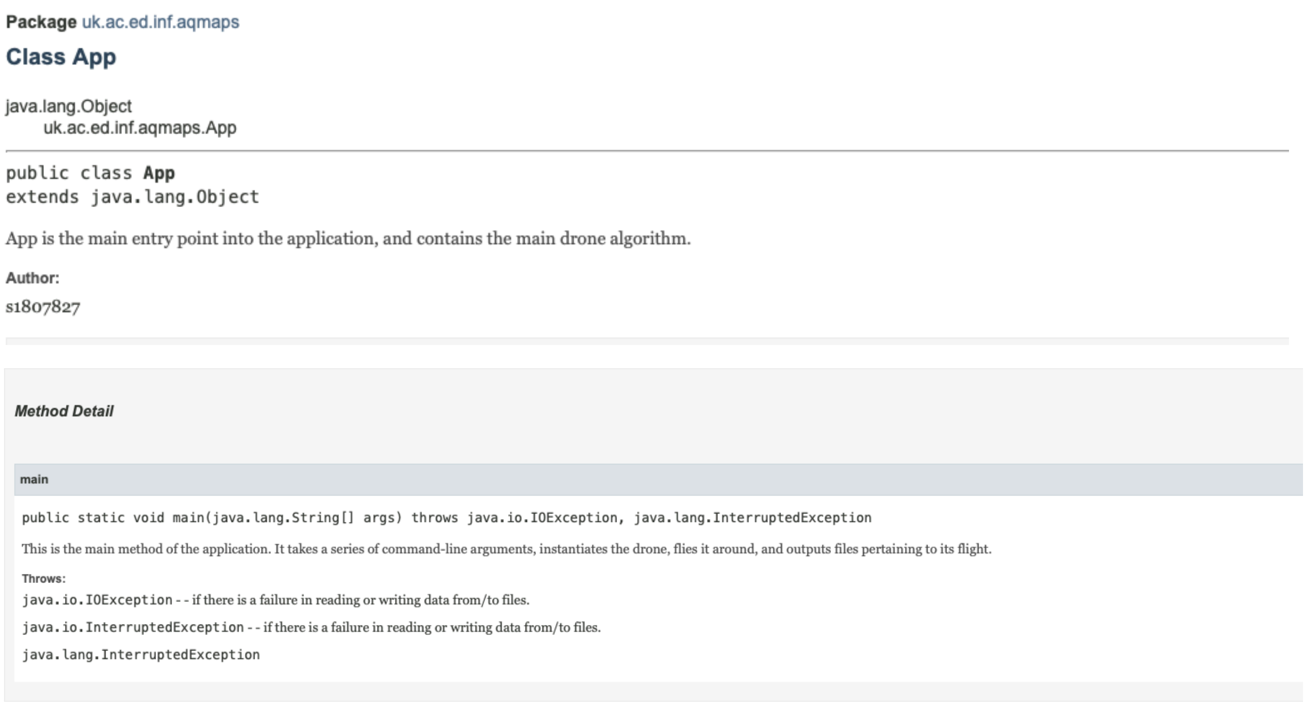
The **ReadWrite** class is a wholly static class, containing a bevy of methods that perform useful parsing, reading, and writing operations. Chief among these is the **toCoordinates()** method, which converts What3Words addresses to **Coordinate** objects. Furthermore, it contains methods which enable the drone to take readings from the sensor - by converting the reading to a hexadecimal color and a marker symbol for displaying on a map. There are also methods to output the drone's moves to a text file and to generate a map of the locations of the sensors along with the drone's flightpath, among others. The **ReadWrite** class underpins all of the other classes, and enables crucial conversions between data types.

The **W3W** class is designed to act as a container for the data for each What3Words address. The data exists in a JSON format on the web server, and this class' purpose is to deserialize the data into a format that can be used elsewhere. As a result of deserialization requirements, it contains a duplicate "Coordinates" subclass that holds the coordinates of the center of the square, as well as its northeast and southwest corners. However, this subclass is not used elsewhere. There are getter methods here as well that fetch data for usage elsewhere. The main usage of the **W3W** class is in a method of the **ReadWrite** class, which converts What3Words addresses to **Coordinate** objects.

Class Documentation

This documentation is a high-level summary of the instance variables and methods in each class. For a more in-depth look at everything, consider seeing the "doc" subfolder of the code submission.

1. App



Package uk.ac.ed.inf.aqmaps

Class App

java.lang.Object
    uk.ac.ed.inf.aqmaps.App

```
public class App
extends java.lang.Object
```

App is the main entry point into the application, and contains the main drone algorithm.

Author:
s1807827

Method Detail

main

```
public static void main(java.lang.String[] args) throws java.io.IOException, java.lang.InterruptedException
```

This is the main method of the application. It takes a series of command-line arguments, instantiates the drone, flies it around, and outputs files pertaining to its flight.

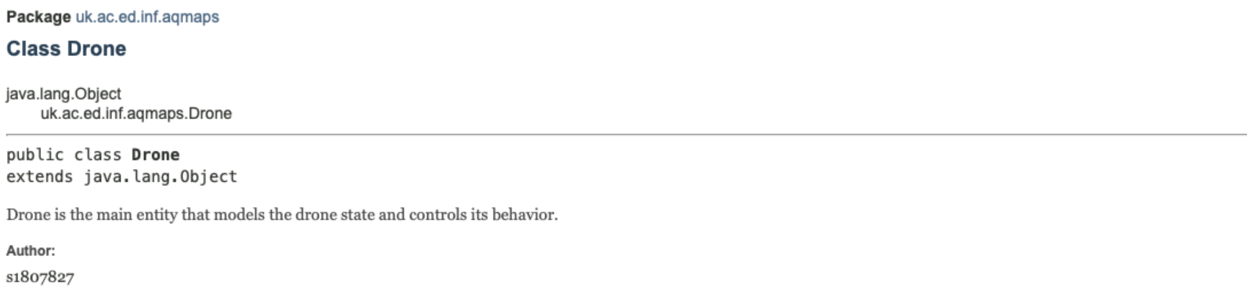
Throws:

`java.io.IOException` - - if there is a failure in reading or writing data from/to files.

`java.io.InterruptedException` - - if there is a failure in reading or writing data from/to files.

`java.lang.InterruptedException`

2. Drone



Package uk.ac.ed.inf.aqmaps

Class Drone

java.lang.Object
    uk.ac.ed.inf.aqmaps.Drone

```
public class Drone
extends java.lang.Object
```

Drone is the main entity that models the drone state and controls its behavior.

Author:
s1807827

Field Summary

Fields

Modifier and Type	Field	Description
`private java.net.http.HttpClient`	`client`	The HttpClient the drone uses to connect to the web server.
`int`	`heading`	The direction the drone is facing in degrees, where 0 is due East on a compass rose, and 180 is due West.
`java.util.ArrayList<com.mapbox.geojson.Feature>`	`markers`	A list of GeoJSON markers the drone has placed for each sensor location that it has visited and taken a reading from.
`int`	`moveCount`	The number of moves the drone has made.
`java.util.ArrayList<MoveData>`	`moves`	A list of MoveData objects that contain details of the drone's move history.
`private int`	`port`	The port on which the drone connects to the web server.
`Coordinate`	`position`	The current position of a drone, as specified by its longitude and latitude.
`java.util.ArrayList<Sensor>`	`sensorList`	A list of the sensors the drone must visit.
`java.util.ArrayList<Sensor>`	`visitedSensors`	A list of the sensors the drone has visited and taken a reading from.

Constructor Summary

Constructors

Constructor	Description
`Drone(Coordinate position, java.net.http.HttpClient client, int port, java.util.ArrayList<MoveData> moves, java.util.ArrayList<Sensor> sensorList, java.util.ArrayList<Sensor> visitedSensors, java.util.ArrayList<com.mapbox.geojson.Feature> markers)`	

Method Detail

takeReading

```
public void takeReading(Sensor sensor)
```

This method enables the drone to take a reading once it is sufficiently close (within 0.0002 degrees) to a sensor, and adds the relevant marker for that sensor to the drone's list of markers.

Parameters:
`sensor` - - the sensor object which it needs to read

move

```
public void move(int heading)
```

This method enables the drone to move 0.0003 degrees in the given direction.

Parameters:
`heading` - - the direction it needs to move in.

inNoFlyZone

```
public boolean inNoFlyZone(java.util.ArrayList<com.mapbox.geojson.Polygon> noFlyZones)
```

This method determines whether the drone is about to enter a No-Fly Zone in the next move.

Parameters:
`noFlyZones` - - a list of No-Fly Zones, as GeoJSON Polygons.

Returns:
Whether or not the drone is about to enter a No-Fly Zone in the next move (true or false value).

calculateHeading

```
public int calculateHeading(Coordinate destination)
```

This method determines the direction to travel in given a set of destination coordinates.

Parameters:
`destination` - - The coordinates the drone needs to move towards.

Returns:
An int which is the direction the drone has to move in to get to the destination point.

calculateNextPositions

```
private java.util.ArrayList<com.mapbox.geojson.Point> calculateNextPositions()
```

This method splits a move of 0.0003 degrees into 100 small points, and is used by the inNoFlyZone() method to calculate whether any of the small points are in a No-Fly Zone.

Returns:
An ArrayList of 100 points between the current position, and the position that is 0.0003 degrees away from the direction the drone is facing in.

setPosition

```
public void setPosition(Coordinate position)
```

This method sets the new position of the drone.

Parameters:
`position` - - The new position of the drone.

getPosition

```
public Coordinate getPosition()
```

This method returns the current position of the drone.

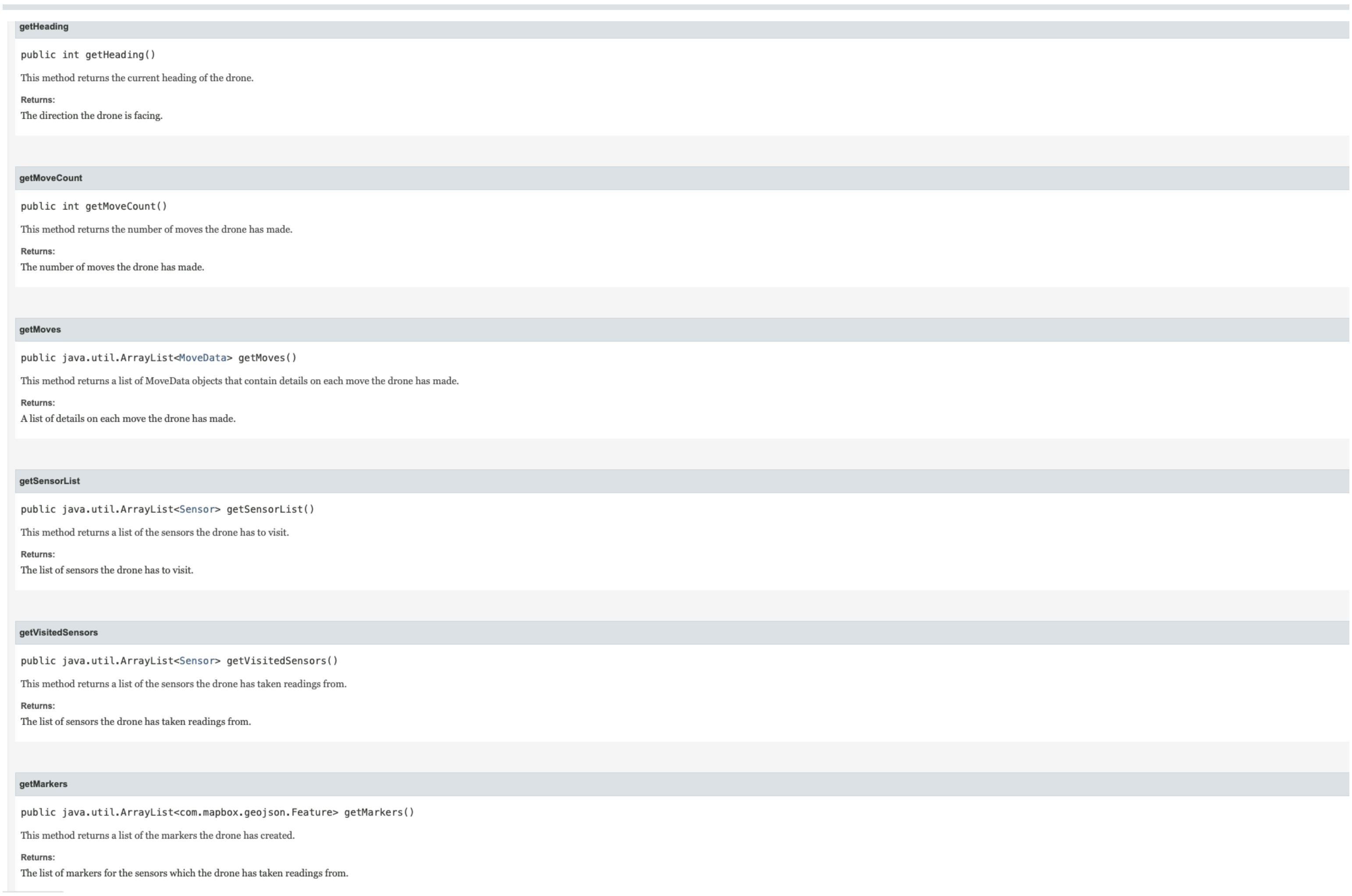
Returns:
The position of the drone.

setHeading

```
public void setHeading(int heading)
```

This method sets the direction the drone is facing in to the given heading.

Parameters:
`heading` - - The direction the drone needs to face.



getHeading

```
public int getHeading()
```

This method returns the current heading of the drone.

Returns:
The direction the drone is facing.

getMoveCount

```
public int getMoveCount()
```

This method returns the number of moves the drone has made.

Returns:
The number of moves the drone has made.

getMoves

```
public java.util.ArrayList<MoveData> getMoves()
```

This method returns a list of MoveData objects that contain details on each move the drone has made.

Returns:
A list of details on each move the drone has made.

getSensorList

```
public java.util.ArrayList<Sensor> getSensorList()
```

This method returns a list of the sensors the drone has to visit.

Returns:
The list of sensors the drone has to visit.

getVisitedSensors

```
public java.util.ArrayList<Sensor> getVisitedSensors()
```

This method returns a list of the sensors the drone has taken readings from.

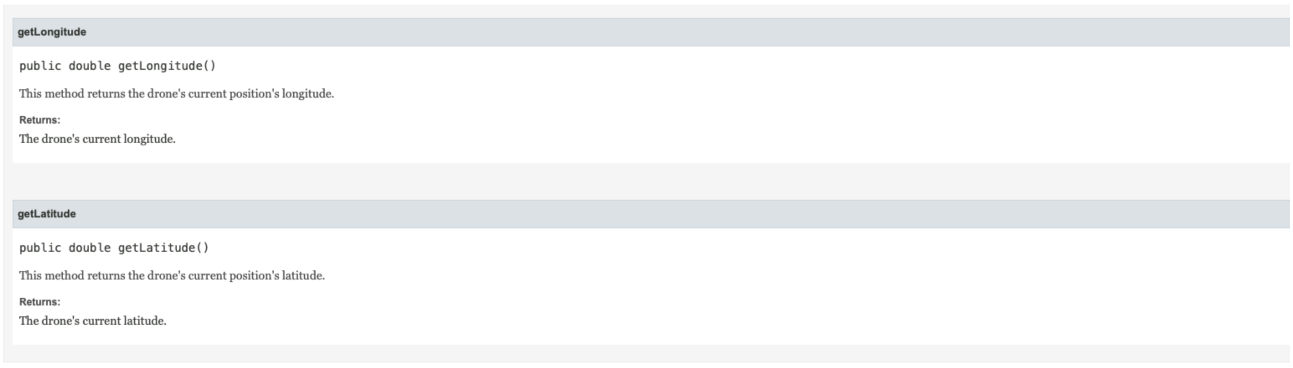
Returns:
The list of sensors the drone has taken readings from.

getMarkers

```
public java.util.ArrayList<com.mapbox.geojson.Feature> getMarkers()
```

This method returns a list of the markers the drone has created.

Returns:
The list of markers for the sensors which the drone has taken readings from.



getLongitude

```
public double getLongitude()
```

This method returns the drone's current position's longitude.

Returns:
The drone's current longitude.

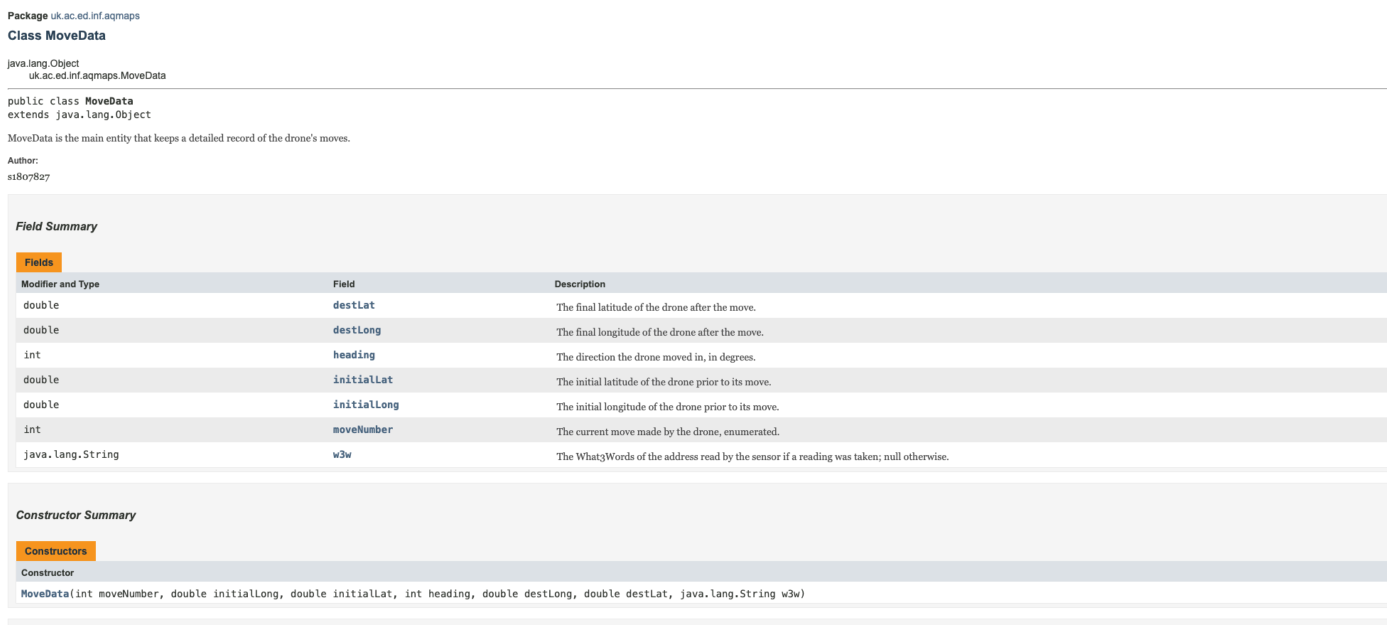
getLatitude

```
public double getLatitude()
```

This method returns the drone's current position's latitude.

Returns:
The drone's current latitude.

3. MoveData



Package uk.ac.ed.inf.aqmaps

Class MoveData

java.lang.Object
    uk.ac.ed.inf.aqmaps.MoveData

```
public class MoveData
extends java.lang.Object
```

MoveData is the main entity that keeps a detailed record of the drone's moves.

Author:
s1807827

Field Summary

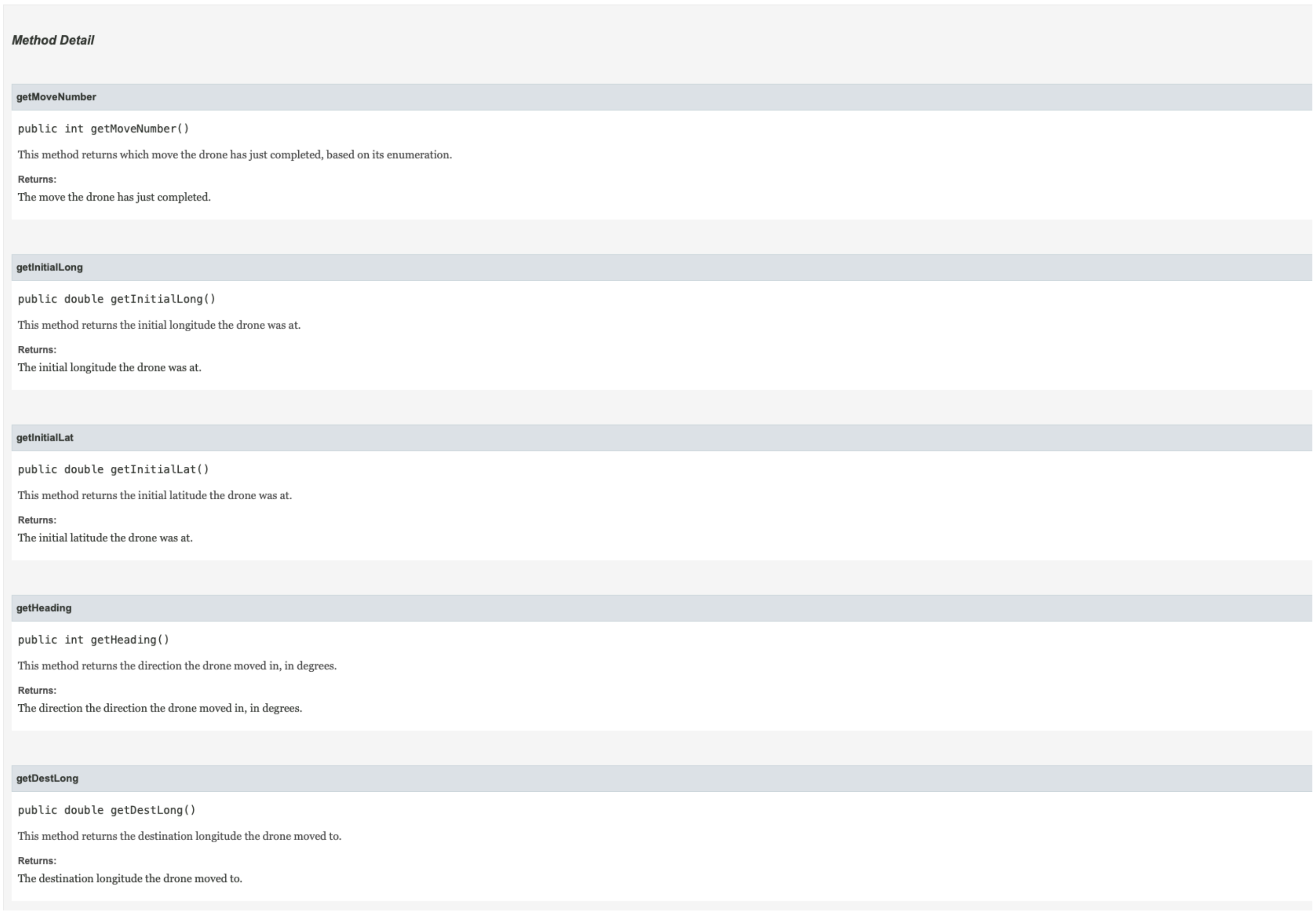
Fields

Modifier and Type	Field	Description
double	destLat	The final latitude of the drone after the move.
double	destLong	The final longitude of the drone after the move.
int	heading	The direction the drone moved in, in degrees.
double	initialLat	The initial latitude of the drone prior to its move.
double	initialLong	The initial longitude of the drone prior to its move.
int	moveNumber	The current move made by the drone, enumerated.
java.lang.String	w3w	The What3Words of the address read by the sensor if a reading was taken; null otherwise.

Constructor Summary

Constructors

Constructor
MoveData(int moveNumber, double initialLong, double initialLat, int heading, double destLong, double destLat, java.lang.String w3w)



Method Detail

getMoveNumber

`public int getMoveNumber()`

This method returns which move the drone has just completed, based on its enumeration.

Returns:
The move the drone has just completed.

getInitialLong

`public double getInitialLong()`

This method returns the initial longitude the drone was at.

Returns:
The initial longitude the drone was at.

getInitialLat

`public double getInitialLat()`

This method returns the initial latitude the drone was at.

Returns:
The initial latitude the drone was at.

getHeading

`public int getHeading()`

This method returns the direction the drone moved in, in degrees.

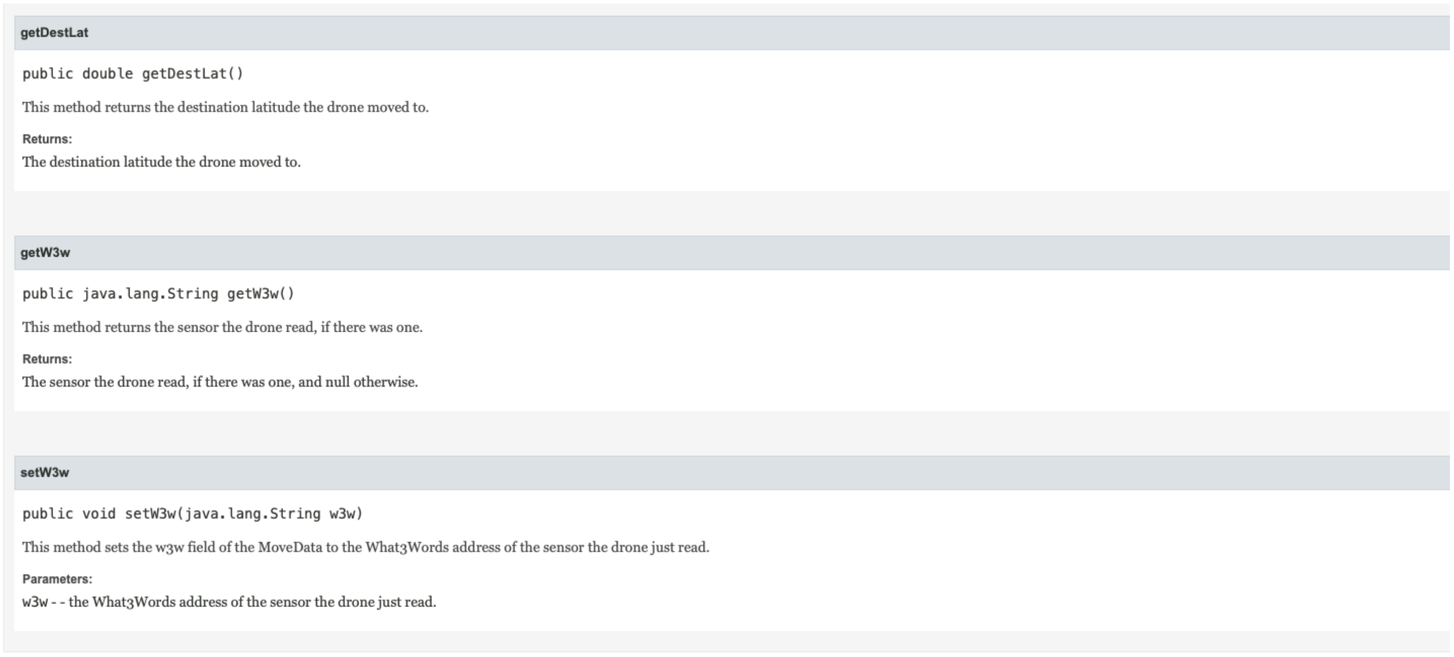
Returns:
The direction the direction the drone moved in, in degrees.

getDestLong

`public double getDestLong()`

This method returns the destination longitude the drone moved to.

Returns:
The destination longitude the drone moved to.



getDestLat

`public double getDestLat()`

This method returns the destination latitude the drone moved to.

Returns:
The destination latitude the drone moved to.

getW3w

`public java.lang.String getW3w()`

This method returns the sensor the drone read, if there was one.

Returns:
The sensor the drone read, if there was one, and null otherwise.

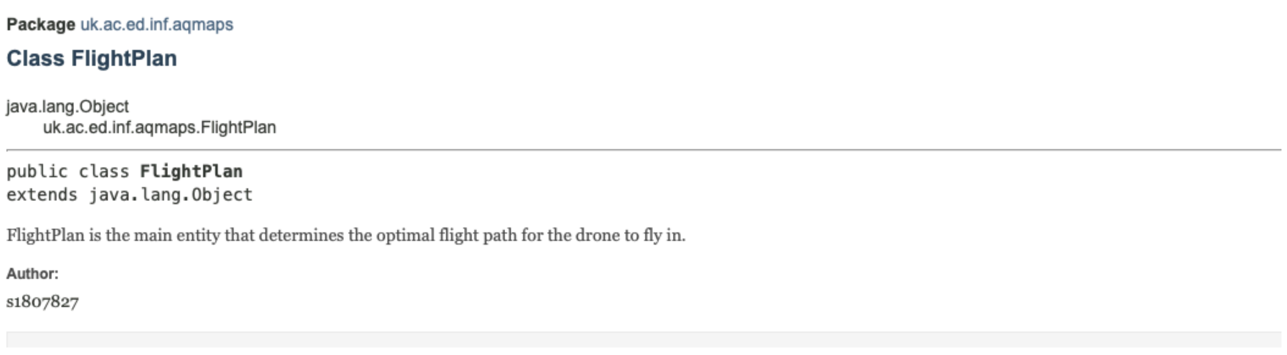
setW3w

`public void setW3w(java.lang.String w3w)`

This method sets the w3w field of the MoveData to the What3Words address of the sensor the drone just read.

Parameters:
`w3w` - - the What3Words address of the sensor the drone just read.

4. FlightPlan



Package uk.ac.ed.inf.aqmaps

Class FlightPlan

java.lang.Object
    uk.ac.ed.inf.aqmaps.FlightPlan

```
public class FlightPlan
extends java.lang.Object
```

FlightPlan is the main entity that determines the optimal flight path for the drone to fly in.

Author:
s1807827

Field Summary

Fields

Modifier and Type	Field	Description
`private java.net.http.HttpClient`	`client`	The HttpClient the drone uses to connect to the web server.
`private double[][]`	`dists`	A matrix containing the distances between any pair of sensors in the geographical area.
`int`	`numberOfSensors`	The number of sensors the drone needs to visit.
`int[]`	`perm`	An array of the indices of the sensors which the drone must visit; the order is the permutation for the drone to visit.
`private int`	`port`	The port on which the drone connects to the web server.
`private java.util.ArrayList<Sensor>`	`sensors`	A list of the sensors the drone must visit.
`private Coordinate`	`startingPosition`	The drone's starting position, expressed as a Coordinate.

Constructor Summary

Constructors

Constructor	Description
`FlightPlan(java.util.ArrayList<Sensor> sensors, java.net.http.HttpClient client, int port, Coordinate startingPosition)`	Creates the initial flight plan, computes the distances between all the sensors, sets the default permutation to the identity permutation.

Method Detail

tourValue

```
public double tourValue()
```

This method returns the total cost (distance) of the path determined by the permutation of the sensors.

Returns:
The total cost (distance) of the path determined by the permutation of the sensors.

Greedy

```
private void Greedy()
```

This method determines the permutation of sensors to given in by iterating through the sensors and picking the next sensor to visit to be the closest sensor to that sensor. It is a greedy best-first search approach.

trySwap

```
private boolean trySwap(int i)
```

This method determines if switching the order of two sensors in the permutation at indices i and i+1 will result in a shorter path.

Parameters:
`i` - - the index of the sensor to switch with its mirror index value.

Returns:
A boolean stating if swapping the sensors resulted in a shorter path - if false, it reverts to the original permutation. If true, it commits to the new permutation.

swapHeuristic

```
private void swapHeuristic()
```

This method swaps a sensor with the sensor at its mirror index value for each possible sensor pair such that the resultant tour value is minimized.

tryReverse

```
private boolean tryReverse(int i, int j)
```

This method reverses the segment of the flightpath between indices i and j, and commits to the reversal if it improves the tour value.

Parameters:
`i` - - the starting index of the segment

`j` - - the ending index of the segment

Returns:
A boolean stating if reversing the segment resulted in a shorter path - if false, it reverts to the original permutation. If true, it commits the new permutation.

TwoOptHeuristic

```
private void TwoOptHeuristic()
```

This method reverses every possible segment (i, j) for all i, j < the number of sensors, and commits the new permutation.

tuneFlightPlan

```
public void tuneFlightPlan()
```

This method finds the optimal flightpath. It first conducts a greedy best-first search, then performs a swap heuristic to tune it, and then finally tunes that even further by performing the Two-Opt heuristic.

getPerm

```
public int[] getPerm()
```

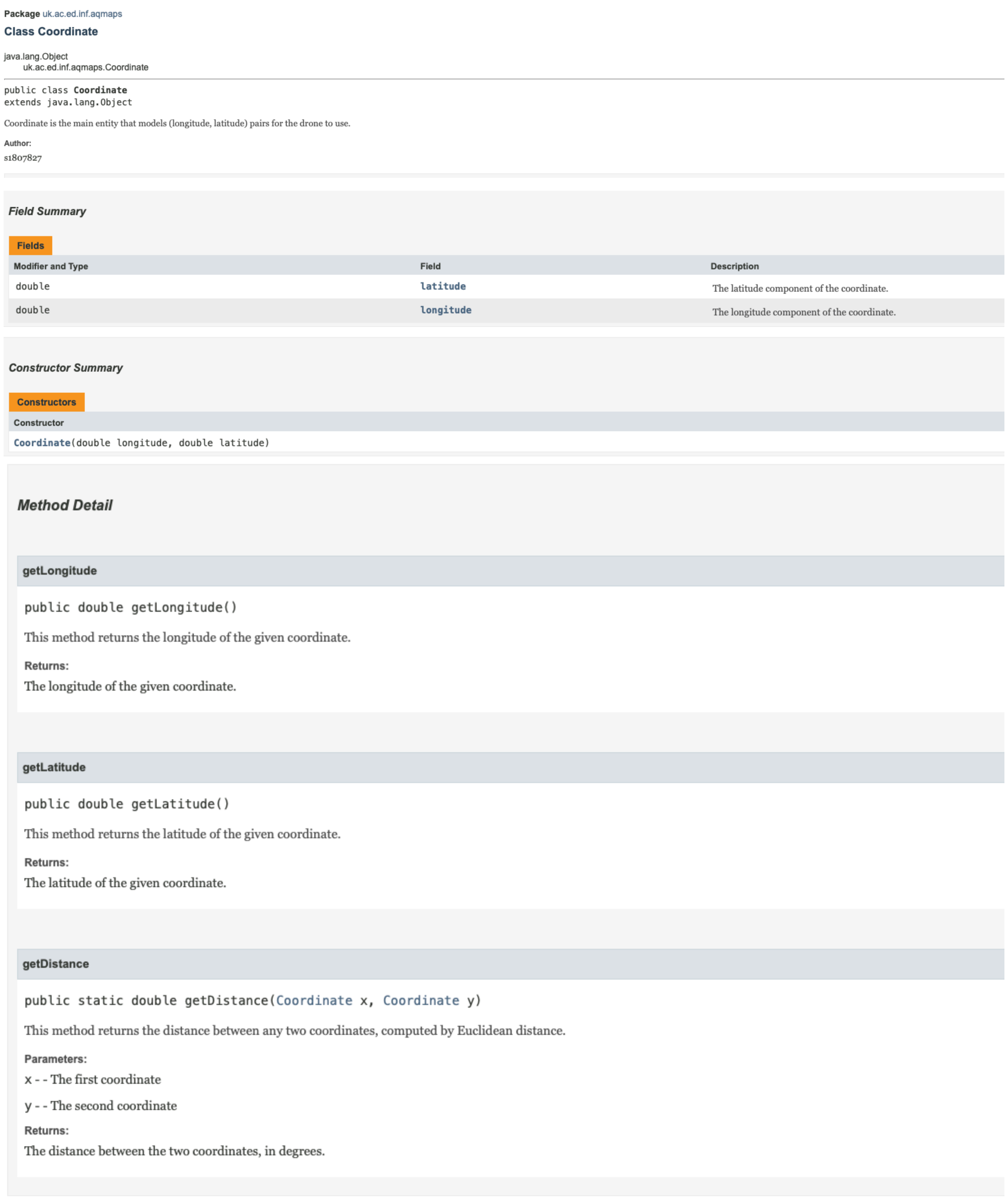
This method returns the permutation of the sensors which the drone should visit.

getNumberOfSensors

```
public int getNumberOfSensors()
```

This method returns the number of sensors the drone needs to visit.

5. Coordinate



Package uk.ac.ed.inf.aqmaps

Class Coordinate

java.lang.Object
    uk.ac.ed.inf.aqmaps.Coordinate

```
public class Coordinate
extends java.lang.Object
```

Coordinate is the main entity that models (longitude, latitude) pairs for the drone to use.

Author:
s1807827

Field Summary

Fields

Modifier and Type	Field	Description
double	latitude	The latitude component of the coordinate.
double	longitude	The longitude component of the coordinate.

Constructor Summary

Constructors

Constructor
`Coordinate(double longitude, double latitude)`

Method Detail

getLongitude

```
public double getLongitude()
```

This method returns the longitude of the given coordinate.

Returns:
The longitude of the given coordinate.

getLatitude

```
public double getLatitude()
```

This method returns the latitude of the given coordinate.

Returns:
The latitude of the given coordinate.

getDistance

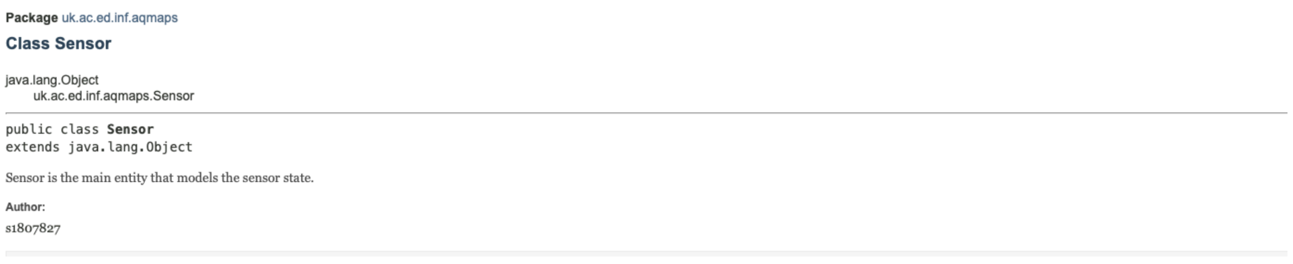
```
public static double getDistance(Coordinate x, Coordinate y)
```

This method returns the distance between any two coordinates, computed by Euclidean distance.

Parameters:
x - - The first coordinate
y - - The second coordinate

Returns:
The distance between the two coordinates, in degrees.

6. Sensor



Package uk.ac.ed.inf.aqmaps

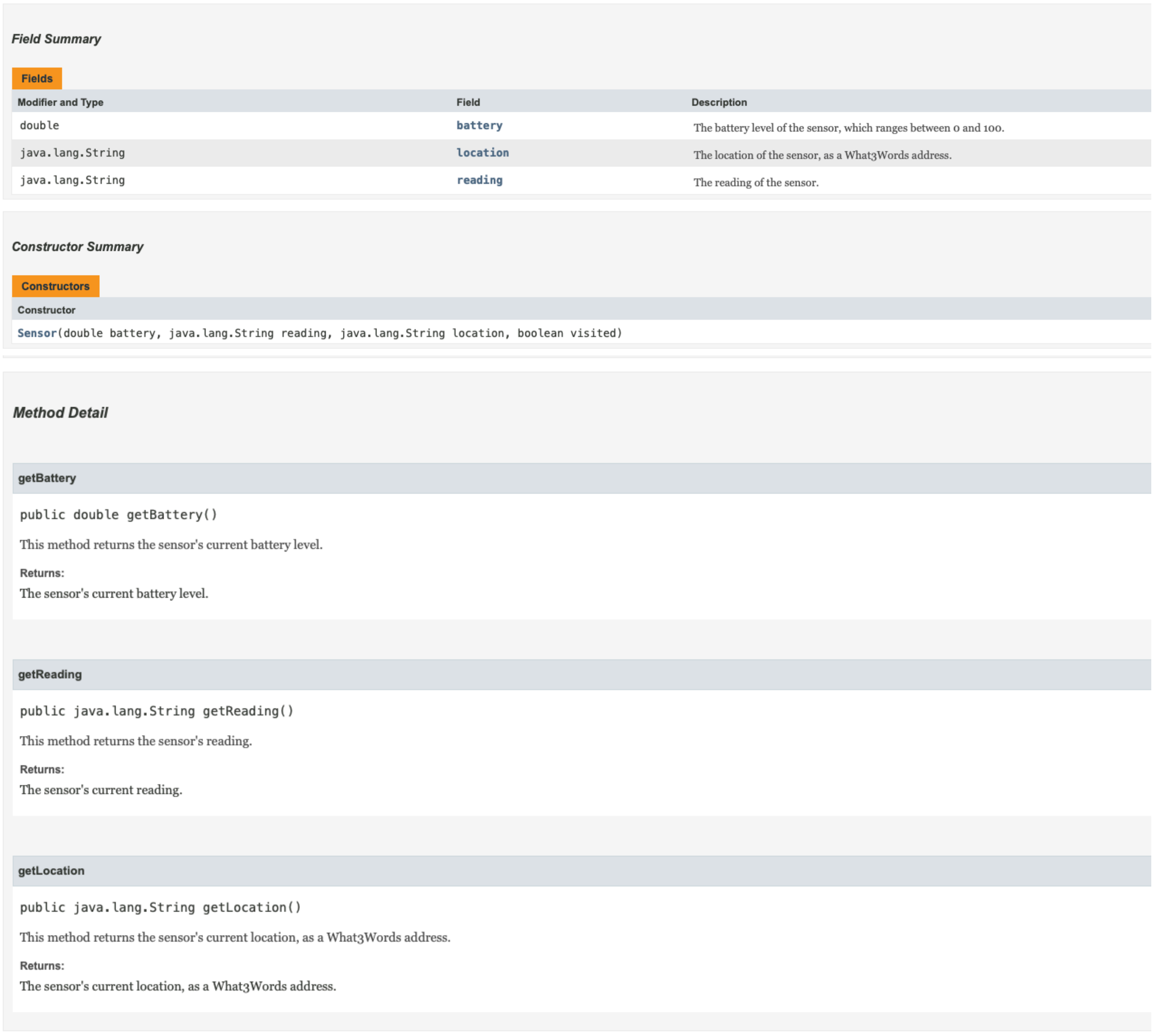
Class Sensor

java.lang.Object
    uk.ac.ed.inf.aqmaps.Sensor

```
public class Sensor
extends java.lang.Object
```

Sensor is the main entity that models the sensor state.

Author:
s1807827



Field Summary

Fields

Modifier and Type	Field	Description
`double`	`battery`	The battery level of the sensor, which ranges between 0 and 100.
`java.lang.String`	`location`	The location of the sensor, as a What3Words address.
`java.lang.String`	`reading`	The reading of the sensor.

Constructor Summary

Constructors

Constructor
`Sensor(double battery, java.lang.String reading, java.lang.String location, boolean visited)`

Method Detail

getBattery

```
public double getBattery()
```

This method returns the sensor's current battery level.

Returns:
The sensor's current battery level.

getReading

```
public java.lang.String getReading()
```

This method returns the sensor's reading.

Returns:
The sensor's current reading.

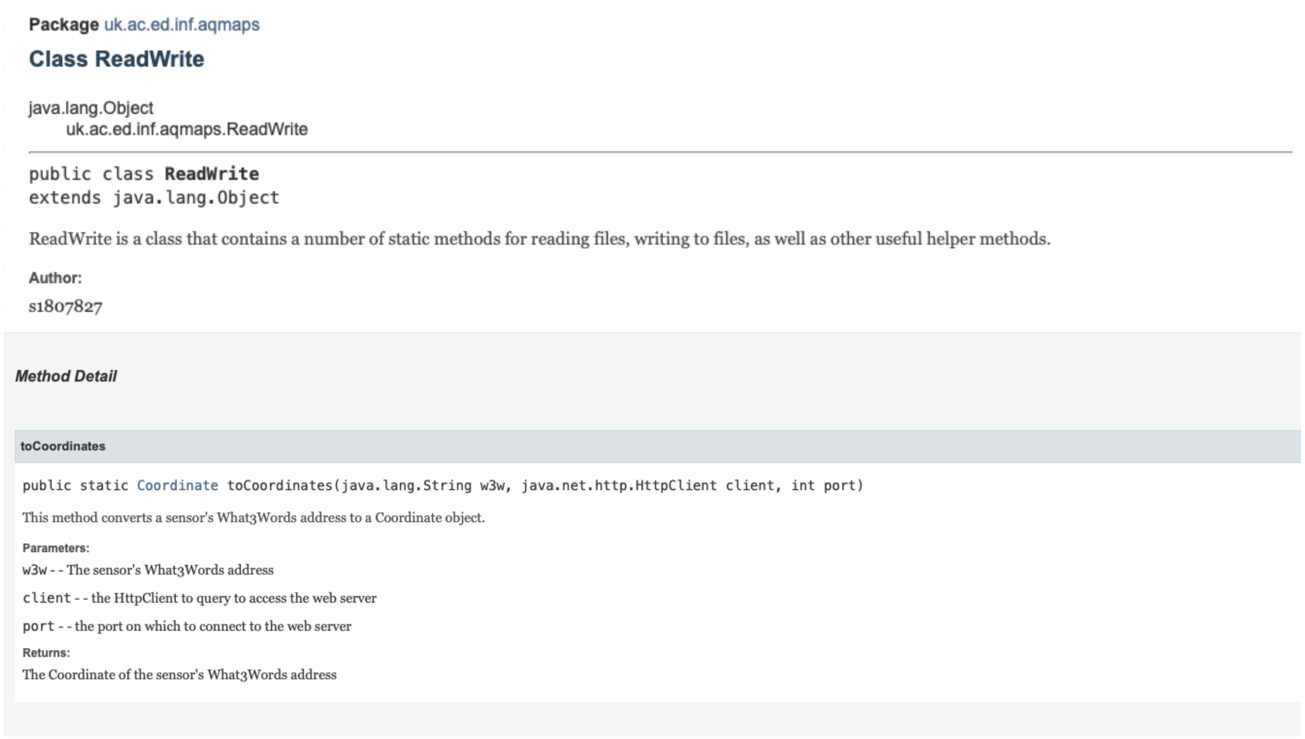
getLocation

```
public java.lang.String getLocation()
```

This method returns the sensor's current location, as a What3Words address.

Returns:
The sensor's current location, as a What3Words address.

7. ReadWrite



Package uk.ac.ed.inf.aqmaps

Class ReadWrite

java.lang.Object
    uk.ac.ed.inf.aqmaps.ReadWrite

```
public class ReadWrite
extends java.lang.Object
```

ReadWrite is a class that contains a number of static methods for reading files, writing to files, as well as other useful helper methods.

Author:
s1807827

Method Detail

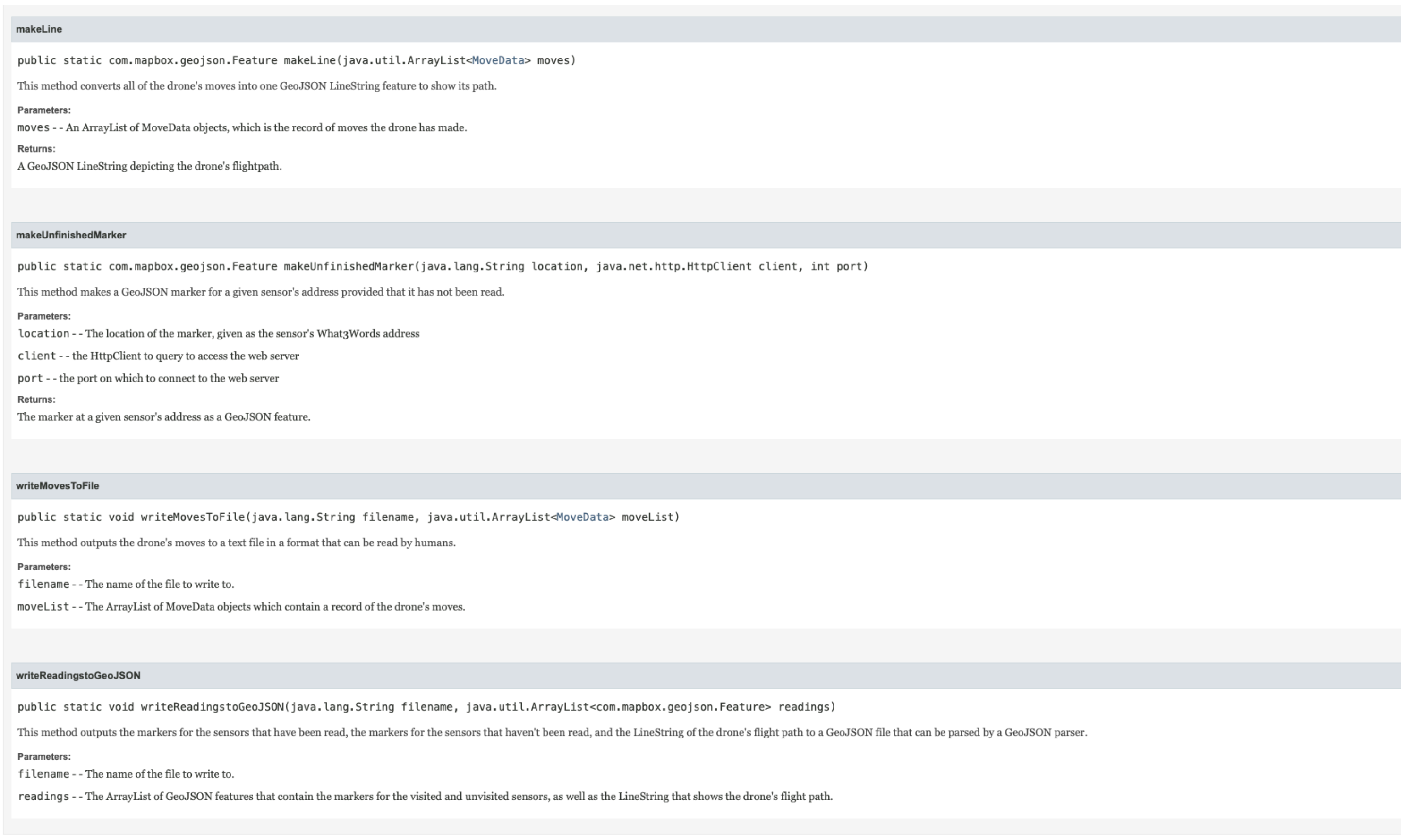
toCoordinates

```
public static Coordinate toCoordinates(java.lang.String w3w, java.net.http.HttpClient client, int port)
```

This method converts a sensor's What3Words address to a Coordinate object.

Parameters:
`w3w` - - The sensor's What3Words address
`client` - - the HttpClient to query to access the web server
`port` - - the port on which to connect to the web server

Returns:
The Coordinate of the sensor's What3Words address



makeLine

`public static com.mapbox.geojson.Feature makeLine(java.util.ArrayList<MoveData> moves)`

This method converts all of the drone's moves into one GeoJSON LineString feature to show its path.

Parameters:
`moves` - - An ArrayList of MoveData objects, which is the record of moves the drone has made.

Returns:
A GeoJSON LineString depicting the drone's flightpath.

makeUnfinishedMarker

`public static com.mapbox.geojson.Feature makeUnfinishedMarker(java.lang.String location, java.net.http.HttpClient client, int port)`

This method makes a GeoJSON marker for a given sensor's address provided that it has not been read.

Parameters:
`location` - - The location of the marker, given as the sensor's What3Words address
`client` - - the HttpClient to query to access the web server
`port` - - the port on which to connect to the web server

Returns:
The marker at a given sensor's address as a GeoJSON feature.

writeMovesToFile

`public static void writeMovesToFile(java.lang.String filename, java.util.ArrayList<MoveData> moveList)`

This method outputs the drone's moves to a text file in a format that can be read by humans.

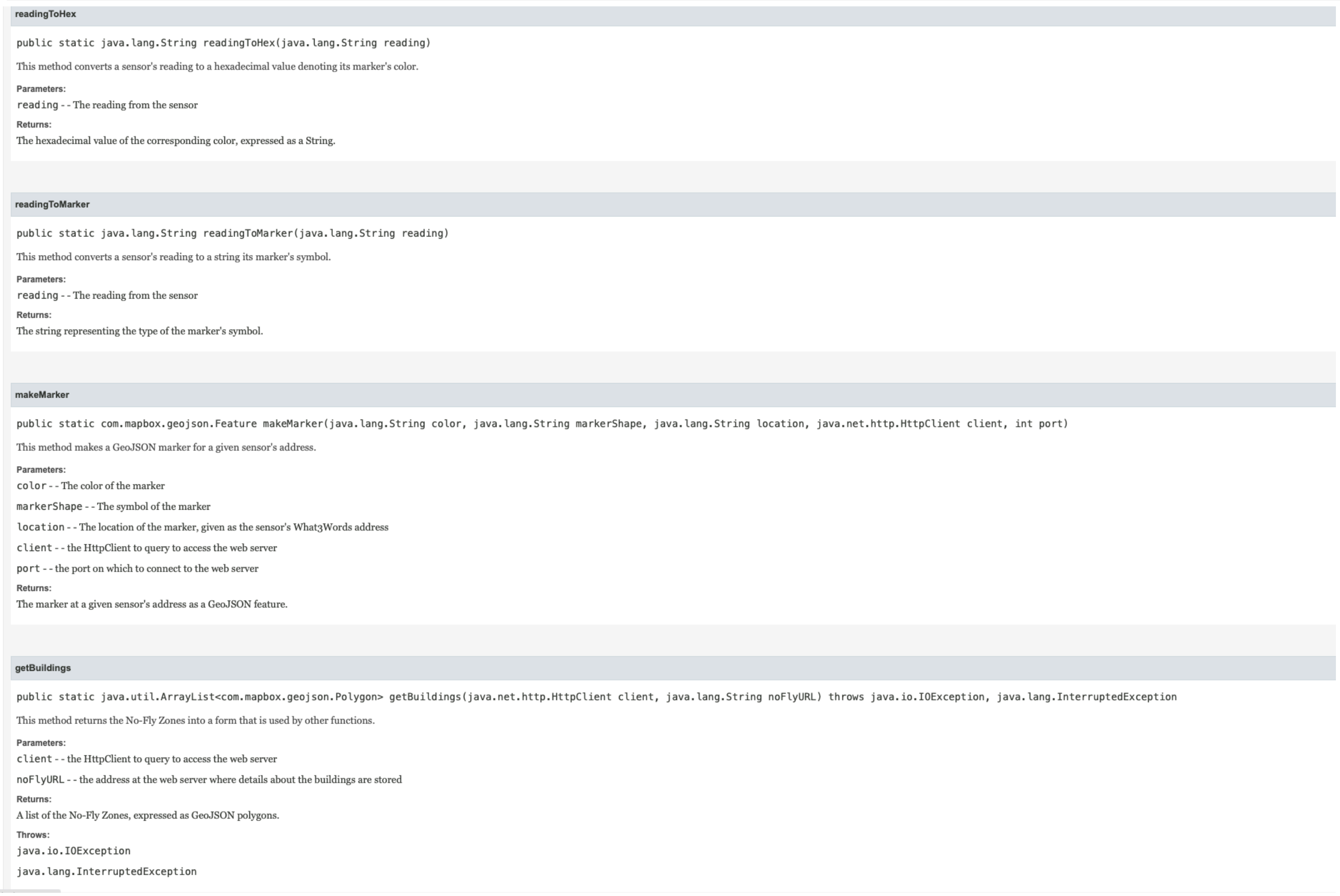
Parameters:
`filename` - - The name of the file to write to.
`moveList` - - The ArrayList of MoveData objects which contain a record of the drone's moves.

writeReadingstoGeoJSON

`public static void writeReadingstoGeoJSON(java.lang.String filename, java.util.ArrayList<com.mapbox.geojson.Feature> readings)`

This method outputs the markers for the sensors that have been read, the markers for the sensors that haven't been read, and the LineString of the drone's flight path to a GeoJSON file that can be parsed by a GeoJSON parser.

Parameters:
`filename` - - The name of the file to write to.
`readings` - - The ArrayList of GeoJSON features that contain the markers for the visited and unvisited sensors, as well as the LineString that shows the drone's flight path.



readingToHex

`public static java.lang.String readingToHex(java.lang.String reading)`

This method converts a sensor's reading to a hexadecimal value denoting its marker's color.

Parameters:
`reading` - - The reading from the sensor

Returns:
The hexadecimal value of the corresponding color, expressed as a String.

readingToMarker

`public static java.lang.String readingToMarker(java.lang.String reading)`

This method converts a sensor's reading to a string its marker's symbol.

Parameters:
`reading` - - The reading from the sensor

Returns:
The string representing the type of the marker's symbol.

makeMarker

`public static com.mapbox.geojson.Feature makeMarker(java.lang.String color, java.lang.String markerShape, java.lang.String location, java.net.http.HttpClient client, int port)`

This method makes a GeoJSON marker for a given sensor's address.

Parameters:
`color` - - The color of the marker
`markerShape` - - The symbol of the marker
`location` - - The location of the marker, given as the sensor's What3Words address
`client` - - the HttpClient to query to access the web server
`port` - - the port on which to connect to the web server

Returns:
The marker at a given sensor's address as a GeoJSON feature.

getBuildings

`public static java.util.ArrayList<com.mapbox.geojson.Polygon> getBuildings(java.net.http.HttpClient client, java.lang.String noFlyURL) throws java.io.IOException, java.lang.InterruptedException`

This method returns the No-Fly Zones into a form that is used by other functions.

Parameters:
`client` - - the HttpClient to query to access the web server
`noFlyURL` - - the address at the web server where details about the buildings are stored

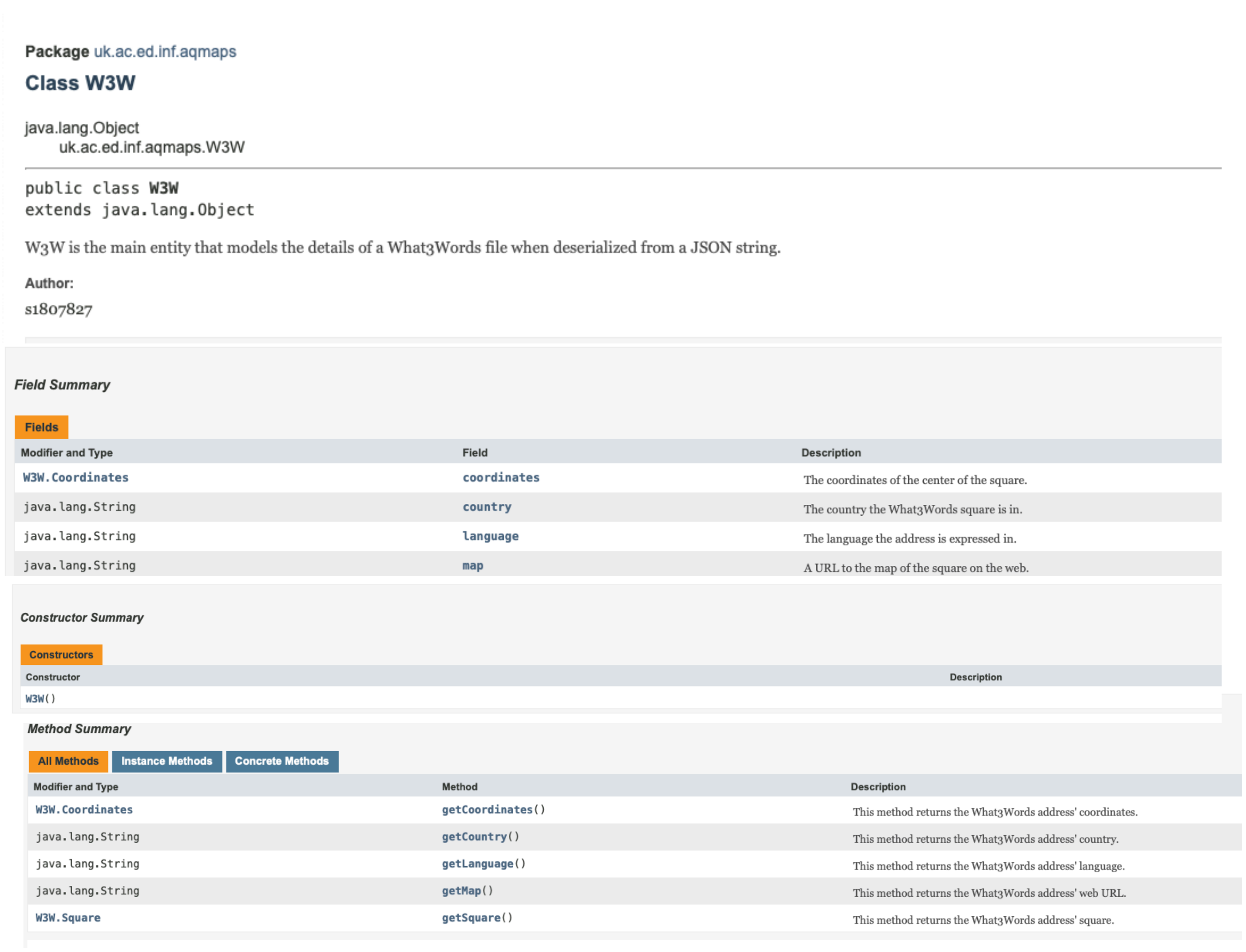
Returns:
A list of the No-Fly Zones, expressed as GeoJSON polygons.

Throws:
`java.io.IOException`
`java.lang.InterruptedException`

8. W3W

The **W3W** class is composed of various fields as well as nested static classes. Documentation for both is given here.

8.1 W3W Main Class



Package uk.ac.ed.inf.aqmaps

Class W3W

java.lang.Object
    uk.ac.ed.inf.aqmaps.W3W

```
public class W3W
extends java.lang.Object
```

W3W is the main entity that models the details of a What3Words file when deserialized from a JSON string.

Author:
s1807827

Field Summary

Fields

Modifier and Type	Field	Description
`W3W.Coordinates`	`coordinates`	The coordinates of the center of the square.
`java.lang.String`	`country`	The country the What3Words square is in.
`java.lang.String`	`language`	The language the address is expressed in.
`java.lang.String`	`map`	A URL to the map of the square on the web.

Constructor Summary

Constructors

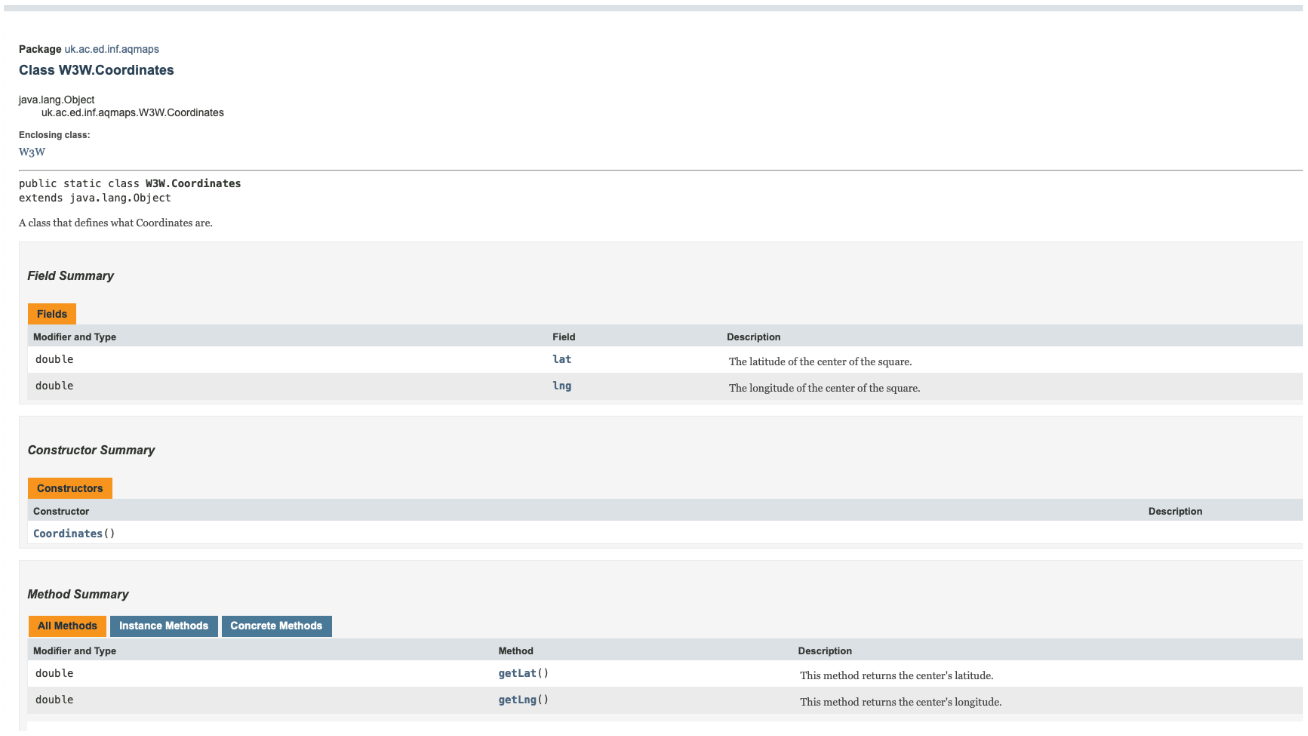
Constructor	Description
`W3W()`	

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`W3W.Coordinates`	`getCoordinates()`	This method returns the What3Words address' coordinates.
`java.lang.String`	`getCountry()`	This method returns the What3Words address' country.
`java.lang.String`	`getLanguage()`	This method returns the What3Words address' language.
`java.lang.String`	`getMap()`	This method returns the What3Words address' web URL.
`W3W.Square`	`getSquare()`	This method returns the What3Words address' square.

8.2 W3W.Coordinates



Package uk.ac.ed.inf.aqmaps

Class W3W.Coordinates

java.lang.Object
    uk.ac.ed.inf.aqmaps.W3W.Coordinates

Enclosing class:
W3W

```
public static class W3W.Coordinates
extends java.lang.Object
```

A class that defines what Coordinates are.

Field Summary

Fields

Modifier and Type	Field	Description
`double`	`lat`	The latitude of the center of the square.
`double`	`lng`	The longitude of the center of the square.

Constructor Summary

Constructors

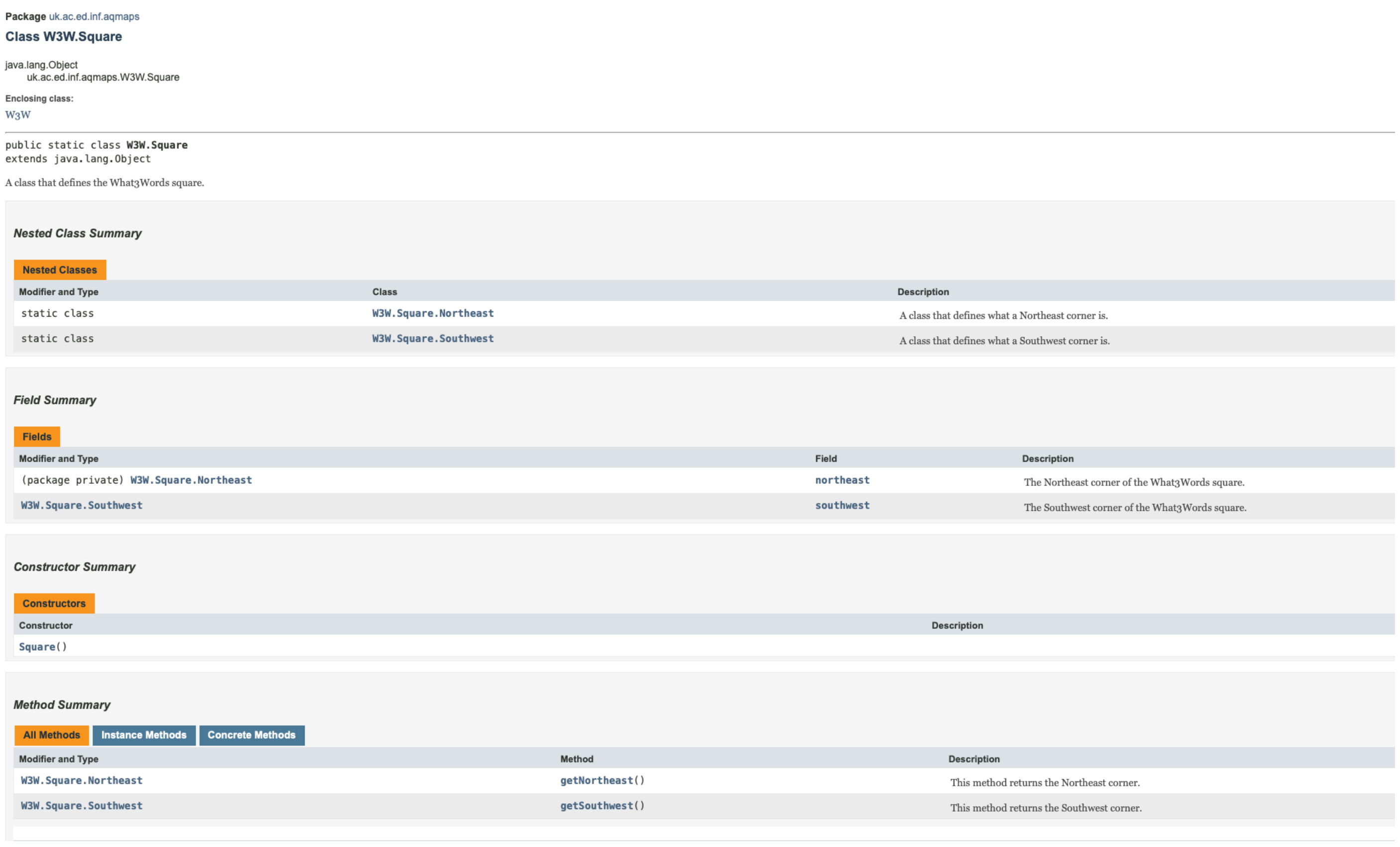
Constructor	Description
`Coordinates()`	

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`double`	`getLat()`	This method returns the center's latitude.
`double`	`getLng()`	This method returns the center's longitude.

8.2 W3W.Square



Package uk.ac.ed.inf.aqmaps

Class W3W.Square

java.lang.Object
    uk.ac.ed.inf.aqmaps.W3W.Square

Enclosing class:
W3W

```
public static class W3W.Square
extends java.lang.Object
```

A class that defines the What3Words square.

Nested Class Summary

Nested Classes

Modifier and Type	Class	Description
`static class`	`W3W.Square.Northeast`	A class that defines what a Northeast corner is.
`static class`	`W3W.Square.Southwest`	A class that defines what a Southwest corner is.

Field Summary

Fields

Modifier and Type	Field	Description
`(package private) W3W.Square.Northeast`	`northeast`	The Northeast corner of the What3Words square.
`W3W.Square.Southwest`	`southwest`	The Southwest corner of the What3Words square.

Constructor Summary

Constructors

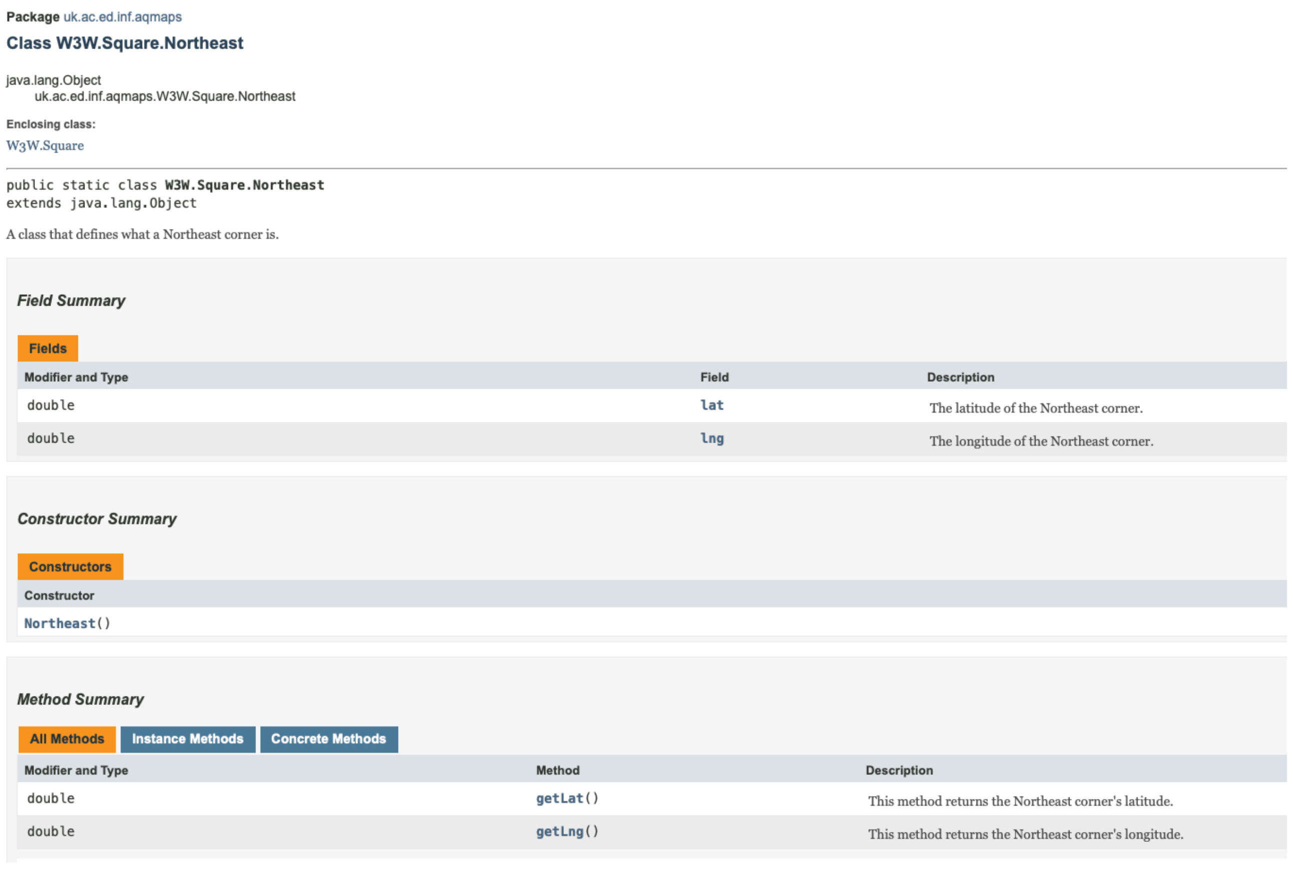
Constructor	Description
`Square()`	

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method	Description
`W3W.Square.Northeast`	`getNortheast()`	This method returns the Northeast corner.
`W3W.Square.Southwest`	`getSouthwest()`	This method returns the Southwest corner.

8.2 W3W.Square.Northeast



Package uk.ac.ed.inf.aqmaps

Class W3W.Square.Northeast

java.lang.Object
    uk.ac.ed.inf.aqmaps.W3W.Square.Northeast

Enclosing class:
W3W.Square

```
public static class W3W.Square.Northeast
extends java.lang.Object
```

A class that defines what a Northeast corner is.

Field Summary

Fields

Modifier and Type	Field	Description
`double`	`lat`	The latitude of the Northeast corner.
`double`	`lng`	The longitude of the Northeast corner.

Constructor Summary

Constructors

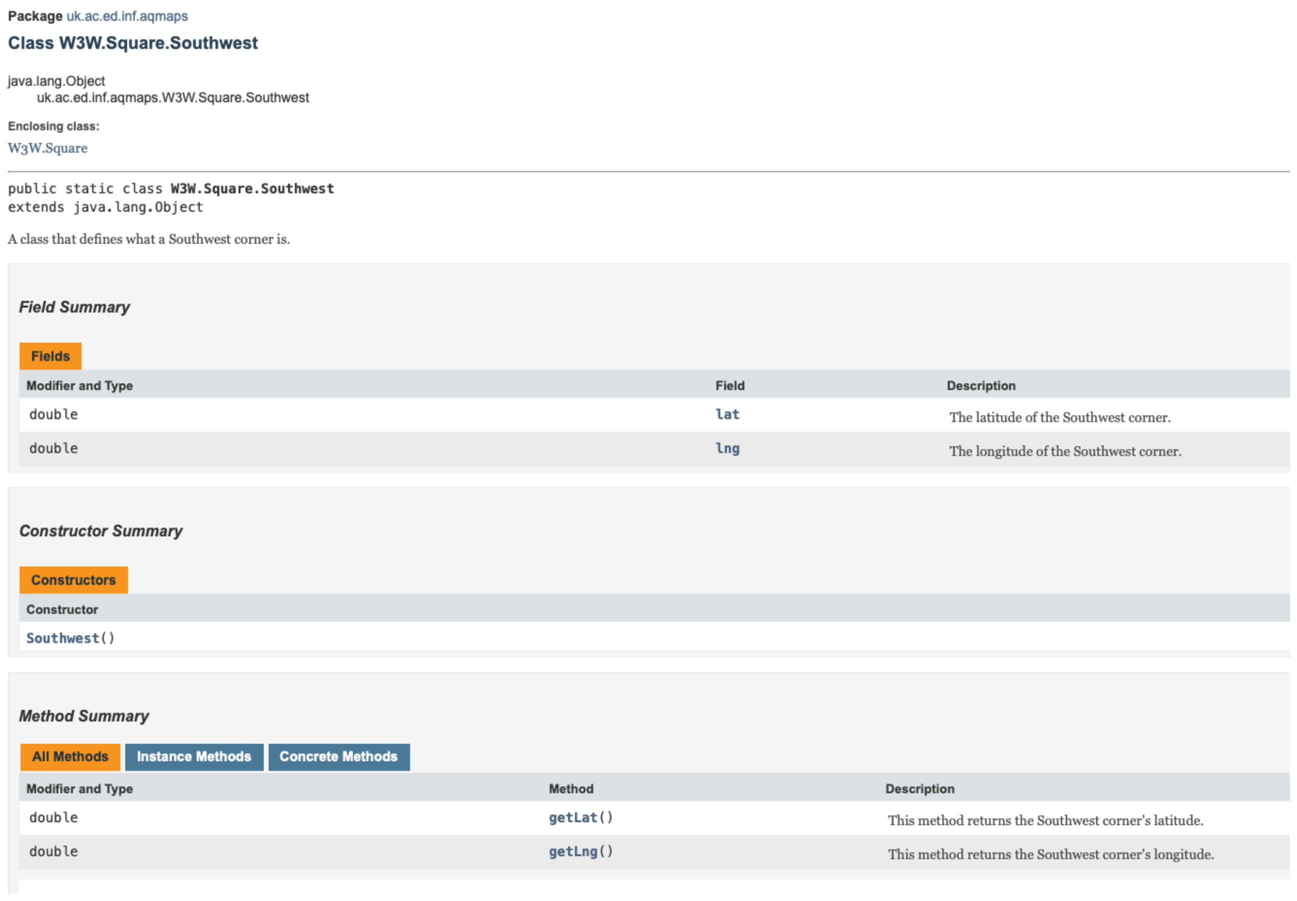
Constructor
`Northeast()`

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method	Description
`double`	`getLat()`	This method returns the Northeast corner's latitude.
`double`	`getLng()`	This method returns the Northeast corner's longitude.

8.2 W3W.Square.Southwest


Package uk.ac.ed.inf.aqmaps

Class W3W.Square.Southwest

java.lang.Object
    uk.ac.ed.inf.aqmaps.W3W.Square.Southwest

Enclosing class:
W3W.Square

```
public static class W3W.Square.Southwest
extends java.lang.Object
```

A class that defines what a Southwest corner is.

Field Summary

Fields

Modifier and Type	Field	Description
`double`	`lat`	The latitude of the Southwest corner.
`double`	`lng`	The longitude of the Southwest corner.

Constructor Summary

Constructors

Constructor
`Southwest()`

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`double`	`getLat()`	This method returns the Southwest corner's latitude.
`double`	`getLng()`	This method returns the Southwest corner's longitude.

Drone Algorithm

The algorithm to control the flight of the drone is determined in two parts: first, the calculation of the order in which the drone visits the sensors to take readings, and second, the drone's ability to detect and avoid No-Fly Zones. Let us begin by discussing the former.

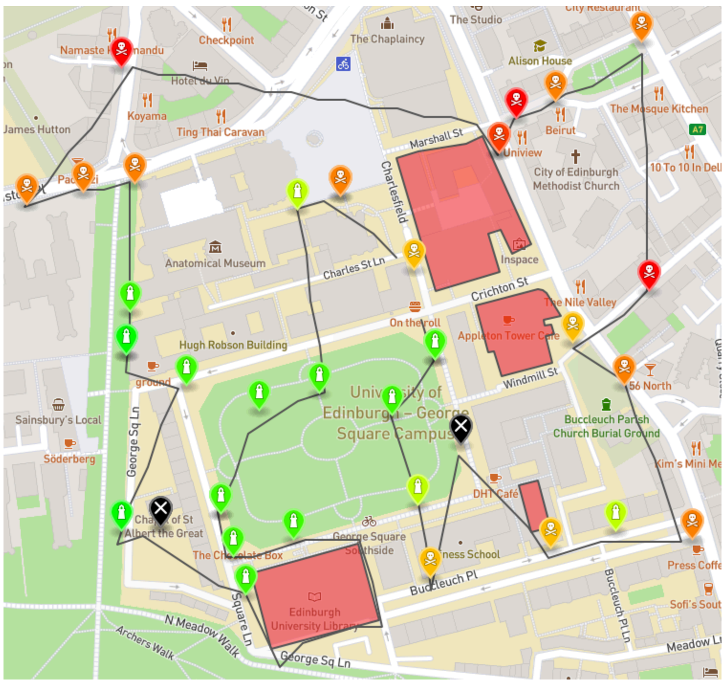
The question of choosing the permutation of sensors to visit is an example of the Euclidean case of the Traveling Salesman Problem (TSP). The TSP is as thus - given a list of cities where one can travel between any pair of cities, what is the shortest path one can take such that one visits each and every city, and ends at the origin city. Such a path is an optimized version of a "Hamiltonian cycle" - a path where each city is visited strictly once, but the total distance, or cost, isn't taken into account. In this problem, the sensors are "cities"; indeed, the drone can travel between any pair of sensors provided the flight path does not intersect with a No-Fly Zone. A naïve solution to determining the order in which to visit the sensors would be to construct the most obvious Hamiltonian cycle, which would be the identity permutation. This implies that the drone would visit the sensors in whatever order they exist in the corresponding *air-quality-data.json* file for that particular day. Although this would work in principle, this approach is flawed, especially when considering that the drone has a strict 150 move limit. Using the identity permutation could thus result in a scenario where the path is so un-optimal and long that it would take more than 150 moves for the drone to complete its tour and return close to its starting position.

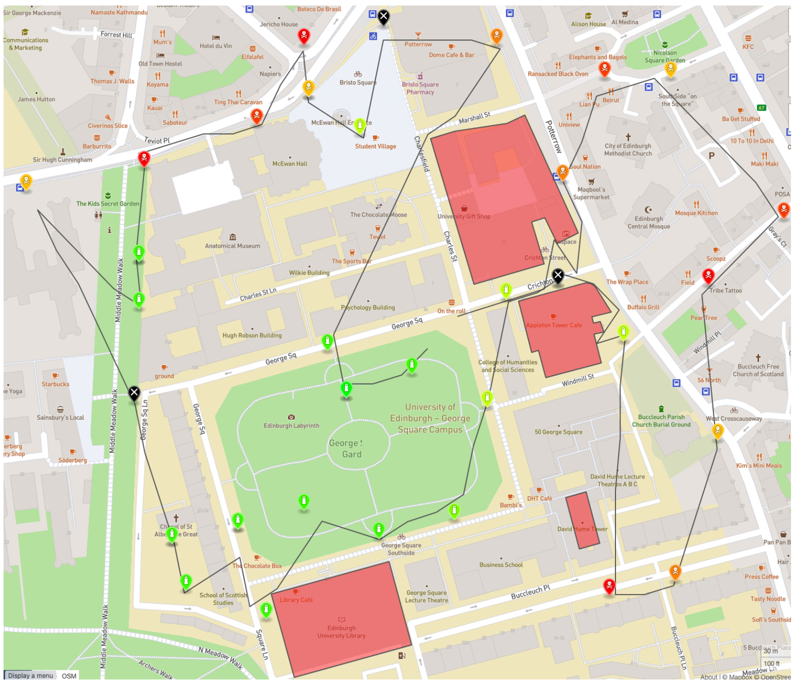
Thus, a better solution might be a greedy best-first search approach. This starts by finding the closest sensor to the drone's initial position. Then, it finds the sensor closest to that sensor, and so and so forth, until all of the sensors have been accounted for. Thus, at each sensor, the algorithm finds the closest sensor to it that has not been already visited. To determine this, first, a matrix of the straight-line Euclidean distances between any pair of sensors is calculated - this is known as the "adjacency matrix" of the "graph" of sensors. Then, the **Greedy()** method in the **FlightPlan** class finds the permutation of sensors to visit as per the solution described above.

While this is a big improvement over the identity permutation or any arbitrary Hamiltonian cycle, this is also a naïve approach as it might result in a sub-optimal path. Thus, another “heuristic”, or approach, is considered. After the greedy algorithm is performed on the permutation, the swap heuristic - as implemented in the **FlightPlan** class as **swapHeuristic() -** swaps the order of every single consecutive pair of sensors in the permutation and checks the length of the flightpath after every swap. If the total distance travelled by the drone is shorter, it commits this new permutation as *the* permutation. If not, it reverts it back to what it was. However, even this method may not result in a perfectly optimized path. Thus, after this is applied to the permutation, the Two-Opt Heuristic - implemented in the **FlightPlan** class as **twoOptHeuristic()** - is applied. The Two-Opt Heuristic reverses the order of every possible segment of the flight plan’s permutation and checks the tour value. Since it requires a starting and stopping index to determine which segment of the flight plan's permutation it needs to reverse, the number of combinations it checks is equal to the square of the number of sensors. Once again, if the reversal of a segment results in a shorter path, it commits that, and reverts it otherwise. Thus, after applying all three heuristics - greedy, followed by swap, followed by Two-Opt - to the identity permutation, an optimal path is found. There is even an improvement of around 0.0001 degrees between the initial application of the greedy and the final “tuned” path with the swap and Two-Opt heuristics.

It is insufficient to simply create an optimal order in which to visit the sensors - one must also consider the possibility of the drone flying into a No-Fly Zone. In order to account for this, before the drone even makes a move, the **calculateNextPositions()** function in the **Drone** class divides the line segment between its current position and its next position into 100 small segments - each of which is three-millionths of a degree in length. Then, the **inNoFlyZone()** function in the **Drone** class checks that each and every one of these points does not exist in any of the No-Fly Zones. If even one of these points is in any No-Fly Zone, the drone subtracts 10 degrees from the direction its facing in - in other words, it rotates 10 degrees in an easterly direction relative to its current position. Then, this process of dividing the line segment and checking is repeated until none of the infinitesimally small line segments are in any of the No-Fly Zones. Only then is a move made, and a reading attempted if it is close enough to a sensor.

The combination of the fine tuning of the flight path, and the rigorous checking to ensure that the drone does not collide with any No-Fly Zones results in the following two flight paths. The left is for the air quality sensors on October 10, 2020, and the right is for the air quality sensors on May 5, 2020. Even though some paths might appear to intersect a No-Fly Zone, upon zooming into the relevant region, it is clear that the drone does not.


Checkpoint
The Chaplaincy
The Studio
Alison House
Hotel du Vin
Koyama
Ting Thai Caravan
James Hutton
The Mosque Kitchen
Beirut
Marshall St
Uniview
City of Edinburgh Methodist Church
10 To 10 In Delhi
Charlesfield
Charles St Ln
Anatomical Museum
Inspace
Crichton St
The Nile Valley
On the roll
Appleton Tower Cafe
Hugh Robson Building
Windmill St
56 North
University of Edinburgh - George Square Campus
Buccleuch Parish Church Burial Ground
Sainsbury's Local
Soderberg
George Sq Ln
DHT Café
Kim's Mini Meals
Chapel of St Albert the Great
The Chocolate Box
George Square Southside
Business School
Press Coffee
Buccleuch Pl
Buccleuch Pl Ln
Sofi's South
Edinburgh University Library
Square Ln
N Meadow Walk
Archers Walk
George Sq Ln
Meadow Ln


Forrest Hill
Hotel du Vin
Bristo Square
Potterrow
Marshall St
McEwan Hall
Student Village
Sir Hugh Cunningham
The Kids Secret Garden
Anatomical Museum
Charles St Ln
Hugh Robson Building
George Sq
Appleton Tower Cafe
Middle Meadow Walk
University of Edinburgh - George Square Campus
Edinburgh Labyrinth
Buccleuch Parish Church Burial Ground
David Hume Lecture Theatre
Business School
Buccleuch Pl
The Chocolate Box
Library Cafe
Edinburgh University Library
George Square Lecture Theatre
N Meadow Walk
Archers Walk
Display a menu
OSM