Exam Report

PG4200

Algorithm and data structures

Høyskolen Kristiania

06/04-24 – 07/05-24

Table Of Contents

The dataset

The database is downloaded from: https://simplemaps.com/data/world-cities.

Assignment 1 – Merge Sort

a) In this assignment, we were sorting the cities latitudes using merge sort. The most common thing to do when it comes to latitudes in the world map is sorting it in ascending order (-90 to 90 degrees) This corresponds to the latitudes on the globe, where the equator is 0 degrees.

b) The number of counts needed to sort the dataset: **748220**. We implemented a Merge Count that counted all the performing merges to solve the problem. The number of counts needed to sort the dataset when randomly ordering the list before merge sort: **748220**. It does not affect the total number of merges. The Merge Sort algorithm works by comparing and sorting the elements in the list based on their values, regardless of their order in the list. Therefore, even though the order of elements changes, the number of merges needed to sort the dataset remains the same because the same set of elements must be compared and sorted regardless of their order. Random shuffling of the list has no effect on the actual sorting performed by the Merge Sort algorithm, and therefore the number of merges remains the same whether we use the random shuffling or not.[1]

Assignment 2 – Quick Sort

a) The regular QuickSort algorithm based on Latitude was performed in this way:
    a. Sorts the list of cities by a partition process where a pivot index is found (the last element), and then splits the list into two partitions. It iterates over the sublist and compares each city to the pivot, moving cities less than or equal to the pivot to the left side of the sublist.
    b. The swap method swaps cities in the list to rearrange their positions during sorting.
    c. The time complexity is O(n log n)

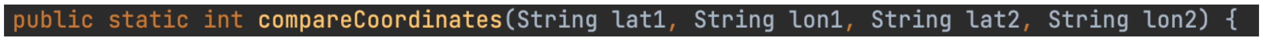
b) The number of quick sorts is: **507410**, and it changes when the list is shuffled. This is because the order of elements affects the number of comparisons and swaps required during the sorting process. When the list is shuffled, the process in quick sort encounters different pivot elements and therefore performs a different number of comparisons and swaps. In contrast, when the list is not randomized, the partitioning process encounters the same elements in the same order during each recursive call. The number of comparisons and swaps remains consistent across multiple executions of Quick Sort on the same list, resulting in a stable count of Quick Sort operations. [1]

Compare coordinates (Latitude and Longitude)

We chose the Haversine formula for calculating distances between latitudes and longitudes due to its accuracy, simplicity and computational efficiency [2]. This formula accurately represents distances on a sphere like Earth and is widely used for geographic calculations. Its simplicity and consistent results make it preferable, and it performs well for tasks requiring distance calculations between points. This is the formula used:

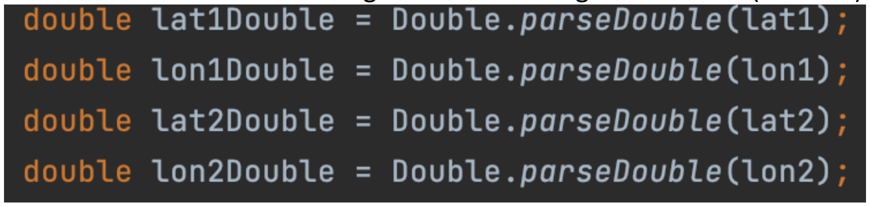
Haversine(θ) = sin^2(θ/2)

The first method: *compareCoordinates* takes parameters which is the latitudes and longitudes for two points.



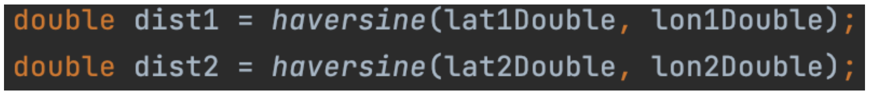
```java
public static int compareCoordinates(String lat1, String lon1, String lat2, String lon2) {
```

Converts the latitudes and longitudes from strings to decimals (doubles):



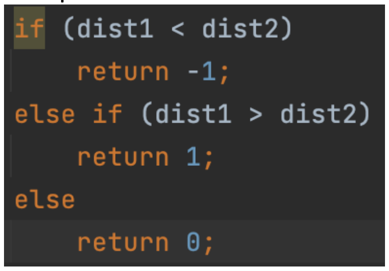
```java
double lat1Double = Double.parseDouble(lat1);
double lon1Double = Double.parseDouble(lon1);
double lat2Double = Double.parseDouble(lat2);
double lon2Double = Double.parseDouble(lon2);
```

Sends the converted latitudes and longitudes into the method which calculates the distance using the haversine formula:



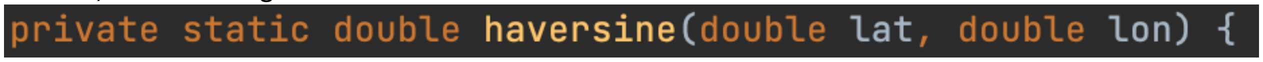
```java
double dist1 = haversine(lat1Double, lon1Double);
double dist2 = haversine(lat2Double, lon2Double);
```

Compares the calculated distances and returns the result:



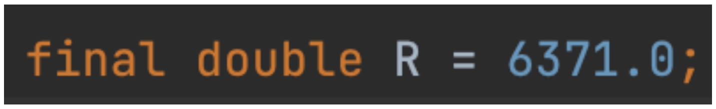
```java
if (dist1 < dist2)
    return -1;
else if (dist1 > dist2)
    return 1;
else
    return 0;
```

The *Haversine* method takes two parameters which is given from the *compareCoordinates* method: lat - latitude, and lon - longitude:



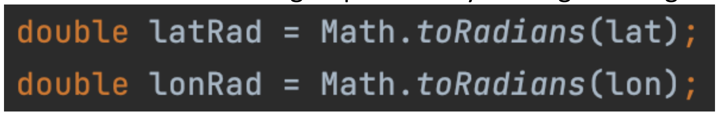
```java
private static double haversine(double lat, double lon) {
```

Declares the radius of the Earth in kilometers:



```java
final double R = 6371.0;
```

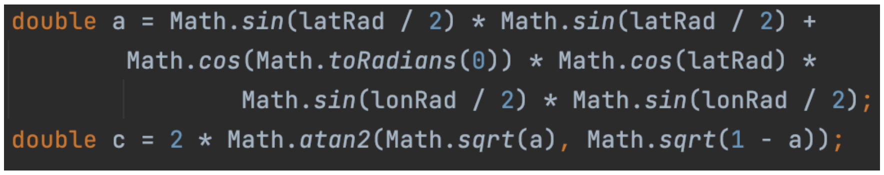
Since the values for the longitudes and latitudes are given in degrees, we have to convert them into *radians*. This is necessary because the trigonometric mathematical functions (cos and sin) further down in the code expect input angles to be in radians rather than degrees. Radians is after all just another unit for measurement for angles particularly dealing with trigonometric functions.



```java
double latRad = Math.toRadians(lat);
double lonRad = Math.toRadians(lon);
```

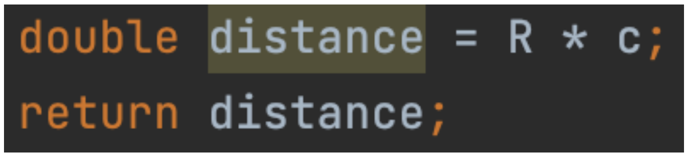
This is the actual *Haversine* formula calculating the distances.

- Math.sin(latRad / 2) * Math.sin(latRad / 2) +
  Calculates the square of the half of the difference in latitudes. The square difference in latitudes
- Math.cos(Math.toRadians(0)) * Math.cos(latRad) *
  Calculates the product of the cosine of half the difference in longitudes and the cosine of the latitude.
  - Math.toRadians(0): converts 0 degrees to radians, used to represent the equator in the formula.
  - Math.cos(latRad) : calculates the cosine of the latitude
- Math.sin(lonRad / 2) * Math.sin(lonRad / 2) +
  Calculates the square of the half of the difference in longitudes.
- These individual calculations are then summed up to compute "a", which represents a part of the Haversine formula.



```
double a = Math.sin(latRad / 2) * Math.sin(latRad / 2) +
        Math.cos(Math.toRadians(0)) * Math.cos(latRad) *
                Math.sin(lonRad / 2) * Math.sin(lonRad / 2);
double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
```

- double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
  - Math.sqrt(a): calculates the square root of "a"
  - Math.sqrt(1 - a): calculates the root of (1 - a)
  - double c: computes the arc tangent of the quotient of the two square roots.
  - Math.atan2(Math.sqrt(a), Math.sqrt(1 - a)): returns the angle whose tangent is the quotient of its arguments for obtaining angles in radians given rectangular coordinates. In this case it is used to obtain an angle that represents a fraction of the Earth's circumference



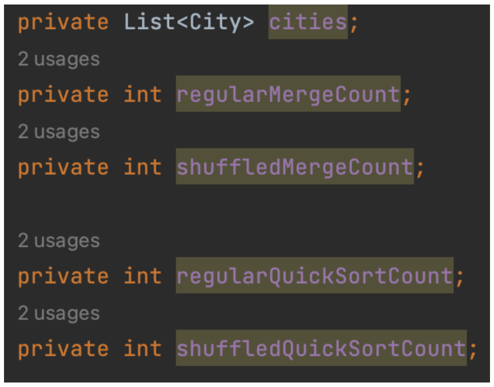
```
double distance = R * c;
return distance;
```

Lastly the distance is calculated by multiplying the output of the calculation in *double c* (which is based on *double a*) with the *R* which represents the Earth's radius and then returns that value.

CSVResult

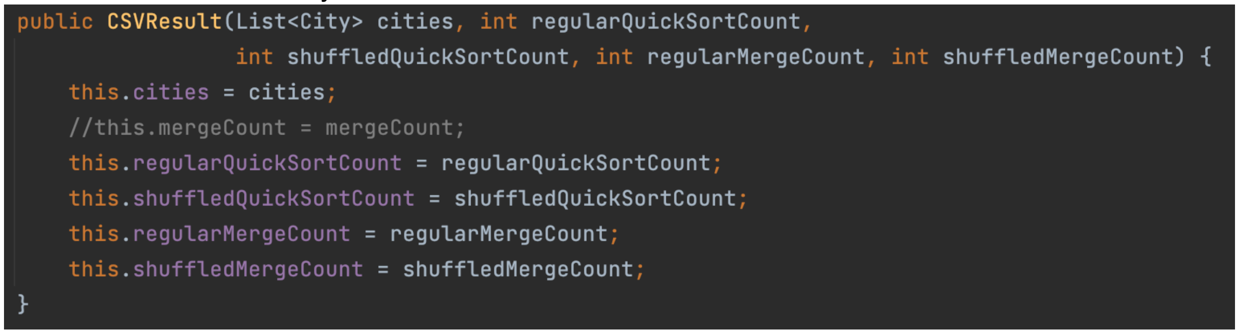
This class is designed to hold results from sorting algorithms performed on a list of cities. In other words is this class for encapsulating the results of sorting operations for cities and providing methods to access these results. It also stores the counts of comparisons or swaps made during the sorting processes for both regular and shuffled versions of these algorithms.

Firstly we initialize the variables holding a list of City objects, the counts of regular and shuffled merge and quick sort



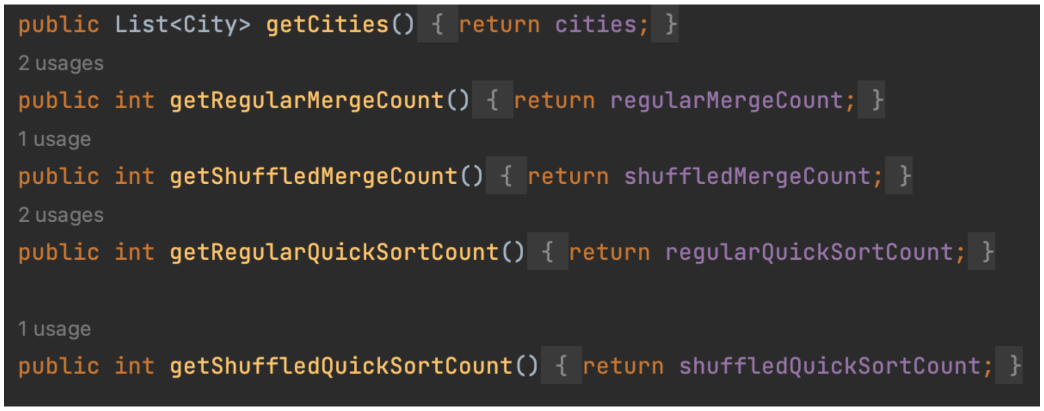
```
private List<City> cities;
2 usages
private int regularMergeCount;
2 usages
private int shuffledMergeCount;

2 usages
private int regularQuickSortCount;
2 usages
private int shuffledQuickSortCount;
```

The constructor which initializes the CSVResult object. The constructor assignments initializes the instance variables of an object when it is created.



```
public CSVResult(List<City> cities, int regularQuickSortCount,
                 int shuffledQuickSortCount, int regularMergeCount, int shuffledMergeCount) {
    this.cities = cities;
    //this.mergeCount = mergeCount;
    this.regularQuickSortCount = regularQuickSortCount;
    this.shuffledQuickSortCount = shuffledQuickSortCount;
    this.regularMergeCount = regularMergeCount;
    this.shuffledMergeCount = shuffledMergeCount;
}
```

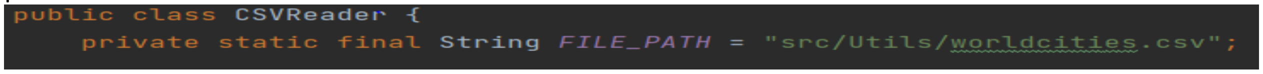
Initialize the *Getter* methods. These retrieve the values of private instance variables of a class. This helps us to have controlled access to the state of an object by returning the value of its private fields, allowing other parts of the program to obtain information about the object without directly accessing its internal state.



```
public List<City> getCities() { return cities; }
2 usages
public int getRegularMergeCount() { return regularMergeCount; }
1 usage
public int getShuffledMergeCount() { return shuffledMergeCount; }
2 usages
public int getRegularQuickSortCount() { return regularQuickSortCount; }

1 usage
public int getShuffledQuickSortCount() { return shuffledQuickSortCount; }
```

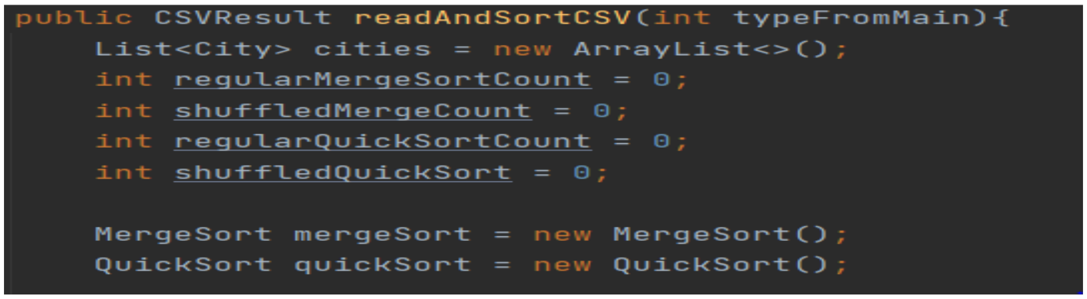
CSVReader

We used a CSVReader class to take the information from the worldcities.csv and turn it into data we could use in our code. This variable is used to find the csv File on the computer. It has to have the right path to the worldcities.csv-file for the solution to work.



```
public class CSVReader {
    private static final String FILE_PATH = "src/Utils/worldcities.csv";
```

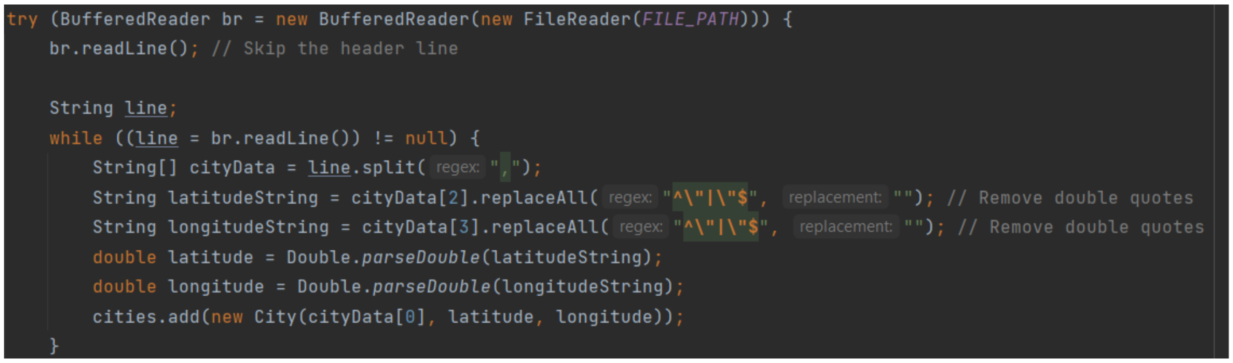
In our CSVReader we Created a CSVResult that controls our application. It takes in an int from the terminal (typeFromMain) that coincides with what you want the application to solve.



```
public CSVResult readAndSortCSV(int typeFromMain){
    List<City> cities = new ArrayList<>();
    int regularMergeSortCount = 0;
    int shuffledMergeCount = 0;
    int regularQuickSortCount = 0;
    int shuffledQuickSort = 0;

    MergeSort mergeSort = new MergeSort();
    QuickSort quickSort = new QuickSort();
```

First we use the variables needed from our CSVResult constructor. We also instantiate A new MergeSort and QuickSort so we can use the different methods from our MergeSort and QuickSort Classes.

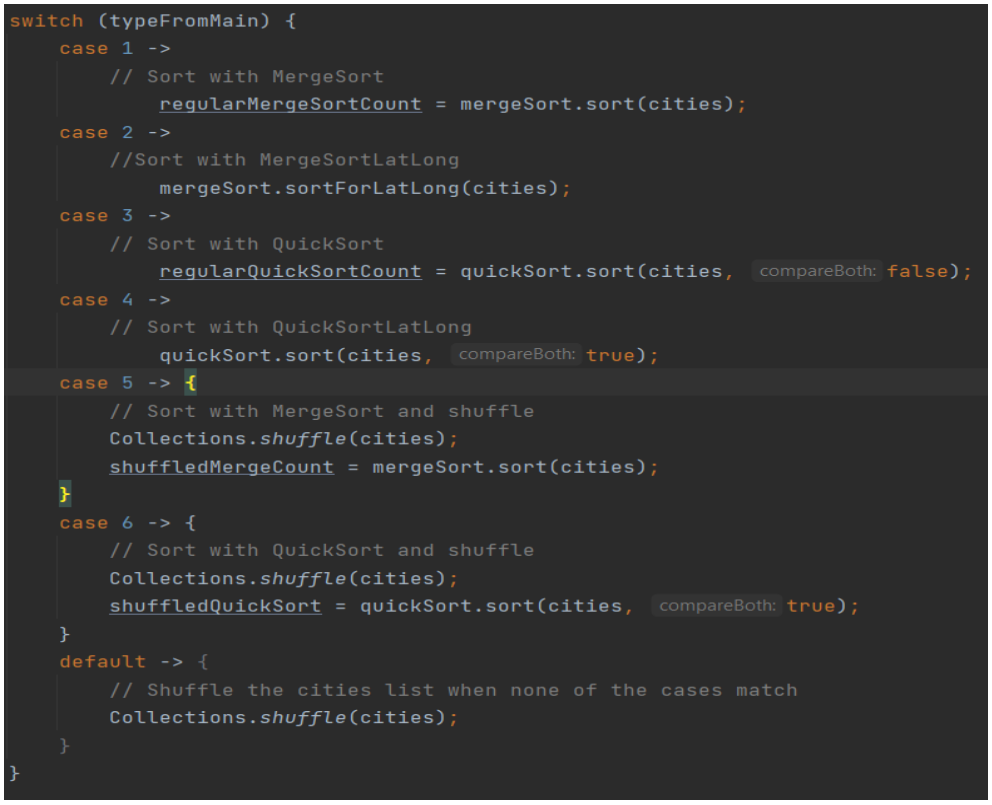
The information in the .csv file needs to be put into variables so we can use the information in our program. We do this through a javaClass that is called BufferedReader. The BufferedReader extends Reader. It`s used to read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays and lines.



```
try (BufferedReader br = new BufferedReader(new FileReader(FILE_PATH))) {
    br.readLine(); // Skip the header line

    String line;
    while ((line = br.readLine()) != null) {
        String[] cityData = line.split( regex: ",");
        String latitudeString = cityData[2].replaceAll( regex: "^\"|\"$",  replacement: ""); // Remove double quotes
        String longitudeString = cityData[3].replaceAll( regex: "^\"|\"$",  replacement: ""); // Remove double quotes
        double latitude = Double.parseDouble(latitudeString);
        double longitude = Double.parseDouble(longitudeString);
        cities.add(new City(cityData[0], latitude, longitude));
    }
```

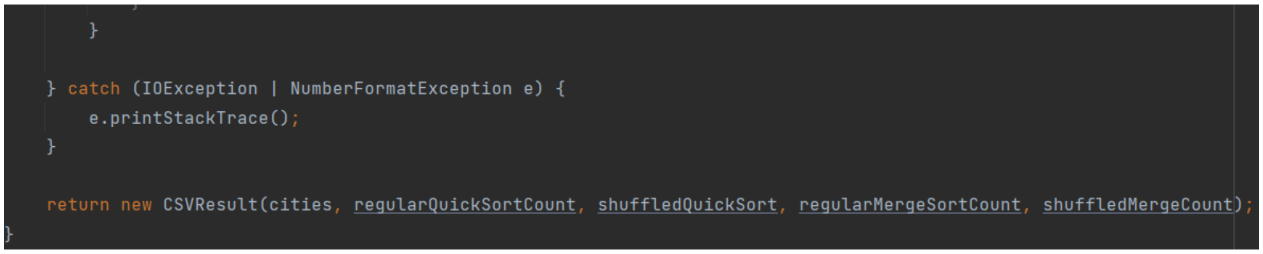
First we Skip the header line in the csv file through br.readLine(), this is so the name of the values in the documents is skipped. Then a variable String line is created, so that every line the bufferedreader reads is stored in a variable. A while loop is used to read through the whole document. The data from the readline is stored in an Array called *citydata*, it also tells the bufferedreader when to start a newline. Since the Assignments want us to use mergesort and quickSort algorithms and sort cities depending on the latitude and longitude value we need to get that data from the document. In the document we can see that latitude is stored in the third column and longitude is stored in the fourth column. Since the values from an array starts from 0 we use the second and third value to create two new String Variables latitudeString and lontitudeString. The String variables are pretty useless to us since we need numbers to Sort the information. So the String variables are converted to Double variables. Finally the information gathered is used to create a new city object and add the city to a list.

now that we have read the information in the .csv file. and converted it to Data in our program. We use a Java Switch Case to control our application.



```
switch (typeFromMain) {
    case 1 ->
        // Sort with MergeSort
            regularMergeSortCount = mergeSort.sort(cities);
    case 2 ->
        //Sort with MergeSortLatLong
            mergeSort.sortForLatLong(cities);
    case 3 ->
        // Sort with QuickSort
            regularQuickSortCount = quickSort.sort(cities,  compareBoth: false);
    case 4 ->
        // Sort with QuickSortLatLong
            quickSort.sort(cities,  compareBoth: true);
    case 5 -> {
        // Sort with MergeSort and shuffle
        Collections.shuffle(cities);
        shuffledMergeCount = mergeSort.sort(cities);
    }
    case 6 -> {
        // Sort with QuickSort and shuffle
        Collections.shuffle(cities);
        shuffledQuickSort = quickSort.sort(cities,  compareBoth: true);
    }
    default -> {
        // Shuffle the cities list when none of the cases match
        Collections.shuffle(cities);
    }
}
```

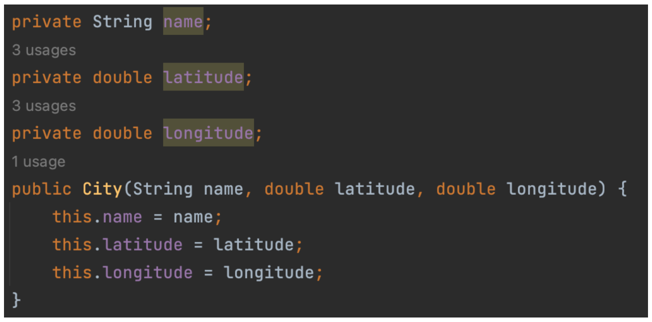
Here we implement all our methods from the MergeSort and Quicksort Classes. when Sorting through the QuickSort method the boolean value in the methods are also set here. Every Java Switch Case needs a catch in case some of the cases do not work so we round it out with a catch. Then we return the new CSVResults that are used in Main().



```
        }

    } catch (IOException | NumberFormatException e) {
        e.printStackTrace();
    }

    return new CSVResult(cities, regularQuickSortCount, shuffledQuickSort, regularMergeSortCount, shuffledMergeCount);
}
```

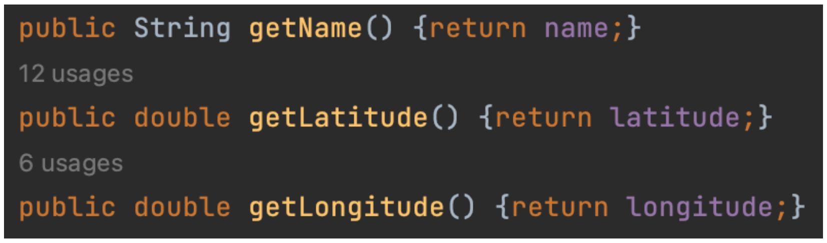
City

Initialize the variables for the City-object, the constructor and the constructor assignments.



```java
private String name;
3 usages
private double latitude;
3 usages
private double longitude;
1 usage
public City(String name, double latitude, double longitude) {
    this.name = name;
    this.latitude = latitude;
    this.longitude = longitude;
}
```

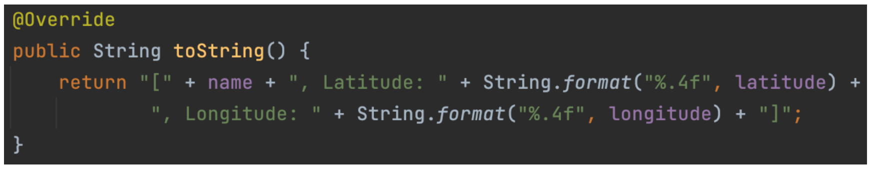
Initialize the getter methods which retrieve the values of the private instance of the variables. This gives us controlled access to the state of the city-object by returning the values of its private fields without directly accessing.



```java
public String getName() {return name;}
12 usages
public double getLatitude() {return latitude;}
6 usages
public double getLongitude() {return longitude;}
```

Lastly the toString method is used to return a string representation of the City object. The toString method is overridden to provide a custom string representation for the “City” objects.

%.4f : formats the floating-point number to include four decimals.

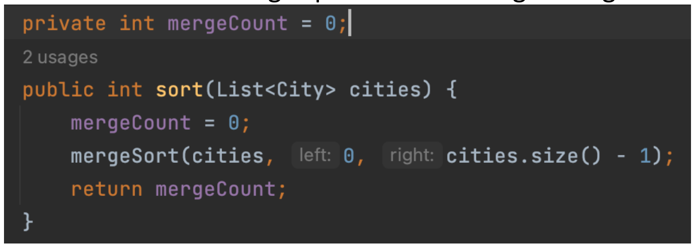


```java
@Override
public String toString() {
    return "[" + name + ", Latitude: " + String.format("%.4f", latitude) +
            ", Longitude: " + String.format("%.4f", longitude) + "]";
}
```

MergeSort

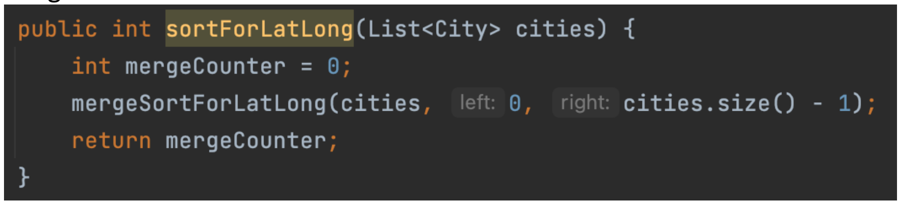
The merge sort algorithm uses the principle of divide and conquer. It divides the values of the array into two sub-arrays.

Initializing a variable that holds the count of merges.
This method mainly calls on the mergeSort method to perform the merges and then keeps and returns the count of total merges performed during sorting.



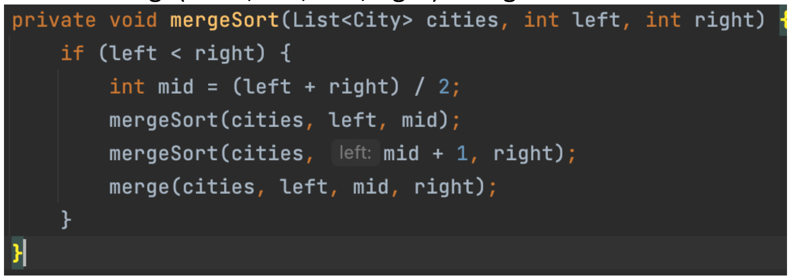
```java
private int mergeCount = 0;
2 usages
public int sort(List<City> cities) {
    mergeCount = 0;
    mergeSort(cities, left: 0, right: cities.size() - 1);
    return mergeCount;
}
```

This method has the same purpose as the method above. The one above are calling on the method to sort based on latitude. However, this one is calling on the method for sorting based on Latitude and Longitude.

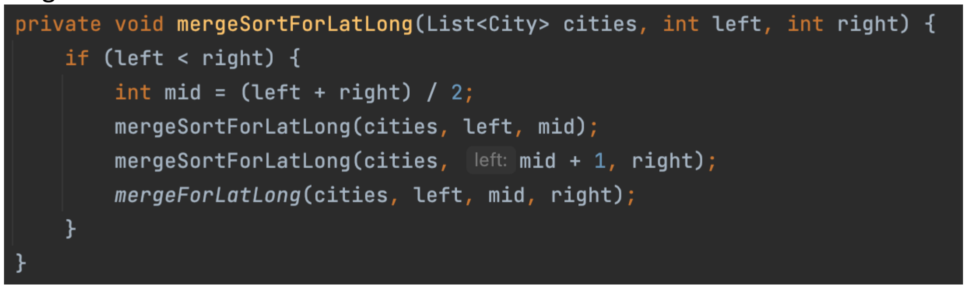

```java
public int sortForLatLong(List<City> cities) {
    int mergeCounter = 0;
    mergeSortForLatLong(cities, left: 0, right: cities.size() - 1);
    return mergeCounter;
}
```

Performs the regular merge sort. It divides the list into smaller sublists until each sublist contains only one element (when “left” is equal to “right”). And then, it merges these sublists in a sorted order.

- It takes three parameters: a list of cities, left and right. The *Left* and *right* parameters represent indexes of the sublist being sorted. *Left* is the index of the leftmost element of the sublist being sorted. *Right* is the index of the rightmost element of the sublist being sorted.
- int mid: this variable calculates the middle index “mid” of the sublist. This index is used to split the sublist into two halves.
- mergeSort(cities, left, mid): sorts the left sublist
- mergeSort(cities, mid + 1, right) : sorts the right sublist
- merge(cities, left, mid, right): merges the sorted sublists.


```java
private void mergeSort(List<City> cities, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(cities, left, mid);
        mergeSort(cities, left: mid + 1, right);
        merge(cities, left, mid, right);
    }
}
```

This method has the same structure as the mergeSort method above. However, there are some differences: It calls on other sorting methods: *mergeSortForLatLong and mergeSortForLatLong.* The two first once method sorts the sublists. The last one merges the sublists based on the latitudes and longitudes.


```java
private void mergeSortForLatLong(List<City> cities, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSortForLatLong(cities, left, mid);
        mergeSortForLatLong(cities, left: mid + 1, right);
        mergeForLatLong(cities, left, mid, right);
    }
}
```

Merges two sorted lists. It receives the sorted sublists from the *mergeSort* method. Then a temporary list is made to store merged elements.
“i”: is the starting index of the left sublist
“j”: is the starting index of the right sublist

while(i <= mid && j<= right):

- In this loop we iterate through both sublists. In the "if"- statement we compare the elements from the left to the right sublists and adding the smaller one to the "temp" list.
- The loop continues until either of the sublists is exhausted.
- During each iteration , we compare the latitude values of the cities at indices "i" and "j". If the latitude of the city at index "i" is less than or equal to the latitude of the city at index "j", we add the city at index "i" to the "temp" list and increment "i". Otherwise, we add the city at index "j" to the "temp" list and increment "j".

while(i <= mid):

- After one of the sublists is exhausted, we may have remaining elements in the other sublist.
- In this loop, we add any remaining elements form the *left* sublist (if any) to the "temp" list by iterating from "i" to "mid"

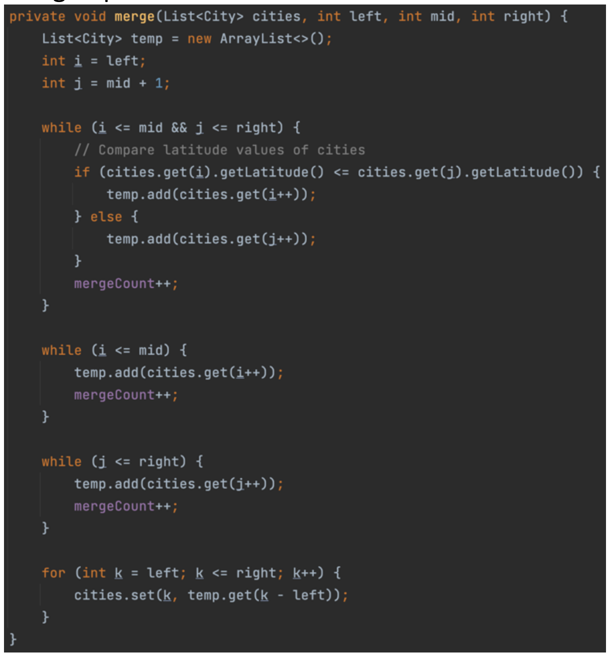
while(j<= right)

- Similarly, we add the remaining elements from the right sublist (if any) to the "temp" list by iterating from "j" to "right".

for(int k=left; k<= right: k++):

- Here we copy the merged elements from the "temp" list back to the original list *cities*.
- We iterate form "left" to "right", copying elements from the "temp" list to the corresponding position in the original list.
- The "k -left" is used to map the indices from the "temp" list back to the original list.

Throughout this whole method, we increment the "mergeCount" variable each time we add an element to the "temp" list. This keeps track of the total number of comparisons and merges performed during the merge operation.



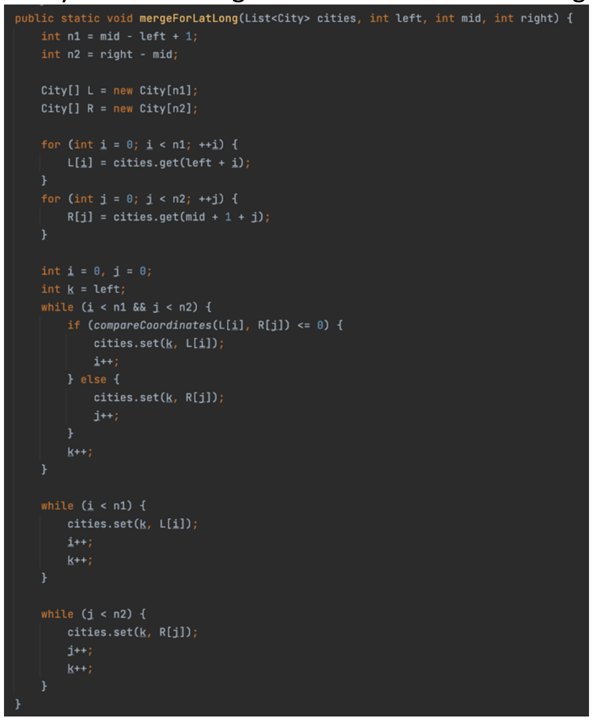
```java
private void merge(List<City> cities, int left, int mid, int right) {
    List<City> temp = new ArrayList<>();
    int i = left;
    int j = mid + 1;

    while (i <= mid && j <= right) {
        // Compare latitude values of cities
        if (cities.get(i).getLatitude() <= cities.get(j).getLatitude()) {
            temp.add(cities.get(i++));
        } else {
            temp.add(cities.get(j++));
        }
        mergeCount++;
    }

    while (i <= mid) {
        temp.add(cities.get(i++));
        mergeCount++;
    }

    while (j <= right) {
        temp.add(cities.get(j++));
        mergeCount++;
    }

    for (int k = left; k <= right; k++) {
        cities.set(k, temp.get(k - left));
    }
}
```

n1 and *n2*: calculates the size of the left and right sublists.
City[ ] L and *City[ ] R* : Creates temporary arrays for the left (L) and right (R) sublists.

The two for-loops copies the elements from the original list to the temporary arrays.
Now, the merging begins.

- “i” and “j” represent the indices of the current elements in the left (L) and right (R) sublists respectively.
- “k”: represents the starting index of the merged sublist in the original list “cities”.
- while (i < n1 && j < n2) : as long as there are remaining elements in both of the left and right sublist, this loop will continue.
    - The longitude and latitude values of the current elements are sent to the *compareCoordinates* method. This method compares the coordinates using the Haversine formula.
    - Then it compares the result/value from the *compareCoordinates* method in the current left sublist (L[i]) with the current element in the right sublist (R[j]). If the left is less than or equal to the right, then the element from the left sublist is added to the merged sublist “cities” at index “k”.
    - Otherwise, if right is less than the left, then the element from the right sublist is added to the merged sublist at index “k”.
    - After setting the element in the merged sublist, “i” or “j” is incremented to move to the next position in the merged sublist
    - Finally, “k” is incremented to move to the next position in the merged sublist.

Lastly the remaining elements in either left or right sublist are being copied back into the original list.



```java
public static void mergeForLatLong(List<City> cities, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    City[] L = new City[n1];
    City[] R = new City[n2];

    for (int i = 0; i < n1; ++i) {
        L[i] = cities.get(left + i);
    }
    for (int j = 0; j < n2; ++j) {
        R[j] = cities.get(mid + 1 + j);
    }

    int i = 0, j = 0;
    int k = left;
    while (i < n1 && j < n2) {
        if (compareCoordinates(L[i], R[j]) <= 0) {
            cities.set(k, L[i]);
            i++;
        } else {
            cities.set(k, R[j]);
            j++;
        }
        k++;
    }

    while (i < n1) {
        cities.set(k, L[i]);
        i++;
        k++;
    }

    while (j < n2) {
        cities.set(k, R[j]);
        j++;
        k++;
    }
}
```

compares two cities based on their latitude and longitude coordinates. The method calls the *compareCoordinates* method to send the coordinates to be compared. But first it coordinates are converted to strings.

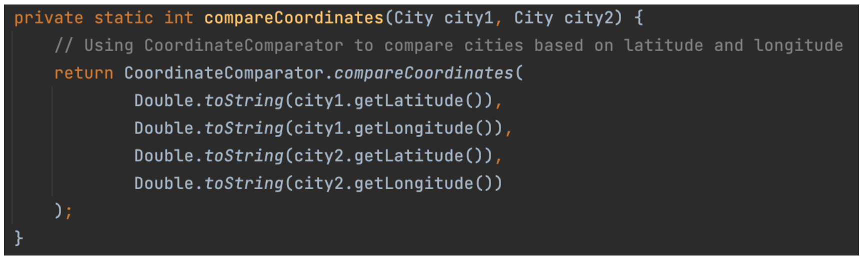
The method returns the calculated distances (dist1 and dist2) to the two cities (city1 and city2).

- If “dist1” is less than “dist2” it means the distance calculated for “city1” is less than the distance calculated for “city2”, so it returns -1.
- When “dist1” is bigger than “dist2”, it signifies that the distance calculated for “city1” is bigger than the distance calculated for “city2”, so it returns “1”.

- If “dist1” and “dist2” are equal, it returns 0.

the return value of the method indicated therefore the result of the comparison based on distances:

- “-1” if “dist1” is bigger
- “1” if “dist2” is bigger
- “0” if both are the same distance

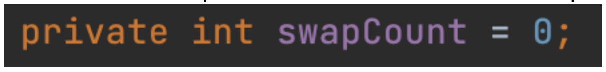


```java
private static int compareCoordinates(City city1, City city2) {
    // Using CoordinateComparator to compare cities based on latitude and longitude
    return CoordinateComparator.compareCoordinates(
            Double.toString(city1.getLatitude()),
            Double.toString(city1.getLongitude()),
            Double.toString(city2.getLatitude()),
            Double.toString(city2.getLongitude())
    );
}
```

QuickSort

In this class we use quicksort algorithms to sort a list of cities on their latitudes and longitudes coordinates. The partitioning scheme is used to divide the list into smaller sublists and recursively sort each sublist.

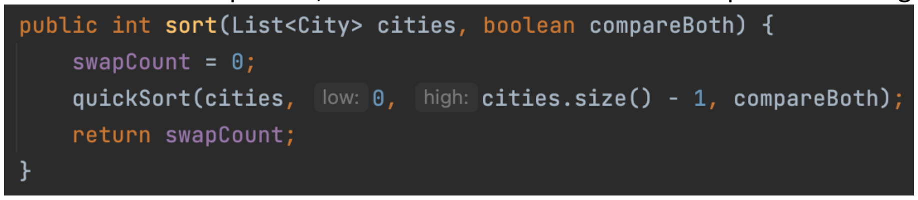
This counter keeps track of the numbers of swaps



```java
private int swapCount = 0;
```

This method performs the quick sort

- swapCount = 0; : reset swap count
- quickSort(cities, 0, cities.size() -1, compareBoth); : calles the quickSortMethod
- return swapCount; : returns the total number of swaps made during sorting



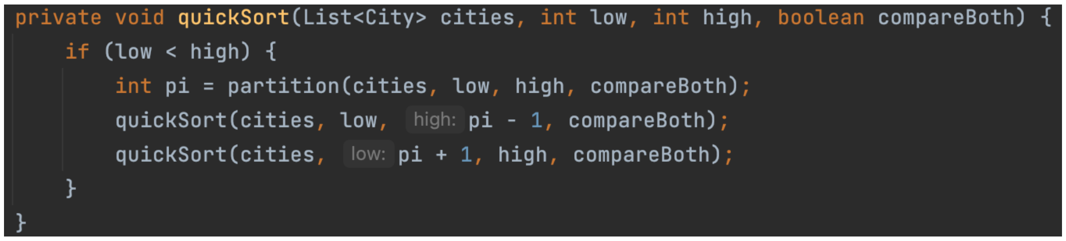
```java
public int sort(List<City> cities, boolean compareBoth) {
    swapCount = 0;
    quickSort(cities, low: 0, high: cities.size() - 1, compareBoth);
    return swapCount;
}
```

Recursive method to partition and sort the list of cities.

- It receives the information from the information given from the *sort* method above.
  - *List<City> cities:* list of cities to be stored.
  - *int low:* represents the index of the first element in the current partition of the list.
  - *int high:* represents the index of the last element in the current partition of the list.
  - *boolean compareBoth:* specifies whether to compare both latitude and longitude or just latitude during the sorting process.
- *if (low < high)*:
  - If the condition is true, it means that there are at least two elements in the current partition. If it is false, it will stop because the partition has only one element or is empty, so there is no need for sorting.
- *int pi = partition(cities, low, high, compareBoth);*

    - *pi* is the partition index which is used to split the list into two partitions. It is also excluded from the recursive calls because it's already in its final sorted position after the partition step.
    - the *partition* method is called to find the pi index (partition index)
- *quickSort(cities, low, pi - 1, compareBoth);*
    - This recursively sorts the elements to the left of the partition.
- *quickSort(cities, pi + 1, high, compareBoth);*
    - Sorts the elements to the right of the partition.

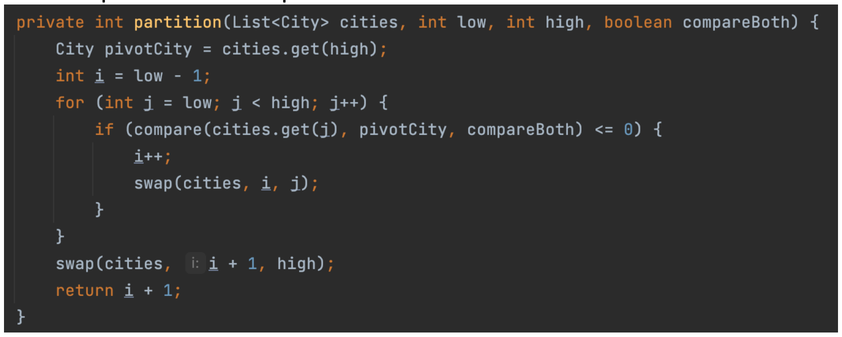
The *low* and *high* parameters determine the range of elements to be sorted in each recursive call. The quickSort algorithm recursively partitions the list around a pivot (pi) element until the entire list is sorted.



```
private void quickSort(List<City> cities, int low, int high, boolean compareBoth) {
    if (low < high) {
        int pi = partition(cities, low, high, compareBoth);
        quickSort(cities, low,  high: pi - 1, compareBoth);
        quickSort(cities,  low: pi + 1, high, compareBoth);
    }
}
```

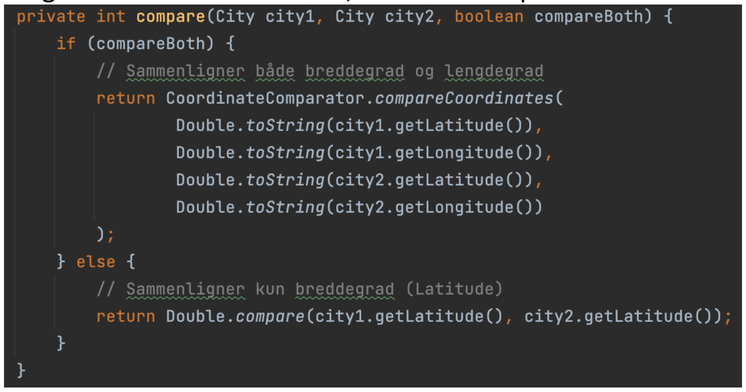
This method performs a partition on the list.

- Firstly, the pivot city is chosen which in this case is the last element.
- A variable *i* is set to keep track of the index of the last element that is less or equal to the pivot, which is the position before the start of the sublist (partition).
- The loop iterates over the sublist from index "low" to "high - 1". Here it examines each city in the sublist to determine if it should be placed before or after the pivot.
- The if-statement checks if the current city is less than or equal to the pivot city.
- If true: i (i++) is incremented to move to the next position of the sublist where the next city less than or equal to the pivot should be placed.
- *swap(cities, i, j)*: call swaps the current city (cities.get(j)) with the city at index "i". This operation effectively moves the current city to the correct position before the pivot.
- After the "for" loop completes, all cities less than or equal to the pivot placed before the pivot, and all cities greater than the pivot are placed after it
- *swap(cities, i + 1, hight)*: swaps the pivot city with the element at index "i + 1". This effectively places the pivot at its correct sorted position in the list.
- Finally, the method returns the partitioning index "i + 1", which indicates the final sorted position of the pivot in the list.



```
private int partition(List<City> cities, int low, int high, boolean compareBoth) {
    City pivotCity = cities.get(high);
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (compare(cities.get(j), pivotCity, compareBoth) <= 0) {
            i++;
            swap(cities, i, j);
        }
    }
    swap(cities,  i: i + 1, high);
    return i + 1;
}
```

Compares cities. The comparison is either based on the latitude value or both longitude and latitude values. If the boolean parameter "compareBoth" is true, the city will be compared based on both longitude and latitude. If not, it will be compared based on only the latitude value.

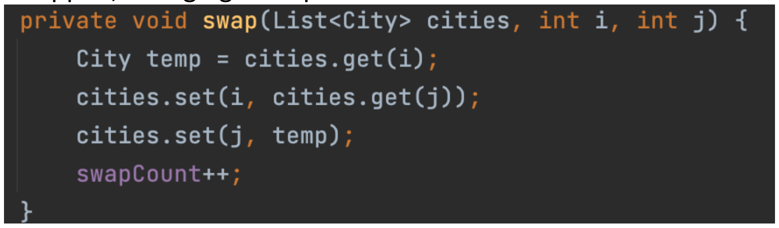


```java
private int compare(City city1, City city2, boolean compareBoth) {
    if (compareBoth) {
        // Sammenligner både breddegrad og lengdegrad
        return CoordinateComparator.compareCoordinates(
                Double.toString(city1.getLatitude()),
                Double.toString(city1.getLongitude()),
                Double.toString(city2.getLatitude()),
                Double.toString(city2.getLongitude())
        );
    } else {
        // Sammenligner kun breddegrad (Latitude)
        return Double.compare(city1.getLatitude(), city2.getLatitude());
    }
}
```

This method swaps the cities in the list.

- Firstly, we create a temporary variable and assigns in the value of the city at index "i" in the list "cities". This is necessary in order to store one of the cities temporarily before the swap.
- *cities.set(i, cities.get(j)):* replaces the city at index "i" in the list "cities" with the city at index "j". it moves the city at index "j" to the position originally occupied by the city at index "i"
- *cities.set(j, temp):* replaces the city at index "j" in the list "cities" with the city stored in the temporary variable "temp". This completes the swap by moving the city originally at index "i" to index "j"
- *swapCount++*: increments the "swapCount" variable.

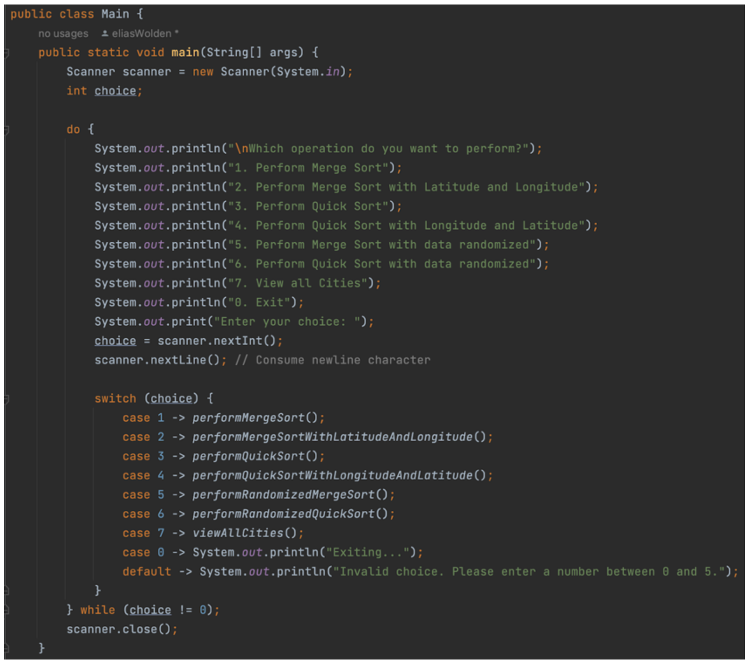
After the execution of the "swap" method, the cities at indexes "i" and "j" in the list "cities" have been swapped, changing their positions in the list.



```java
private void swap(List<City> cities, int i, int j) {
    City temp = cities.get(i);
    cities.set(i, cities.get(j));
    cities.set(j, temp);
    swapCount++;
}
```

Main

Scanner: take user input
int choice: variable that holds the users choice. Prints out the menu for all execution options. Switch-case based on the user choice.

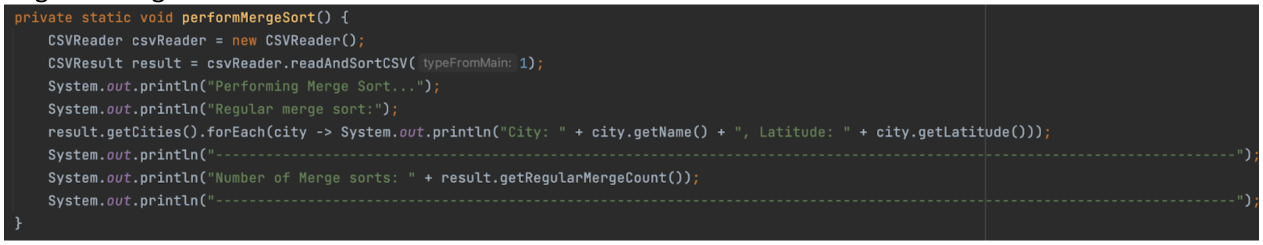


```java
public class Main {
    no usages   eliasWolden *
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int choice;

        do {
            System.out.println("\nWhich operation do you want to perform?");
            System.out.println("1. Perform Merge Sort");
            System.out.println("2. Perform Merge Sort with Latitude and Longitude");
            System.out.println("3. Perform Quick Sort");
            System.out.println("4. Perform Quick Sort with Longitude and Latitude");
            System.out.println("5. Perform Merge Sort with data randomized");
            System.out.println("6. Perform Quick Sort with data randomized");
            System.out.println("7. View all Cities");
            System.out.println("0. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();
            scanner.nextLine(); // Consume newline character

            switch (choice) {
                case 1 -> performMergeSort();
                case 2 -> performMergeSortWithLatitudeAndLongitude();
                case 3 -> performQuickSort();
                case 4 -> performQuickSortWithLongitudeAndLatitude();
                case 5 -> performRandomizedMergeSort();
                case 6 -> performRandomizedQuickSort();
                case 7 -> viewAllCities();
                case 0 -> System.out.println("Exiting...");
                default -> System.out.println("Invalid choice. Please enter a number between 0 and 5.");
            }
        } while (choice != 0);
        scanner.close();
    }
```

All the four methods have the same structure. Within the *csvReader.readAndSortCSV()* different numbers are sent in as a parameter, indicating what sorting method to be executed in the CSVReader file.

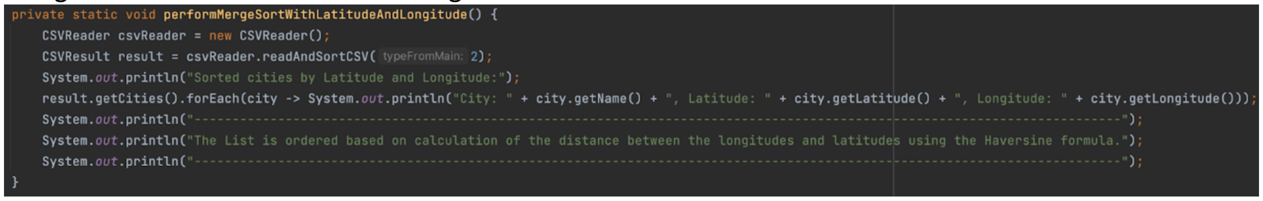
- *CSVReader csvReader = new CSVReader()*: creates an instance of the CSVReader class, which reads the CSV file.
- *CSVResult result = csvReader.readAndSortCSV(1)*: calls the "reandAndSortCSV()" method of the "CSVReader" instance, passing "x" as an argument. This argument acts as a signal to perform a sorting method specified in the CSVReader file. The method returns a "CSVResult" object containing the sorted list of cities and other information.
- *Prints:*
    - prints the city's name, latitude, and longitude.
    - the mergeSortCount performed.

Regular Merge Sort



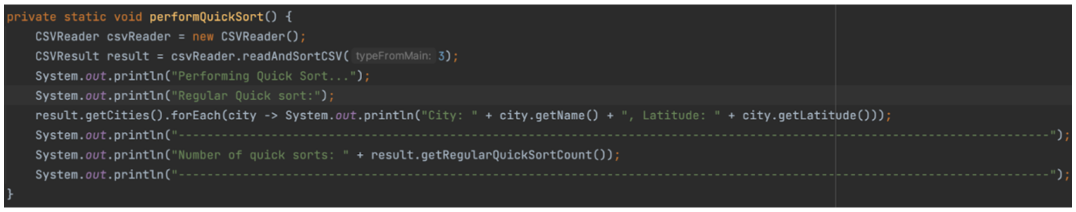
```java
private static void performMergeSort() {
    CSVReader csvReader = new CSVReader();
    CSVResult result = csvReader.readAndSortCSV( typeFromMain: 1);
    System.out.println("Performing Merge Sort...");
    System.out.println("Regular merge sort:");
    result.getCities().forEach(city -> System.out.println("City: " + city.getName() + ", Latitude: " + city.getLatitude()));
    System.out.println("----------------------------------------------------------------------------------------------------");
    System.out.println("Number of Merge sorts: " + result.getRegularMergeCount());
    System.out.println("----------------------------------------------------------------------------------------------------");
}
```

Merge Sort based on latitude and longitude:



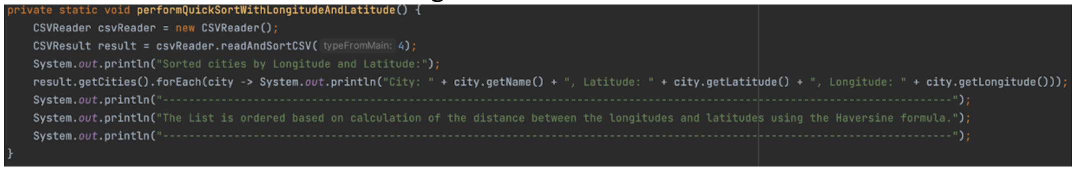
```java
private static void performMergeSortWithLatitudeAndLongitude() {
    CSVReader csvReader = new CSVReader();
    CSVResult result = csvReader.readAndSortCSV( typeFromMain: 2);
    System.out.println("Sorted cities by Latitude and Longitude:");
    result.getCities().forEach(city -> System.out.println("City: " + city.getName() + ", Latitude: " + city.getLatitude() + ", Longitude: " + city.getLongitude()));
    System.out.println("----------------------------------------------------------------------------------------------------");
    System.out.println("The List is ordered based on calculation of the distance between the longitudes and latitudes using the Haversine formula.");
    System.out.println("----------------------------------------------------------------------------------------------------");
}
```

Regular QuickSort.



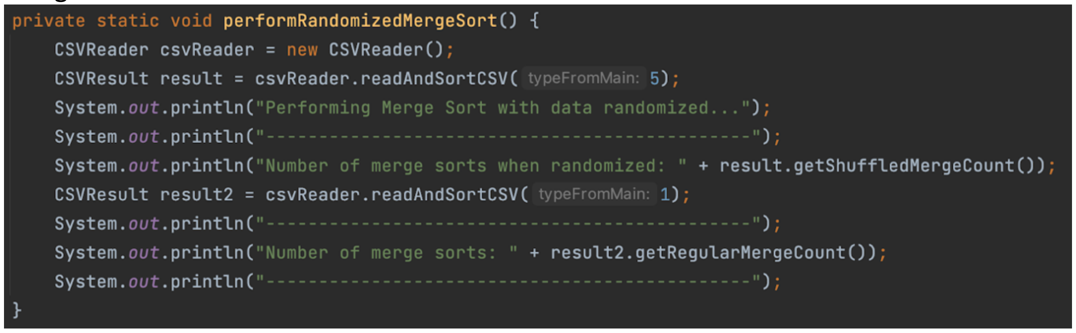
```java
private static void performQuickSort() {
    CSVReader csvReader = new CSVReader();
    CSVResult result = csvReader.readAndSortCSV( typeFromMain: 3);
    System.out.println("Performing Quick Sort...");
    System.out.println("Regular Quick sort:");
    result.getCities().forEach(city -> System.out.println("City: " + city.getName() + ", Latitude: " + city.getLatitude()));
    System.out.println("----------------------------------------------------------------------------------------------------------------");
    System.out.println("Number of quick sorts: " + result.getRegularQuickSortCount());
    System.out.println("----------------------------------------------------------------------------------------------------------------");
}
```

QuickSort based on latitude and longitude:



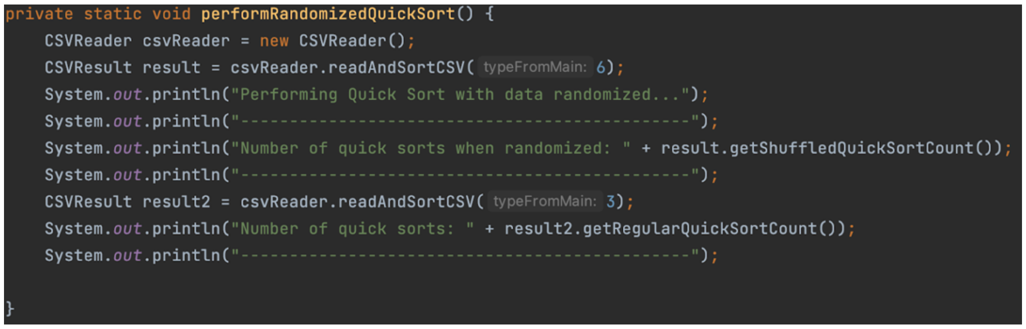
```java
private static void performQuickSortWithLongitudeAndLatitude() {
    CSVReader csvReader = new CSVReader();
    CSVResult result = csvReader.readAndSortCSV( typeFromMain: 4);
    System.out.println("Sorted cities by Longitude and Latitude:");
    result.getCities().forEach(city -> System.out.println("City: " + city.getName() + ", Latitude: " + city.getLatitude() + ", Longitude: " + city.getLongitude()));
    System.out.println("--------------------------------------------------------------------------------------------------------------");
    System.out.println("The List is ordered based on calculation of the distance between the longitudes and latitudes using the Haversine formula.");
    System.out.println("--------------------------------------------------------------------------------------------------------------");
}
```

Merge Sort with shuffled list



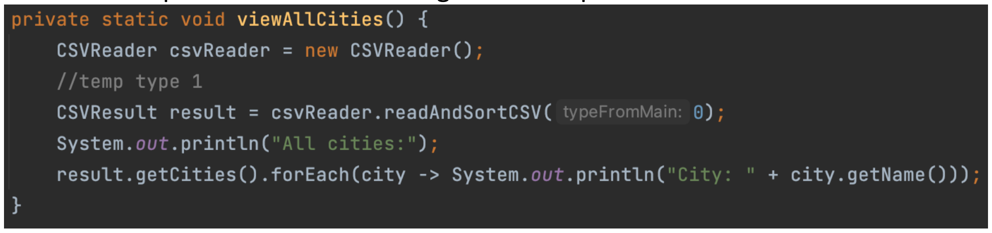
```java
private static void performRandomizedMergeSort() {
    CSVReader csvReader = new CSVReader();
    CSVResult result = csvReader.readAndSortCSV( typeFromMain: 5);
    System.out.println("Performing Merge Sort with data randomized...");
    System.out.println("---------------------------------------------");
    System.out.println("Number of merge sorts when randomized: " + result.getShuffledMergeCount());
    CSVResult result2 = csvReader.readAndSortCSV( typeFromMain: 1);
    System.out.println("---------------------------------------------");
    System.out.println("Number of merge sorts: " + result2.getRegularMergeCount());
    System.out.println("---------------------------------------------");
}
```

Quick Sort with shuffled list



```java
private static void performRandomizedQuickSort() {
    CSVReader csvReader = new CSVReader();
    CSVResult result = csvReader.readAndSortCSV( typeFromMain: 6);
    System.out.println("Performing Quick Sort with data randomized...");
    System.out.println("---------------------------------------------");
    System.out.println("Number of quick sorts when randomized: " + result.getShuffledQuickSortCount());
    System.out.println("---------------------------------------------");
    CSVResult result2 = csvReader.readAndSortCSV( typeFromMain: 3);
    System.out.println("Number of quick sorts: " + result2.getRegularQuickSortCount());
    System.out.println("---------------------------------------------");

}
```

Method that prints all the cities using lambda expression.



```java
private static void viewAllCities() {
    CSVReader csvReader = new CSVReader();
    //temp type 1
    CSVResult result = csvReader.readAndSortCSV( typeFromMain: 0);
    System.out.println("All cities:");
    result.getCities().forEach(city -> System.out.println("City: " + city.getName()));
}
```

Conclusion

In conclusion, the assignments focused on sorting cities' latitudes using Merge Sort and Quick Sort algorithms provided valuable insights into their respective efficiencies and behaviors.

For Merge Sort, the task involved sorting latitudes in ascending order, reflecting the geographical representation on the globe. The analysis revealed that regardless of the initial order of elements, the number of merges required remained constant. This stability underscores the algorithm's consistent performance, emphasizing its reliability in sorting datasets efficiently.

On the other hand, Quick Sort's implementation showcased its partitioning process and the pivotal role of shuffling in altering the number of comparisons and swaps. Unlike Merge Sort, Quick Sort's performance is influenced by the initial order of elements, leading to varying counts of operations when the list is shuffled. This sensitivity to input order highlights the dynamic nature of Quick Sort and the importance of understanding its behavior for optimal utilization.

In summary, while both Merge Sort and Quick Sort offer effective solutions for sorting tasks, their characteristics differ in terms of stability and sensitivity to input order. By comprehensively evaluating their performance and behavior, we gain valuable insights into selecting the most suitable algorithm for specific sorting requirements, ensuring efficient data processing and analysis.

References

[1]

Haversine formula to find distance between two points on a sphere (2022) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/haversine-formula-to-find-distance-between-two-points-on-a-sphere/ (Accessed: 30 April 2024).

[2]

Gupta, R. (no date) *LO 3_SortingAlgorithms_PG4200-1.pptx*, *LO 3 _ Sorting Algorithms* . Available at: https://kristiania.instructure.com/courses/11838/files/1330818?module_item_id=459016 (Accessed: 30 April 2024).