Lab 13: Concurrency

Learning Objectives

1. Understand how to write programs with Threads in Java
2. Practice using abstract classes and methods
3. Practice using real-world data in large datasets

Overview

Applications such as Google Maps use large amounts of data. Each segment of a road, water feature, or railroad is a record in a database containing coordinate (latitude/longitude) information, street name, segment type, and so on.

In this lab, we'll look at a simplified version of that data. We'll first design classes that allow us to perform queries on the data. Next we'll connect our queries to a graphical interface so you can visualize the data. Finally we'll add concurrency to our program, first allowing each query to be run in a separate thread and then allowing each query to be run in an arbitrary number of threads.

The Data

We have provided you with map data for two counties: Tippecanoe County, Indiana and Santa Clara County, California. We have also provided you with the classes *MapDataset* and *MapRecord*. These classes allow you to load the map data into memory and access it. **Note: You should <u>not</u> modify the *MapRecord* or *MapDataset* classes.**

You can download all of the Java files for this lab here.

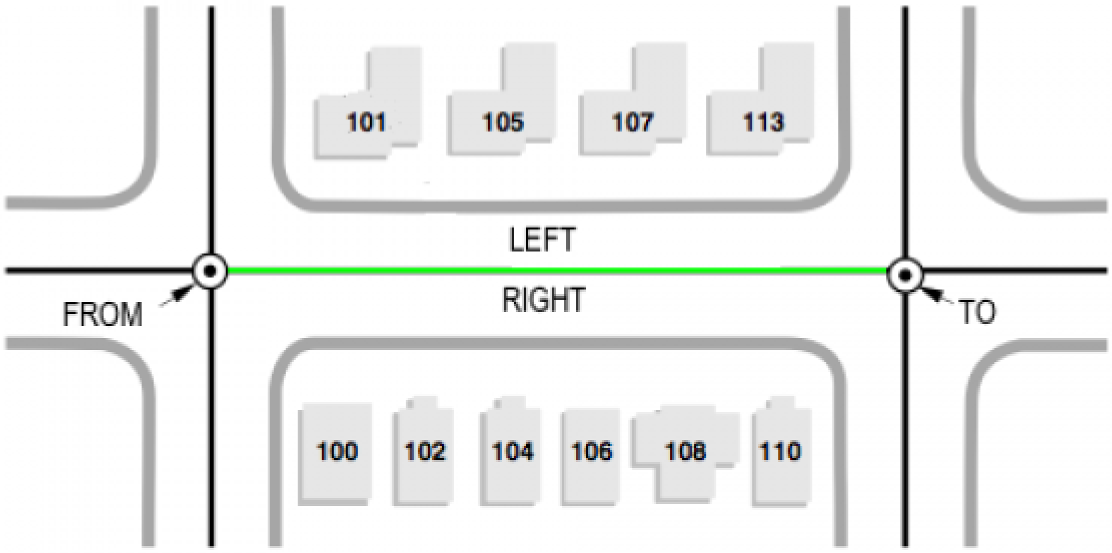
To load a map from data files into a map dataset in memory, construct a new *MapDataset*, passing in the location of the folder containing the data files for that county. For example (assuming that the data folders were contained in the “Lab13” folder in your home folder):

```
MapDataset tippecanoe = new MapDataset("~/Lab13/Tippecanoe");
```

A map dataset is a collection of map records whose order is undefined. As you might expect, a *MapDataset* object is a collection of *MapRecord* objects which describe road, water, or railroad segments on a map. Each map record defines a segment and has the following data:

```
public class MapRecord
{
  public double fromLongitude;
  public double fromLatitude;
  public double toLongitude;
  public double toLatitude;
  public String name;
  public String type;
  public String fromAddressLeft;
  public String toAddressLeft;
  public String fromAddressRight;
  public String toAddressRight;
  public String zipLeft;
  public String zipRight;
  public String tractLeft;
  public String tractRight;
  public String blockLeft;
  public String blockRight;
}
```

A sample map record for the green segment below might look like this:



```
fromLongitude: -87.0095248
fromLatitude: 40.562891
toLongitude: -87.030500
toLatitude: 40.562822
name: W Elm St
type: A31
fromAddressLeft: 101
toAddressLeft: 113
fromAddressRight: 100
toAddressRight: 110
zipLeft: 49706
zipRight: 49710
tractLeft: 001800
tractRight: 001800
blockLeft: 1002
blockRight: 1003
```

- `fromLongitude`, `fromLatitude`, `toLongitude`, and `toLatitude` are the coordinates of the endpoints of the segment in the real world.
- name is the complete street name of the segment.
- `type` is a code that describes the segment (*x* is a digit between 1 and 9):
  - A*xx* - Road
    - A00 - Road with category unknown
    - A1*x* - Primary road with limited access
    - A2*x* - Primary road without limited access
    - A3*x* - Secondary and connecting road
    - A4*x* - Local, neighborhood, and rural road

    - A5*x* - Vehicular trail
    - A6*x* - Road with special characteristics
    - A7*x* - Road as other thoroughfare
  - B*xx* - Railroad
  - C*xx* - Miscellaneous Ground Transportation
  - D*xx* - Landmark
  - E*xx* - Physical Feature
  - F*xx* - Nonvisible Feature (such as political boundaries)
  - H*xx* - Hydrography (water features)
- `fromAddressLeft` and `toAddressLeft` are the address range on the left side of the street
- `fromAddressRight` and `toAddressRight` are the address range on the right side of the street
- `zipLeft` is the zip code on the left side of the street
- `zipRight` is the zip code on the right side of the street
- `tractLeft` is the census tract number (a grouping used in the U.S. Census) on the left side of the street
- `tractRight` is the census tract number on the right side of the street
- `blockLeft` is the census block number (a grouping used in the U.S. Census) on the left side of the street
- `blockRight` is the census block number on the right side of the street

Note that the map dataset only contains the data in a map record. The width or color of the road and the buildings on the street are not part of the map dataset.

All of this data is taken from the U.S. Census Bureau TIGER/Line Files and its documentation. The data is free to download and use. If you're interested, you can find more detail about these files at http://www.census.gov/geo/maps-data/

Once you construct a new *MapDataset* object, you can get the number of records in the map dataset with the *getNumRecords* method, and you can get a specific map record with the *getRecord* method. Records are numbered from 0 to `getNumRecords() - 1`.

Here's an example that loads map datasets for Tippecanoe County and Santa Clara County and prints out all of the map records in each dataset that are within a certain area:

```
MapDataset tippecanoe = new MapDataset("Tippecanoe");
MapDataset santaclara = new MapDataset("SantaClara");

for (int i = 0 ; i < tippecanoe.getNumRecords() ; i++)
{
  MapRecord record = tippecanoe.getRecord(i);

  if (record.isIn(MapBounds.TIPPECANOE))
    record.print();
}

for (int i = 0 ; i < santaclara.getNumRecords() ; i++)
{
  MapRecord record = santaclara.getRecord(i);

  if (record.isIn(MapBounds.SANTACLARA))
    record.print();
}
```

When you run this, you should get:

```
Loading Nodes: 10174 nodes
Loading Streets: 2408 streets
Loading Edges: 14449 edges
Loading Nodes: 55000 nodes
Loading Streets: 17640 streets
Loading Edges: 76721 edges
--------------------------------------------------------------------------
----
Coordinates: (-87.075866,40.562895) --> (-87.076406,40.56294)
Name: W County Line Rd N
Type: A41
Address range left: 9001-9027
Address range right: -
Zip / Tract / Block (left): 47906 / 010201 / 3006
Zip / Tract / Block (right):  /  /

--------------------------------------------------------------------------
----
Coordinates: (-87.057203,40.562849) --> (-87.075866,40.562895)
Name: W County Line Rd N
Type: A41
Address range left: 8011-8999
Address range right: -
Zip / Tract / Block (left): 47906 / 010201 / 3005
Zip / Tract / Block (right):  /  /

and so on...
```

What to Do

Step 1 - Printing all the records in the datasets

When we query a map dataset, we want to go through all of the map records in the dataset and for each record that meets some criteria, we want to do something with it (such as print out its information or display it in a window).

Let's start by reading in map datasets for Tippecanoe and Santa Clara counties, and outputting all of the map records for each dataset to the console.

To do this, compile and run the code that we have provided. It does the following::

1. Creates two *MapDataset* objects, one for Tippecanoe County and one for Santa Clara County.
2. Loops through all of the map records in each dataset, printing them out using the *print* method of the *MapRecord* class as long as the record is in the appropriate area.

Once you've run the program and are convinced that it works, stop your program from running as there are over 90,000 map records in both counties combined!

Step 2 - Refactoring the loop

Note that you probably wrote almost the same code for both loops. Let's refactor the code to simplify this. In doing so, we'll make more concrete the idea of a query; that is, going through all of the map records in a dataset and performing some action on records that meet a certain criterion. In this simple case, we just want all of records that are within the area we specify.

First, we'll make a *MapAreaQuery* class (**We did this for you!**):

1. Create a *MapAreaQuery* class.
2. Add a constructor for the *MapAreaQuery* class that is passed a *MapDataset* object and a *MapBounds* object and saves them in instance variables.
3. Add a *run* method that takes no arguments and loops through all of the map records in the dataset, printing out each record that is within the given map bounds. Print them out using the *print* method of the *MapRecord* class.

Second, modify your *main* method (**You still need to do this!**):

1. After the code that creates the two map datasets, add code to create two *MapAreaQuery* objects, one for each dataset.
2. Remove the code that loops through the records in each dataset.
3. Call the *run* method for each *MapAreaQuery* object.

Test your code. It should work exactly as in step 1.

Step 3 - Displaying the data in a window

Printing out the records to the console is somewhat boring. Let's see if we can display the map data in a window. Use the *MapWindow* class to display map records in a given color. Here's how…

Note: You should not modify the *MapWindow* class.

First, modify your *MapAreaQuery* class:

1. Import `java.awt.Color` at the start of the class file.
2. Modify the constructor to add a *MapWindow* parameter and save this parameter in an instance variable.
3. Modify the *run* method as follows:
   - Set the geographic boundaries of the window:window.setMapArea(bounds); where *bounds* is the boundary (min and max longitude and latitude) of the geographic area;
   - Display each map record in the map window in black rather than print it: record.display(window, Color.black);
   - Redraw the map window after displaying all of the records:window.redraw();

Of course, use your variable names instead of *record*, *bounds*, and *window*.

Second, modify your *main* method:

1. Create two new *MapWindow* objects – one will be for displaying the Tippecanoe data and one for the Santa Clara data. The parameters are the (*x*,*y*) location of the upper-left corner of the window on the screen, the *width* and *height* of the window, and a boolean signifying whether you want the

window to redraw after displaying each record or wait for an explicit redraw command. A typical window might be created as follows:MapWindow window = new MapWindow(100, 100, 640, 480, false); Be sure to offset the upper-left corners of the windows so that they don't overlap completely.

2. Change the calls creating the *MapAreaQuery* objects to add the appropriate map window and map bounds as additional arguments. Map bounds you can use are *MapBounds.TIPPECANOE* (for the geographic boundary for Tippecanoe County), *MapBounds.SANTACLARA* (for the geographic boundary of Santa Clara County), *MapBounds.PURDUE* (for the area around Purdue), and *MapBounds.APPLE* (for the area around Apple's main campus in Silicon Valley).

Step 4 - Making another query (optional)

This step is optional. It's interesting, but not essential to completing the lab.

Let's make a new query that only displays records that are part of a census tract boundary (remember that a census tract is an area used by the U.S. Census Bureau for aggregating and reporting population statistics).

To do this:

1. Create a *MapTractBoundaryQuery* class.
2. Add a constructor that takes a *MapDataset* object, a *MapBounds* object, and a *MapWindow* object and saves them in instance variables:
3. Add a *run* method that:
   - sets the geographic bounds of the map window;
   - loops through all of the records and displays only records within the given map bounds where the left tract code is different from the right tract code and is therefore a census tract boundary (remember that the tract codes are both *String*s); and finally
   - redraws the window.

Test your code by modifying the *main* method appropriately to create and run some sample queries.

Step 5 - Abstracting the code common to both queries

We still seem to duplicating code in each query to loop through all of the records in the dataset. Let's see if we can abstract that out by putting the loop in a *MapQuery* class that is a superclass for the various map query classes. Each individual map query will now only process one record that is passed to it, and the *MapQuery* class will be responsible for looping through all of the records within the given map bounds and passing each record for processing to a subclass' method.

Note that since we can't use *run* as the name of the method for the processing each individual record in the subclass (why not?), let's change the method name in the subclasses from *run* to *process*. This also more accurately describes what we're doing to each record. Note too that we can now store the map dataset and map bounds objects in instance variables in the *MapQuery* class rather than in each subclass. (Why would we want to do this?)

But we're not quite done. Our original *run* method in the map query classes did more than just loop through the records. It also set the map area of the window before the loop and redrew the window after the loop. Let's abstract out that preprocessing and postprocessing functionality into *preprocess* and *postprocess* methods that are called in our *MapQuery* class but defined in our map query subclasses. We'll also add some timing using the *Stopwatch* class so that we can see how long our queries take. **Note: You should <u>not</u> modify the *Stopwatch* class.**

So…

First:

1. Create a *MapQuery* class. This class should be abstract because we'll never create a *MapQuery* object directly (why not?).
2. Add a constructor for the *MapQuery* class that is passed a *MapDataset* object and a *MapBounds* object and saves them in instance variables. The constructor should also create a *Stopwatch* object in an instance variable (the *Stopwatch* class is provided to you).
3. Add an abstract *process* method that is passed a *MapRecord* and doesn't return anything. Note that we will not define the *process* method in the *MapQuery* class, but by declaring it as *abstract*, we will force subclasses of the *MapQuery* class to define it.
4. Add abstract *preprocess* and *postprocess* methods that take no arguments and don't return anything. Note that we will not define these methods in the *MapQuery* class, but by declaring them as *abstract*, we will force subclasses of the *MapQuery* class to define them.
5. Add a *run* method that first calls the *preprocess* method. It then loops through all of the records in the dataset calling *process* passing it each record in turn. Finally it calls the *postprocess* method. It should also start the *Stopwatch* object before the loop and stop it after the loop.
6. Add an *getElapsedTime* method that returns the elapsed time of the *Stopwatch* instance variable.

Second, in each of your map query classes:

1. Change it to be a subclass of the *MapQuery* class.
2. Remove the instance variables that hold the map dataset and map bounds objects (remember, we now store them in the *MapQuery* class).
3. Change the constructor to call the superclass' constructor and pass it the dataset.
4. Remove the code from the *run* method that sets the map area for the window and move it to a new *preprocess* method.
5. Remove the code from the *run* method that redraws the window and move it to a new *postprocess* method.
6. Rename the *run* method to *process* with a *Record* as an argument.
7. Remove the loop code from the *process* method and process only one record (the one passed in to the *process* method) at a time.

Finally:

1. Add code to your *main* method to get and print the elapsed time for each query.

Test your code. Note the elapsed time for each query.

Step 6 - Make the queries run in their own threads

So far, so good. Note however, that each query starts running only after the previous query finishes. It would be better if we could run multiple queries simultaneously. Here's where Java's threads make the difference. Since our *MapQuery* class already has a *run* method, let's make the *MapQuery* class a subclass of the *Thread* class. Then in our *main* method we just need to call the *start* method for our queries rather than the *run* method and each query will run in its own thread!

Go ahead and do that now. Don't forget to test your code.

Step 7 - Letting each query run in multiple threads

At this point, we have each map query running in its own thread. But it would be nice if we could have a single query run in multiple threads. To do this, we'll define a new class, *MapQueryThread*, that is responsible for processing a given subset of map dataset for a map query. We'll then modify our *Query* class to set up the *MapQueryThread* objects, start them running, and wait for them to finish. Finally, we'll need to slightly modify our various map queries so that we can pass them the number of threads we want them to use when running.

First, modify the *MapQuery* class:

1. Add an instance variable to the *MapQuery* class that holds an array of *Thread* objects.
2. Add a *getDataset* method to the *MapQuery* class that returns the dataset for the query.
3. Modify the *MapQuery* constructor to add the following code:this.threads = new

```
MapQueryThread[numThreads];

int threadSize = dataset.getNumRecords() / numThreads;

int begIndex = 0;
int endIndex = threadSize - 1;
int threadIndex = 0;

while (threadIndex < numThreads)
{
  threads[threadIndex++] = new MapQueryThread(this, begIndex, endIndex);

  begIndex = begIndex + threadSize;
  endIndex = endIndex + threadSize;
```

```
    if (endIndex > dataset.getNumRecords() - 1)
      endIndex = dataset.getNumRecords() - 1;
  }
```

4. In the *run* method, instead of looping through the records and processing each record in turn:
   - Loop through all of the *MapQueryThread* objects you created in the constructor and call their *start* method.
   - Then loop through all of the *MapQueryThread* objects and wait for them to finish by calling their *join* method. Note that you'll need to use a *try-catch* block to catch a possible *InterruptedException*.

Second, create a new *MapQueryThread* class that is a subclass of the *Thread* class:

1. Create instance variables for the *MapQuery* it is a thread for and the beginning and ending index of the dataset.
2. Add a constructor that is passed a *MapQuery* object and a beginning and ending index and stores them in the appropriate instance variables.
3. Add a *run* method that loops through the beginning index to the ending index (inclusive) and calls the *process* method of the *MapQuery* and passes it the appropriate record of the dataset of the query.

Third, modify your various map queries so that their constructor is passed a *numThreads* parameter and calls the superclass' constructor with that parameter.

Finally, modify your *main* program to call the queries with different numbers of threads.

Test your code. Your code most likely generated an exception. The problem is that multiple threads may be trying to access the map window object simultaneously. To fix this, any time you want to access a shared object, put it in a *synchronized* block. For example,

```
synchronized( <shared object> )
{
  // code using the <shared object>
}
```

Remember that synchronization only works with objects. If you're using a counter in one of your queries, you'll need to make it an *Integer* object rather than an *int* in order to synchronize it.

Test your code again. Are queries faster if they use more threads? Why or why not?

Step 8 - One last query (optional)

This step is optional. If you have time, you may want to do it, but you do not need to.

So far, we've only been drawing our maps in black. Change your *MapAreaQuery* to display the segments in different colors based on their type. For example, you might display water features in blue, major highways in red, minor roads in gray, and railroads in black. Or be creative and come up with your own color scheme.

Finally, make one other interesting query and test it; the more creative, the better!

If you're really stumped, make a query that takes a geographic coordinate (long/lat) and a radius (in miles) and displays all of the segments in a circle centered on the given geographic coordinate with the given radius. You can use the static *MapWindow* function *geographicDistance* to get the distance (in miles) between two geographic coordinates:

```
double distance = MapWindow.geographicDistance(longitude1, latitude1,
longitude2, latitude2);
```

Grading

50 points plus 10 points for each step (2, 3, 5, 6, and 7) you complete up to a total of 100 points.

Turn-in

To turn-in your .java file(s), move to your cs180 directory using the following two commands:

```
$ cd ~
$ cd cs180
$ cd lab13
```

Then run the following turnin command AS IS:

```
turnin -v -c cs180=COMMON -p lab12 *.java
```

IMPORTANT: make sure to turn-in your .java files immediately after you finish each of the steps 2, 3, 5, 6, and 7.