Development Log

Academic week #7 (17th to 23rd Nov)

During the practical Friday lesson, I researched more about tests and how to organize them as it will be when having hundreds of tests for one program. This is because each Junit testing of a methods needs multiple tests so one method per test method is too compact and one separate class file per test is too wide to be practical.

Did eventually come across test categories Java annotations but the only valid ones are class names which defeat the point of them. Source - https://junit.org/junit4/javadoc/4.12/org/junit/experimental/categories/Categories.html

Did also come across another JUnit Java annotation which were much more useful for organizing huge amounts of tests:

- Tag annotation for distinguishing types of tests - https://www.baeldung.com/junit-filtering-tests
- Nested class annotation - https://www.baeldung.com/junit-5-nested-test-classes

Due to major indigestion and nausea problems, I couldn't do anything else.

Academic week #8 (24th to 30th Nov)

Due to my 5-month stomach I had to talk to student engagement for an additional extension to the first COMP7007 assignment.

Was a bit confused when mark scheme said 10% was for this log doc; I thought David said it didn't count towards marks. Should prob ask him abt that at some point.

Started this development log.

Project requirements

Copying from my lecture notes, this assignment should be mainly abt:

- Web APIs
- JUnit
- Records
- Lambdas (including streams)

Assignment confirms this but with additional things:

- Development log
- Retrieval and decoding of data
- Interactive functionality
- README file describing how to use the interactive front end

I Believe “Retrieval and decoding of data” means using JSONs well, but unsure what “Interactive functionality” means. Is simple CLI not enough? Does it need to be advanced enough to require a README UI manual file?

“*Note that you are not required a complete set of JUnit tests. You just need to demonstrate that you can write effective JUnit tests. Ten tests would be sufficient, for instance. You are not required to write tests that fail (that is, ones that evidence errors in the program).*” ← oh thank you David! :D

“*text-based interactive*” ← What console requires a manual? What if I print the manual in the console instead?

Project plan – list

Okay, so far, this what I think the program should do:

1. Print what the programs do and how to interface with it.
2. Ask and take in for which the program wants to fetch weather data from; those being:
   - specify date ranges
   - magnitudes of interest
   - locations of interest for the data to be retrieved
3. Fetch the data (from https://earthquake.usgs.gov/fdsnws/event/1/query?format[...]) but ensure it returns JSON, not XML – make program expect JSONs only as per assignment specification.
4. To demonstrate use of Records, format the wanted JSON data into a Record. Use nested records if data structure is a bit too complex... actually doing nested records will show a greater understanding of Records.
5. Do local processing, even if API may provide it already, ensure some local processing happens to get marks.
6. Print the results of the processing such as area of most affected, averages, comparisons, etc. “*possibly multiple times*” indicates that the more the better.
7. Make sure user knows what user can run diagnostic tests (JUnit testing) to ensure program works fine.
8. Ask user if user wants to query (start from step #2) again to allow multiple queries to be made.

Lambdas and Streams must be used in step #4 and #5.

Records are quite simplistic in nature, so I will have to use it fully but without violating Single Purpose, Loose Coupling, or whatever it is called.

Because most likely things will not go according to plan, divide and conquering with single purpose classes is best as it allows great adaptability. Such is how the program will be planned. The plan:

- Class for taking in preferences, fetching JSON, and formatting wanted data into a Record using Lambdas and Streams. Also include validation of presences and use extensive error-prevention code when interacting with the web API. Class should be singleton pattern as it is generalist and only one is needed for the whole program (unless concurrency which I will not use).
- Class for using Lambdas and Streams to locally process the query-response Records. Again, singleton class.
- Class with Static main to start the program. Ig one would call this a command pattern class.
- Interpreter and composite pattern class to handle CLI and interface it to the other classes.

- JUnit testing class for testing the fetching JSON class as a lot of things will go wrong with that – both mistakes and external errors. Each method tested will have a dedicated nested class, and each method of those nested classes will test a specific aspect of the tested method.

This plan assumes that it is a good idea to include concepts from the first half of COMP7007 too.

This plan is just the baseline draft, polishing and additional functionality may be added depending on how early I finish the product – so how my stomach fares.

In respect to the plan I think the classes should be named (respectively):

- RecordFetcher
- RecordProcessor
- TestingRecordFetcher
- EarthquakeInterpreter
- Main

Names are convenient to me but may change them, near the end, to fit the assignment's views of good OOP.

Project plan – UML

Based on these descriptions I will make a UML **but** a very quick and simple one as I don't think UML counts as marks here.

I don't think this program is complex enough to make use of inheritance, abstract classes, interfaces, etc. Hope this does not deduct marks.

Searching for a UML diagram maker that can generate skeleton (empty) Java classes and methods from it. Found this - https://www.jetbrains.com/guide/java/tips/generate-uml-diagram/ .

IDEA IDE's built-in UML plugin works but the creational tools always freeze the IDE so I must make the skeleton classes themselves first then reflect them in the UML.

Spent the remainder 3 days finishing the A1 work then proceeded to have a 2-day straight severe headache.

Academic week #9 (30th Nov to 6th Dec)

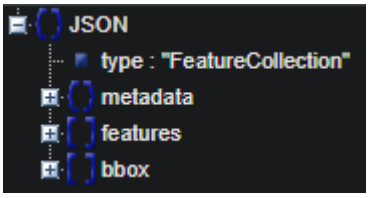
When making the skeleton classes I made some new additions:

- 'ClientActions' interface to simplify dictation of what user can and cannot do – easily keep track of.
- The query report record will have a nested record of the API retrieved data while the report record will contain the local processing in its constructor.
- An enumeration for all query params – easily keeps track of.
- Query args validation will be delegated to a leaf class – to prevent 'RecordRetriever' class with so many methods that it looks like a "god class".
- The sheer amount of query args makes it more suitable to pass them as a EnumMap than individually.

Analyzing JSON structure

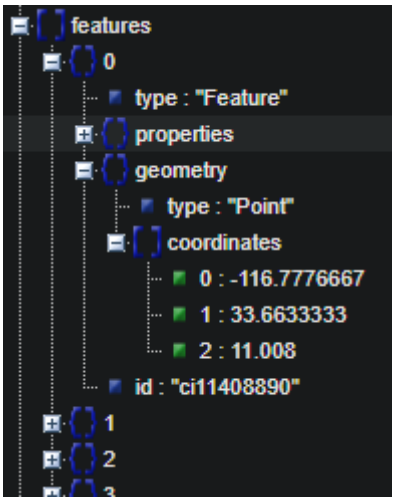
Used the sample query
“ https://earthquake.usgs.gov/fdsnws/event/1/query?format=geojson&starttime=2014-01-01&endtime=2014-01-02 ” to see json structure.

uppermost structure:



“metadata” can be good to confirm API version and the HTTP status code values, but “features” is the main thing we want.

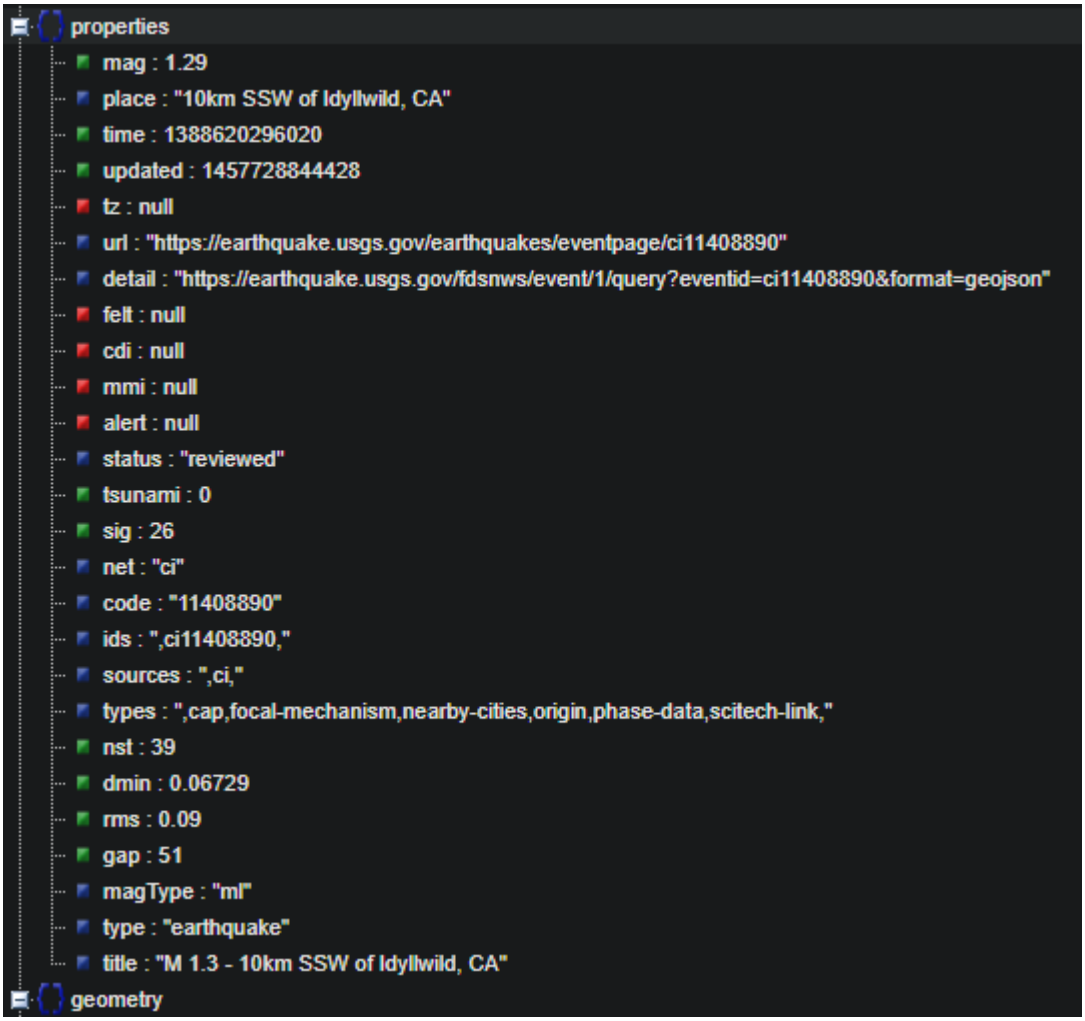
Each Earthquake entry is in Features labelled as indexes:



Not sure what they mean by “feature”.

Geometry object is simply about location, and its data structure is nicely small.

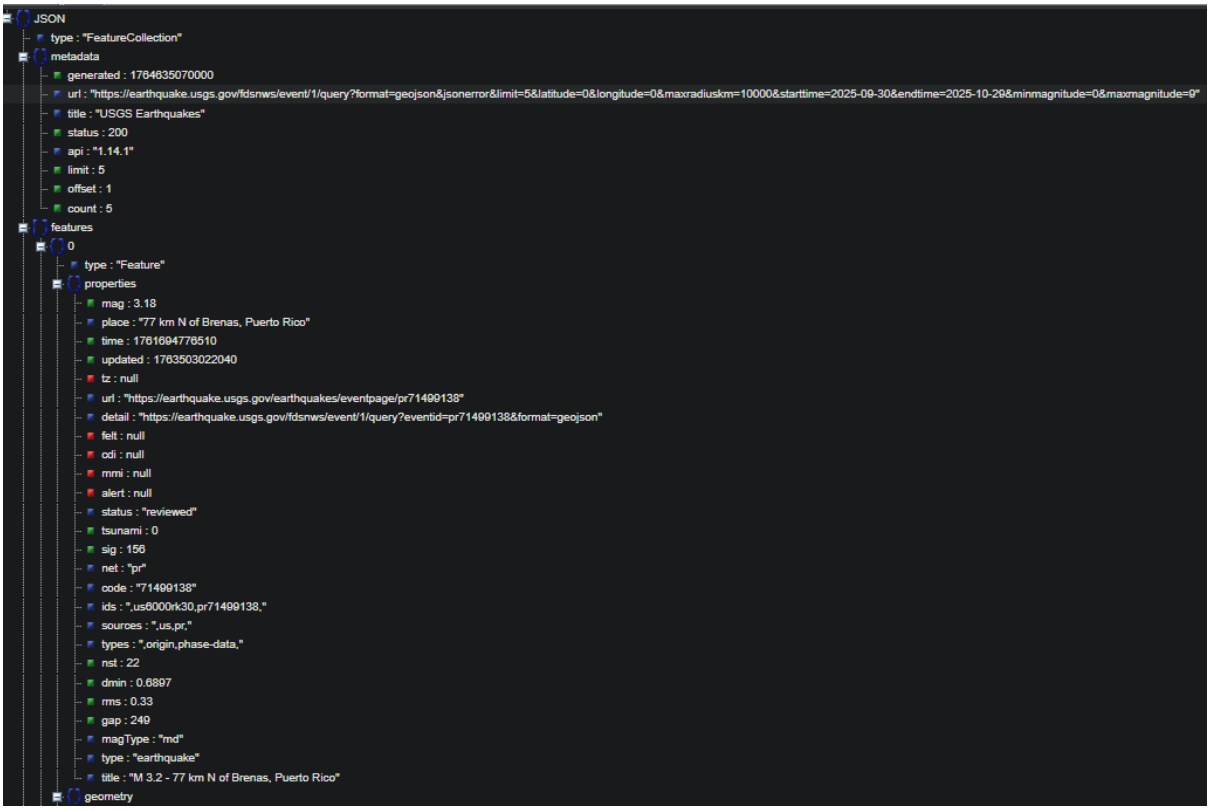
Properties on the other hand consists on more content and having confusion abbreviations:



Going to put URL of what the program would send (based on what I put in 'QueryParam' enum and example values):

https://earthquake.usgs.gov/fdsnws/event/1/query?format=geojson&jsonerror&limit=5&latitude=0&longitude=0&maxradiuskm=10000&starttime=2025-09-30&endtime=2025-10-29&minmagnitude=0&maxmagnitude=9

API request surprisingly worked first time:



Features has numbers 0-4 so they are indeed indexes.

Structure of “properties” appears near or is identical. Referred to “ https://earthquake.usgs.gov/data/comcat/index.php ” for info about the abbreviations, and these have varying levels of relevance from exactly what we want to wondering why they are there in the first place.

Here is a list of what we want in every “features” entries:

- “mag” – magnitude’s magnitude
- “magType” – scale of magnitude used. In the JSON, seems different scales are used for different entries for some reason.
- “place” – human readable information – will be displayed in raw data but not processed on for obvious reasons.
- “time” – original time of earthquake (not to be confused with “updated”)
- “coordinates” – structured as [longitude, latitude, depth]

Other useful entries:

- “api” – check version to warn user of any unexpected behaviour if API underwent a major update recently.
- “status” – check if the API query was successful (code 200) or not.

From this more additions was made to the skeleton classes project:

- Enumeration for the wanted “features” entries
- Said enumeration will be used in an enumeration dictionary array list to store each earthquake instance.
- Earthquake data records will contain the enumeration dictionary array list alongside the methods that fetches data sets from it, such as:

    - Returning array of LocalTimeDate instances of every earthquake entry
    - Returning array of magnitudes (as doubles) but not before converting them all into being of the same earthquake magnitude scale.
- Earthquake type should also be its own enums… so many enums

To simplify the Earthquake status enum record I will:

- Split up ‘COORDINATES’ to longitude, latitude, and depth so I use 3 doubles instead of another enum dict
- Convert magnitude scale in every entry into a common one so I do not need to store the magnitude type.

Now I don’t think a enum for magnitude types is needed as I only find enums worthwhile if both the enum constant names and their string representations are exclusively used.

Looked at https://www.usgs.gov/programs/earthquake-hazards/magnitude-types

Magnitude Type	gnitude Range	Distance Range	Equation	Comments
Mww (Moment W-phase) (generic notation Mw)	~5.0 and larger	1 - 90 degrees	$M_W = 2/3 * (\log_{10}(M_0) - 16.1)$, where M_0 is the seismic moment. Note this is also unit-dependent; the formula above is for moment in dyne-cm. If using metric units (N.m), the constant is 9.1.	Derived from a centroid moment tensor inversion of the W-phase (~50-2000 s; pass band based on size of EQ). Computed for all M5.0 or larger earthquakes worldwide, but generally robust for all M5.5 worldwide. Provides consistent results to M~4.5 within a regional network of high-quality broadband stations. Authoritative USGS magnitude if computed.
Mwc (centroid)	~5.5 and larger	20 - 180 degrees	$M_W = 2/3 * (\log_{10}(M_0) - 16.1)$, where M_0 is the seismic moment.	Derived from a centroid moment tensor inversion of the long-period surface waves (~100-2000 s; pass band based on size of EQ). Generally computable for all M6.0 worldwide using primarily the Global Seismograph Network. Only authoritative if Mww is not computed, not published otherwise.
Mwb (body wave)	~5.5 to ~7.0	30 - 90 degrees	$M_W = 2/3 * (\log_{10}(M_0) - 16.1)$, where M_0 is the seismic moment.	Derived from moment tensor inversion of long-period (~20-200 s; pass band based on size of EQ) body-waves (P- and SH). Generally computable for all M5.5 or larger events worldwide. Source complexity at larger magnitudes (~M7.5 or greater) generally limits applicability. Only authoritative if Mww and Mwc are not computed.
Mwr (regional)	~4.0 to ~6.5	0 - 10 degrees	$M_W = 2/3 * (\log_{10}(M_0) - 16.1)$, where M_0 is the seismic moment.	Based on the scalar seismic-moment of the earthquake, derived from moment tensor inversion of the whole seismogram at regional distances (~10-100 s; pass band based on size of EQ). Source complexity and dimensions at larger magnitudes (~M7.0 or greater) generally limits applicability. Authoritative for <M5.0. Within the continental US and south-central Alaska where we have a large number of high quality broadband stations we expect we can compute an Mwr consistently for events as small as M4.0. In some areas of the country, with relatively dense broadband coverage, we can compute Mwr consistently to as small as M3.5.
Ms20 or Ms (20sec surface wave)	~5.0 to ~8.5	20 - 160 degrees	$M_S = \log_{10}(A/T) + 1.66 \log_{10}(D) + 3.30$.	A magnitude based on the amplitude of Rayleigh surface waves measured at a period near 20 sec. Waveforms are shaped to the WWSSN LP response. Reported by NEIC, but rarely used as authoritative, since at these magnitudes there is almost always an Mw available. Ms is primarily valuable for large (>6), shallow events, providing secondary confirmation on their size. Ms_20 tends to saturate at about M8.3 or larger.

^ barely understandable table format

Seems like the scales used are: mww, mwc, mwb, mwr, ms20/ms, mb, mfa, ml, mb_lg, md, mi/mwp, me, mh, finite fault, and mint (‘/’ means either name). Purposely made them all lowercase as it will be easier to match magnitude types with one of the letter cases.

According to to further research:

- Mw (Moment Magnitude) is the closest thing to a universal magnitude type/scale “practically”.
- mww, mwc, mwb, mwr, and mwp are all Mw, just calculated using different methods, so no need to convert them.
- Everything else requires more data, than provided, to convert – using seismic moment, joules, log(10) math operations, etc.

I think… I will just filter magnitude scales for now and maybe go back to later if I have extra time.

API does have a filter arg “&magtype=” but it can only filter one per query so its better to get all and filter locally to reduce the number of API requests – minimise chances of external errors.

So apparently Java methods do not support default arguments.

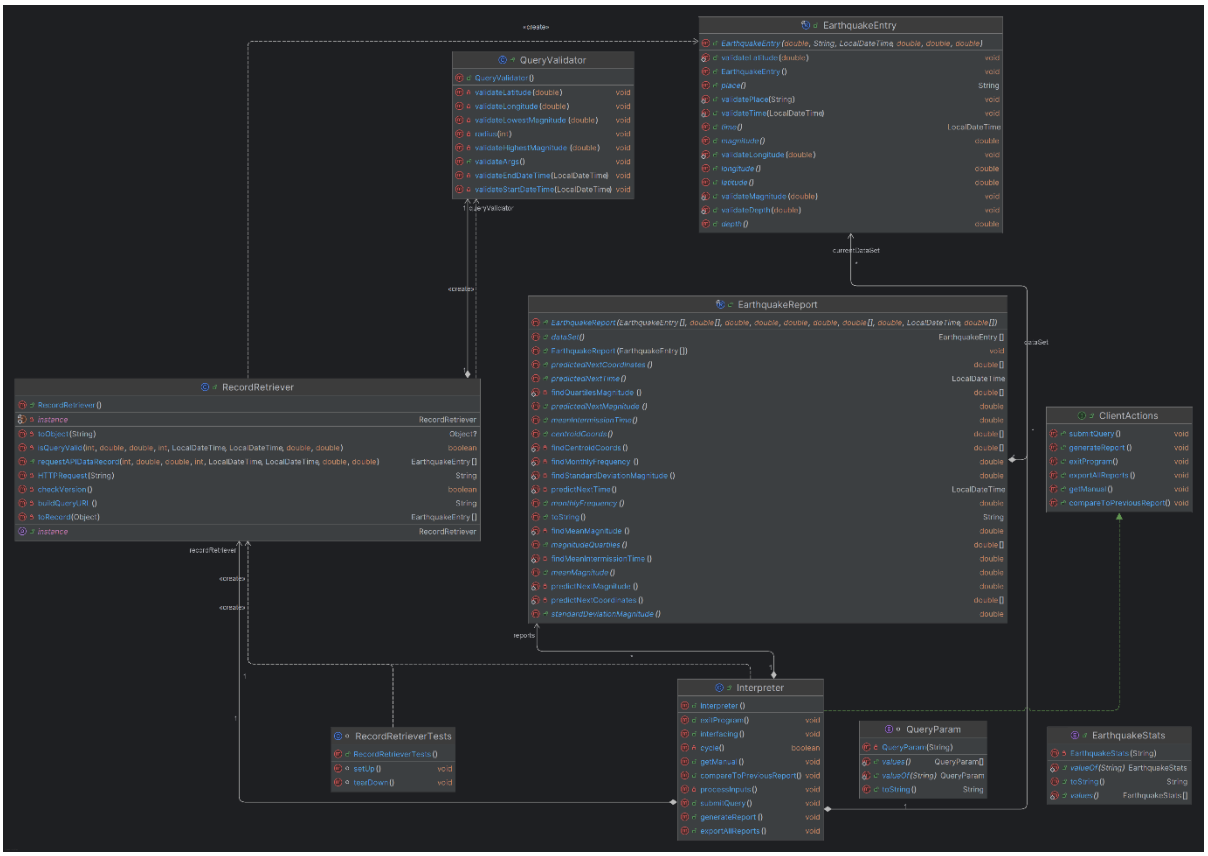
So apparently end-of-year mitigation does in fact not extend deadline to end of year thus I will need to speed up because I have either 1 or 2 weeks left.

I have forgetting that the data of each entry have different data type so a record in needed over a enum map.

Per entry record makes the data set record useless as the array of entry records can be stored directly in the report record.

This also makes Earthquake stats useless but will be kept incase for later use.

UML diagram so far:



^ Obviously due to only using skeleton classes, relationships are not complete.

Libraries

To complete the UML, had to import the necessary dependencies.

Idk why, but there is so many variations of the date/time built-in libraries. Decided to use ‘java.time.LocalDateTime’ but that may change later.

David did mention to use an external library for JSONs. Asked some experienced classmates for which one is the best and they suggested “Gson” (https://github.com/google/gson). Seems to do at least the basics of “convert a JSON string to an equivalent Java object” so I installed it using Marven.

Also installed Junit for obvious reasons.

Implementing code

After realising the severe short time left, I decided to start implementing the code straight away.

There is no absolute limits for magnitude and depth so gotta research about that.

Used static methods and compact canonical constructor for the record component validation.

Earthquake entry record has been written up.

To make lambda streams easier to make and recognise, as well as demonstrating advanced understanding, will make them multiline with comments (instead of single-line).

Finding means are easy but quartiles are much more difficult to do so in a compact matter.

Compact code examples makes use of "[true or false statement] ? [code for true] : [code for false]". **This is a very useful way, and I think I remember that being demonstrated in one of the lectures.** Will use it.

So apparently method references (like "EatthquakeEntry::latitude") are not lambdas (like EatthquakeEntry -> EatthquakeEntry.latitude()). Apparently to the internet, method references are not lambdas and the assignment brief only shows lambdas for marks not method references. **Does that mean using the better-practice method references, instead of lambdas, makes me lose marks?!**

^ That realisation causes me to decide to focus more on stream use, than lambda, until I get an answer to this paradox!

Disabled "EarthquakeReport.predictNextMagnitude" and "EarthquakeReport.predictNextCoordinates" as their predicted values are, simply put, the means that are already calculated/processed. They may be reactivated if have time to code a better way to predict rather than just means. Prediction next time is not mean so will still use that.

Swapped 'java.time.LocalDateTime' with 'java.time.ZonedDateTime' to ensure that all date/times are in UTC time zone. Declaring the UTC time zone also makes me import 'java.time.ZoneId'.

Similar story with magnitude standard deviation, will do at the end if have spare time.

Forgot to add/override toString method to the 'EarthquakeEntry'. Just implemented that using 'String.format' to keep scope code clean (good practice).

Due to different measurements having differing accuracy, they are reflected by the number of decimal places permitted/rounded:

Measurement(s)	**Precision**
Latitude and longitude	3 decimal places
magnitude	1 decimal place
depth	2 decimal places
time of earthquake	Seconds (integer)
Earthquake time prediction	Hours (1 d.p.)
Earthquake mean intermission time	Days (2 d.p.)

'EarthquakeEntry.toString' overridden.

Back to 'EarthquakeReport'.

Seems canonical constructor demands the '.this(' must be called first in this scope. This makes code inefficient as static methods that requires results in duplicate method calls. Search for a solution which seems to use a static factory method as explained in https://www.baeldung.com/java-constructors-vs-static-factory-methods . After reading I fully understand what it exactly is and how to use it as an alternative to the canonical constructor. **The only part I am confused is the "of" syntax in the static factory method declaration – I know how to use it but not sure why its needed.** Sources says the syntax is optional, but IDEA ide says otherwise. My guess is that it is another one of those good practices that IDEA ide enforces; this is so because this is not the first time IDEA ide forbids me certain codes that were proven to have nothing wrong with it.

Completely forgotten about depth. Since it's a relatively simple measurement, and is not as important as the other statistics (such as magnitude), only the mean depth is needed.

New client action (interface method) added – view raw data set – in case user wants to see it again.

Second commit-and-push has been done

Consulted sources and determined that for geographic coords, it is ordered as latitude then longitude. Sources:

- https://www.movable-type.co.uk/scripts/latlong.html
- https://simonwrigley.medium.com/lat-lon-or-lon-lat-8adaf6441ccd

Confused why USGS permits negatives depths but https://www.usgs.gov/faqs/what-does-it-mean-earthquake-occurred-a-depth-0-km-how-can-earthquake-have-a-negative-depth explains that this is because of margin of errors. This page also says that 0km depth does not mean near non-existent earthquake (as I initially thought) but instead means unable to determine depth of an Earthquake instance. To reflect this in the assignment program, all earthquake entries, with a non-positive (including zero), are rejected to ensure local processing reliability.

For upper limit, depth is invalid if exceeding 800km (200 less than ASGS api acceptance) because the deepest recorded depth was ~735.8 km. Source - https://en.wikipedia.org/wiki/Deep-focus_earthquake .

Some utility methods are added to 'QueryValidator' to help with the validator methods.

Some method names, across the project, have been changed to make use of the "timestamp" terminology.

Realised query params need to be converted from string to validate, this makes a query param record the default choice as enum map can only hold one data type in the values (of the key-value pairs).

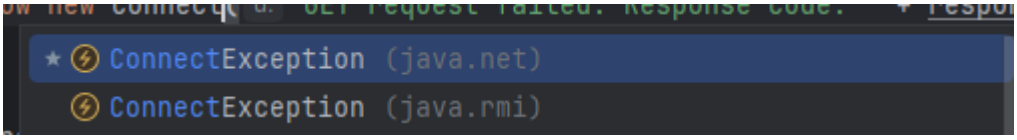
Added depth and limit validator methods to 'QueryValidator'.

Will turn 'QueryValidator' into a record another time, it is 2:30 am and I am tired and nauseous.

Gonna write 'HTTPRequest' method by using David's lecture example as a template. Seems to keep code organized, David made use of method overloading in the 'Fetcher' class.

Seems the sole purpose of “URI” is for stricter validation before making URL and easier to make using components instead of one single string arg. ‘HTTPRequest’ will assume that giving URL is already valid so URI will be used in another method.

URL seems to be a String wrapper class with extra methods specialised for HTTP communication from what I am seeing.



^ what? They even have the same parameter! **Which one do I use?!**

Academic week #10 (7th to 14th Dec)

So it seems u cant just convert json to any object, you either have to define the class that the json gets converted to or use a “JsonObject”. The later will be used.

Now that I think about it, JSON is not java core/vanilla code so how do I make streams compatible with JsonArray as it is an object instead of an array? … Turns out the answer was ‘StreamSupport.*stream*(‘ and ‘.spliterator()’.

Okay I think I made the first actually useful use of a lambda instead of a method reference:

```
StreamSupport.stream(allInstances.spliterator(), parallel: false) Stream<JsonElement>
    //^ Uses 'StreamSupport' to address JsonArray's lack of a '.stream()' method.
    .map(JsonElement::getAsJsonObject) Stream<JsonObject>
    .filter( JsonObject instance -> {
        if (!instance.has( memberName: "magType")) throw new JsonSyntaxException("Server response JSON has invalid stru
        //^ Prevents 'NullPointerException' when checking 'magType' field.
        return instance.get("magType").getAsString().startsWith("mw");
        //^ Only want Moment Magnitude type earthquakes.
        //^ '.startsWith' is used as there are multiple Moment Magnitude types such as 'mwc', 'mww', etc.
    })
    .forEach(filteredInstances::add);
    //^ Add all filtered instances into new JSON array.
    //^ As 'JsonArray' isn't supported by streams, we cannot use '.collect(' terminal method - uses for-loop instead.
```

Do the unfamiliarity of the ‘com.google.gson’ library, ‘RecordRetriever’ has been altered – adding new methods, changing existing method declarations, etc.

Processing of JSON object has inflated the “RecordRetriever” class too much, will need to split it into “HTTPRetriever” and “JSONToRecord”.

Might be easier to use static classes instead of singletons but I never used static classes before so **idk if it is right to use them**. Internet sources only explains the differences between them as 1 instance vs 0 instance.

As mentioned previously, ‘QueryValidator’ class has been changed to ‘APIQuery’ record.

So apparently, unlike unchecked exceptions, checked exceptions require method signatures when their ‘throw’ code is used.

Despite ‘JsonObject’ having no problems in the code itself, it is not recognised, and thus gives error, when mentioned in the Java docs as “@param JsonObject”.

‘HTTPRetriever.HTTPRequest’ is too big, thus I decided to split it into ‘HTTPRetriever.sendToServer’ and ‘HTTPRetriever.fetchJSON’.

Didn’t do anything for a 3 days because I had to stop taking medication for a week, before taking a breath test, that and dieting before the test made me very unwell.

Will need dedicated method (JSONToRecord.‘unixToDateTime’) to convert unix epoch time to ‘ZonedDateTime’ as time is the only earthquake data needing conversion.

Surely ‘.getAsJsonPrimitive(’ includes Long, right? … never mind, I found ‘.getAsLong(’

Did another commit (3rd one).

I have validation methods for data values but not the for the data types themselves – validation user inputs. This will be covered in ‘Interpreter’.

Debating if ‘Interpreter.processInputs’ is worth existing but decided to keep it when realising that validating user-input timestamp will be a lot more complex than e.g. magnitude decimal value.

Thinking about how to user input a time stamp. Was thinking for user to input one time scale (days, minutes, etc) but that will overwhelm the user with inputs so on single input format will be used instead: YYYY-MM-DD:HH (UTC is assumed).

To validate this timestamp, without requires many steps and checks, regex will be used as I have used them before in other languages - experienced.

Now that timestamp validation has been done, will need to make a new ‘Interpreter’ method that converts that string to ‘ZoneDateTime’ format as its not as simple as parse method call.

Now that’s done, I should prob have a ‘Interpreter.viewManual’ method as the string will be too big to be in one single line. This will relate to the ‘ClientActions.getManual’ interface action. Makes ‘ClientActions.getManual’ more than just a forwarder method.

Actually… I don’t need ‘Interpreter.viewManual’, I will put the string in the interface manual action itself.

So it turns out that ‘.append’ (of ‘StringBuilder’) can chain upon itself.

Added starting timestamp and ending timestamp to the interpreter class as those are two query args needed by the report generation.

Did I mention I made a ‘InputType’ enumeration?

Made a ‘Command’ enumeration.

‘Interface.cycle()’ don’t actually need to return bool as the exit method will just exit the whole program itself.

So apparently if I have `public enum Command { QUERY }`, calling `Command.QUERY` won’t give an integer… Must be implemented differently in other languages. Whatever it is it will be too complex to fix, so ill just ignore enum in this enhanced switch statement.

In interpreter class, ‘exportAllReports()’ method will just export to console, and ‘compareToPreviousReport’ will be unimplemented, both to save development time (at least at the moment).

How on earth did I forget to make a static ‘Main.main(’?!

Alr, ill get to test run the code before continuing.

Fixes from test runs:

- Forgot to add new line character (‘\n’) in the ‘Interpreter.getManual()’ meth.
- ‘Interpreter.submitQuery’ output console instead of throwing exception when something goes wrong.
- Forgot to a try statement in ‘Instance.interfacing’ to print checked and unchecked errors thrown by other try statements from the called methods.
- Forgot to add input instructions for each individual instructions, could add a print statement to ‘Interpreter.processInput’ but I could add a print statement for the query args via lambdas instead (implemented in ‘Interpreter.)!

```
    public void submitQuery() {
        APIQuery query = new APIQuery(
            () -> {
                System.out.println("Enter limit (integer between 1 and 20000): ");
                return Integer.parseInt(this.processInput(InputType.INTEGER));
            },
            Double.parseDouble(this.processInput(InputType.DECIMAL)),
```

^ Well that didn’t work.
Ig ill just add string param instead of lambda.

- Replaced invalid inputs exceptions, in ‘Interpreter.processInputs’ with console outputs as user/client should be giving another chance to give valid input instead of cancelling the whole operation.
- Changed magnitude validation from ` (magnitude < -5.0 || magnitude > 10.0) ` to ` (magnitude > -5.0 && magnitude < 10.0) , also double checked the other mathematical validation statements.
- HTTP connection gets code 400. Investigated example generated URL:
  “ https://earthquake.usgs.gov/fdsnws/event/1/query?format=geojson&jsonerror&limit=1&latitude=2.0&longitude=3.0&maxradiuskm=4&starttime2000-01-01T01:00:00Z&endtime=2010-01-01T01:00:00Z&minmagnitude=7.0&maxmagnitude=8.0&minDepth=9.0&maxDepth=10.0 ”
  Seems the error JSON actually shows the reason for the rejection.

```
Error 400: Bad Request

Unknown parameter "minDepth".

Usage details are available from https://earthquake.usgs.gov/fdsnws/event/1

Request:
/fdsnws/event/1/query?format=geojson&amp;jsonerror&amp;limit=1&amp;latitude=

Request Submitted:
2025-12-10T20:01:47+00:00

Service version:
1.14.1
```

Forgot to add “=” to `START_TIME("starttime"),` and needed to get rid of some capitalisation.

- Some accidently inverted if-statements.
- Forgot that ‘api’ field is actually sub-field.
- Gson library treats Strings, in JSON, as primitives instead of objects – changed `.getAsJsonObject("place")` to `.getAsJsonPrimitive("place")`.

//^ Gson lib lacks java documentation – very inconvenient.

ZonedTimeDate cannot be stored in JSON which means I cannot have ZonedTimeDate in the earthquake entry as otherwise ` gson.fromJson(formattedInstanceObj, EarthquakeEntry.class); ` will not work!!!!!!!!!!!!!!

Pushed 4th commit (realistically more like half a commit)

- Fixed out-of-index for-loop errors

To get around the ZonedTimeDate problem, I changed ZonedTimeDate type component in `EarthquakeEntry` to string and added `EarthquakeEntry.getTime()` to retrieve the string timestamp as ZonedTimeDate.

Adding more lambdas and streams

Now I am goin to go through the code and see any opportunity to replace any code piece with lambdas. Most replacements are:

- Replacing index-orientated for-loops with a ‘IntStream.range(‘ stream.
- Replacing loops with streams with ‘.forEach’.
- Replacing ‘.collect’ instead of ‘.forEach’ for concatenation/appending operations in streams.
- Using method references instead of basic lambdas where possible.

While I was already aware of these, I came across mention of a command map from https://stackoverflow.com/questions/41291743/java-8-mapstring-runnable-control-flow . However, I cannot find a tutorial for it for some reason. Maybe it not that popular? The closest thing is a tutorial named “Replace Multiple If Else Conditions With Map” but it is mostly behind a paywall. I will have to code in a try and error brute force way.

```java
private void cycle() {  1 usage  jh1662
    String command = processInput(InputType.INTEGER,  promptMessage: "Enter command (type '7' for help): ");
    switch (command) {
        case "1" -> this.submitQuery();
        case "2" -> this.generateReport();
        case "3" -> this.viewRawDataSet();
        case "4" -> this.exportAllReports();
        case "5" -> this.compareToPreviousReport();
        case "6" -> this.exitProgram();
        case "7" -> this.getManual();
        default -> System.out.println("Invalid command; please input a valid command number (1-7).");
    }
}
```

^ before

```java
private void cycle() {  1 usage  jh1662 *
    Map<String, Runnable> commandMap = Map.of(
        //* Map but the values are lambdas (specifically method references), instead of data, for better readability.
        "1", this::submitQuery,
        //^ Would be `case "1" -> this.submitQuery();` in a enhanced switch statement.
        "2", this::generateReport,
        k3: "3", this::viewRawDataSet,
        k4: "4", this::exportAllReports,
        k5: "5", this::compareToPreviousReport,
        k6: "6", this::exitProgram,
        k7: "7", this::getManual
    );
    String command = processInput(InputType.INTEGER,  promptMessage: "Enter command (type '7' for help): ");
    //^ Take valid integer-parsable string user input for command selection.
    commandMap.getOrDefault(command, () -> System.out.println("Invalid command; please input a valid command number (1-7).")).run();
    //^ Run the corresponding command lambda, or print invalid command message as specified in the arg lambda.
}
```

^ after

Did I mention that I used a regex statement, at one point, for timestamp validation?

Done Javadocs for the rest of the classes, records, and enums except the unused/uncalled enums.

JUnit

For some reasons, while program works fine, it does not in the Junit tests as it seem to bot be able to read the second string from `System.*setIn*(new ByteArrayInputStream(consoleInput.getBytes()));`. After SEVERAL HOURS of debugging and research, the reason seems to be because of the program declares `this.scanner = new Scanner(System.in); ` once per interface loop, instead of once per program run. I thought scanner declaration must be only in method scope for good practice, as only one method (‘.processInput’) uses it, but it seems it can be an acceptation as scanner field can be a good representer of ‘Interface’ class state.

To prevent inputting console 1000s times I will change the tested class from retriever to ‘Interpreter’ class.

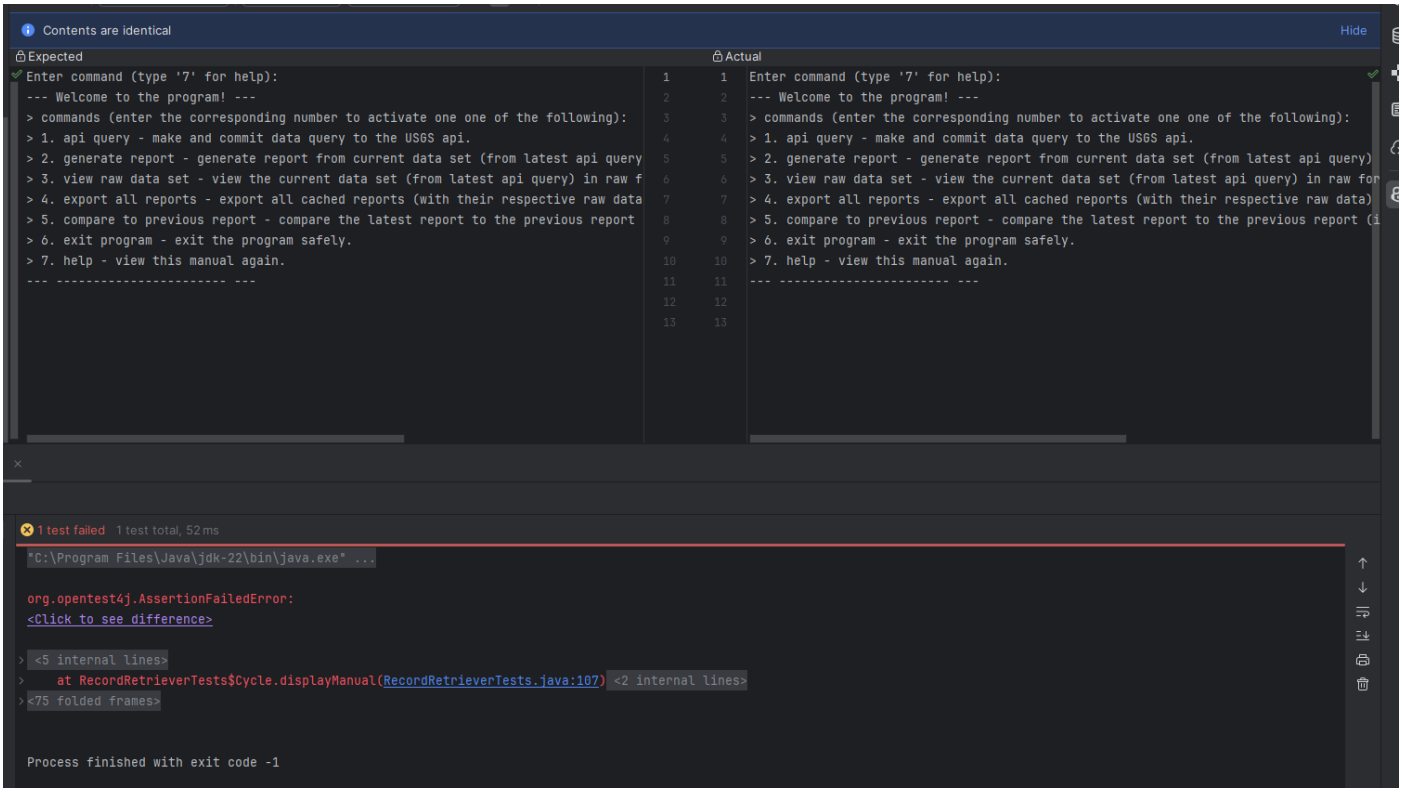
As mentioned before, the JUnit testing class one class where each tested method will be done by a nested class and each test case will be each own method; this is so that I can immediately pinpoint what exact test case, rather than the just group of tests, has failed.

To combat the problem of the inability to test private methods, I will use I learnt in David's lecture (reflection code) to change private method modifier without changing the private method declaration code itself.

So I set automatic console input, now I gotta set console output capture or something like that. Done so using the built-in Java classes 'ByteArrayOutputStream' and 'PrintStream'.

At first the output didn't work but after a while found out that was because 'Interpreter' (tested class) must be declared after the predefined input code.

okay this is stupid, Junit failed an equal assertion because one used LF and the other CRLF.



^ This makes ZERO SENSE.

Okay OKAY I got an idea

```
Assertions.assertEquals(expected.replace( target: "\r\n",  replacement: "\n").trim(), printed.replace( target: "\r\n",  replacement: "\n").trim());
```

^ test case succeeded – yippie!

Now regarding testing the exit feature, that cannot be tested as the `system.exit(0)` also terminates the Junit testing before results can be analysed. Because of this, that feature cannot be tested.

When manually testing, invalid commands simple result in a "is invalid input; must be an integer." Junit testing gives a NullPointerAcception even for valid commands. It seems that Junit is unable to recognise the error the IllegalStateException which is strange as `e.printStackTrace();` shows it but `Assertions.assertEquals(expectedExceptionMsg, e.getMessage());` just sees actual as "null". Okay, After hours AGAIN, it seems that the caught error was wrapped as a "InvocationTargetException" exception and `e.getCause()` is used to unwrap the error. I think I do remember hearing something like that in David's lectures but It was difficult to concentrate due to keeping my vomit down (was sick) -_- .

In a confliction as Interpreter must be instantiated before the `System.setOut(new PrintStream(outputStream));` but such, unlike method reflection code, field reflection code require instantiation before that. Resulted to adding new parameter and setup logic to the overloaded method.

I am starting to get to a point where I don't have to debug every test creation which is a good thing; gotta love reflection code saving my Junit testing.

Testing the cycle method seem to make it a massive, nested class. May need to consider having each nested class dedicated to a command type instead of just the method.

When figuring out how to call methods of fields, rather than setting value, from reflection code, I came across:

- https://stackoverflow.com/questions/49825141/add-an-object-to-an-arraylist-using-reflection
- https://www.baeldung.com/java-reflection-class-fields

These sources basically make use of ".get(" to achieve the goal.

I wonder if its possible to put a HTML table in a java doc so I can show what tests are done and why. I wonder that because I already use "<p>" in them. Appears that one can! Will do that sometime in the future.

Yeah... will definitely rearrange how the nested test classes work; but this will massive change the testing structure thus another commit will be pushed.

Pushing 5th commit – Second half of the half commit and first structured Junit testing (incomplete)

Academic week #11 (15th to 19th)

JUnit

Changes (in the test file):

- Renamed file to more relevant naming.
- Added helper methods to help shorten test cases.
- Due to nested class being unable to call members of the outer class, any member of the outer class must be static!
- Seems that instead of having `return` in each case of an enhanced switch, you just put `return` before `switch` declaration – very useful information.
- Trivial features (like exiting), that requires only one text case will all be put in a "miscellaneous" nested class as its not worth having single-method nested classes.
- I have realised I have put the exit command ("6") in some tests but that is not necessary as those test the 'cycle' method instead of the 'interface' method – removed them.
- Was going to split nested class up but I am close to spending as much time on the testing code than the actual program!

- Everytime when I try to call `giveExampleValidEarthquakeReport`, it causes the IDE to freak out and causes every static member to give a “Compact source files are not supported at language level '22'”

Spent few more days debugging. The problem is that unlike normal debug, Junit debug mode often gives errors before reaching the error-causing code line which cause me unable to pinpoint the location of fault. Also, those wrap errors and other unknown error types it very difficult to gather the other context of the error.

While I think I have done enough tests in both quantity and quality, I still make earthquake entry record creation tests; those tests because they are quicker to make Junit testing than manual console execution.

I remember David talking about try-catch statements being the only way to expect errors in Junit, when I asked him about any Junit-specific way for error expectance, but it seems there in fact one called `assetThrows(`.

Well, I am happy to announce that both valid edge tests and invalid edge test cases, of Earthquake entry construction, were all successful.

So my system.out stream has been redirected, and “Assertions.*fail*(” is terminal, but there is “System.err.println” which prints to a different console stream and does so without ending the program (contrary to its name/syntax).

Other changes

Deleted ‘Command’ and ‘EarthquakesStats’ enums are they are not called in the program.

Fixed miscalculation in monthly frequency.

Edited report rendering so it says when mean intermission time is not appropriate to generated instead of just saying zero as the mean value.

Written up Interpreter method for actual comparison but later delegated responsibility to ‘EarthquakeReport.compareTo(’.

Moved README.md out of ‘src’ folder as it is not a source file, just as the development log.

Thought making method take string instead of record will show good practice (loose coupling), but not doing it is better practice show as it shows my use and understanding of Records.

Added `.trim()` to minimise unexpected behaviour from calling ‘Interpreter.processInput(‘.

Yeah okay now im confused some error require ‘.getCause’ while ‘.getMessage’ gives null but sometimes it’s the exact opposite.

Should prob make timestamp, in console output, more human readable than like “2000-09-09T06:01:53.400Z[UTC]”. Added ‘EarthquakeEntry.getHumanReadableTimestamp” that makes it more human-readable like “Thursday, January 8, 1970 12:24 UTC”.

Java code not Javaing

So apparently enhanced switches, for enums, don't need enum declaration for every case so removed them e.g: `case InputType.STRING ->{` to `caseSTRING ->{`.

So it seems that 'DateTimeFormatter' not only is used for formatting for human readable but also can be used to validate timestamp string inputs! Okay so 'ZonedDateTime.parse(' tolerates invalid timestamps, so I used the stricter 'LocalDateTime.parse(' instead. That does not work... I have to find another way to fix it.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd").withResolverStyle(ResolverStyle.STRICT);
```

//^ So this never works

```
DateTimeFormatter formatter = new DateTimeFormatterBuilder().appendPattern("yyyy-MM-dd").parseStrict().toFormatter(Locale.ROOT);
```

//^ **But this works for some unknown reason**

WHY?!!!!!!

```
128         case TIMESTAMP -> {
129             try{
130                 //* This scope is the main reason for this method ('this.processInputs') to exist.
131                 //* String format expected: YYYY-MM-DD:HH (UTC is assumed).
132                 //* Because the local processing only uses up to by-the-hour precision, we only validate up to that - user won't input minutes and shorter sc
133                 //: Validating timestamp requires many steps and checks - thus we use regex to simplify the process.
134                 String timestampCheckRegex = "^\\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\\d|3[01]):([01]\\d|2[0-3])$";  timestampCheckRegex: "^\d{4}-(0[1-9]|1[0-2
135                 //^ Not only checks the format, but also the ranges of month, day, and hour.
136                 if (!userInput.matches(timestampCheckRegex)) {  timestampCheckRegex: "^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01]):([01]\d|2[0-3])$"
137                     System.out.println(userInput + " is invalid input; must be in 'YYYY-MM-DD:HH' format (UTC time zone).");
138                     return this.processInput(InputType.TIMESTAMP, promptMessage);  promptMessage: "Enter the start timestamp (format 'YYYY-MM-DD:HH' in UTC t
139                 }
140
141                 //: Checks the number components are within bounds.
142                 DateTimeFormatter formatter = new DateTimeFormatterBuilder().appendPattern("yyyy-MM-dd").parseStrict().toFormatter(Locale.ROOT);  formatter:
143                 try { LocalDate.parse(userInput.split( regex: ":")[0], formatter); }  userInput: "2001-02-30:00"  formatter: "Value(YearOfEra,4,19,EXCEEDS_PA
144                 catch (Exception e) {
145                     System.out.println(userInput + " is invalid input; unable to parse timestamp due to one of the number components being out of bounds.");
146                     return this.processInput(InputType.TIMESTAMP, promptMessage);
147                 }
148             }
149             catch (Exception e) { System.out.println("something has happened wrong");}
150         }
```

So the number is working now but the error doesn't get caught despite happening in line 143, **HOW ON EARTH DOES THAT MAKE SENSE?!**

Well at least the error is handled somehow.

I am also confused why Junit also cannot detect the error either...

I have tried everything it was either this, all inputs being reject, or all inputs being accepted.

After much more research and concerningly deep debugging sessions, it seems to Java's own problem:

```
146                 try {
147                     System.out.println("DEBUG: about to parse");
148                     LocalDate.parse(datePart, formatter);
149                     System.out.println("DEBUG: parse succeeded");
150                 } catch (Exception e) {
151                     System.out.println("DEBUG: CAUGHT " + e.getClass().getName() + " — " + e.getMessage());
152                     e.printStackTrace(); // shows exact throw site and stack trace
153                     return this.processInput(InputType.TIMESTAMP, promptMessage);
154                 }
```

^ In this sample code snippet, only the first 2 print lines prints worked before an uncaught error seem pop out of nowhere.

I will stop this part as it has been 20 hours across 2 says now.

David if you are reading this please tell me what is wrong because I genuinely have no idea.

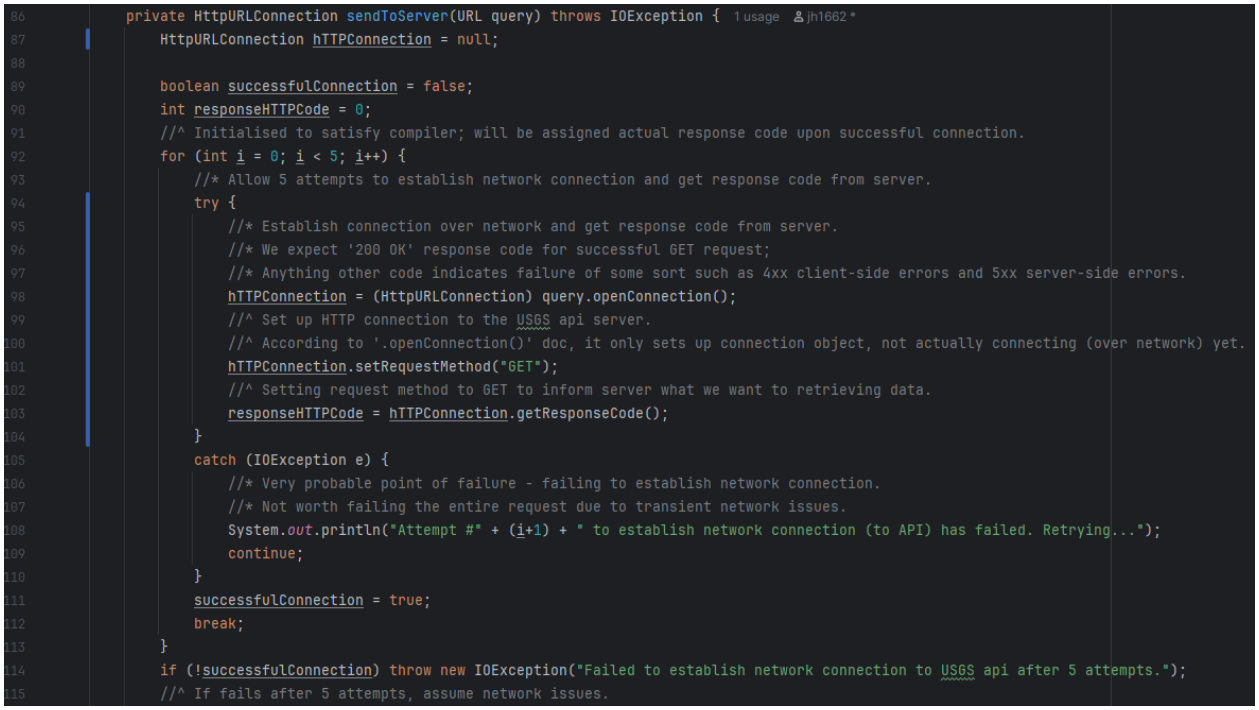
Final polishing

- ` *new BufferedReader(new InputStreamReader(hTTPConnection.getInputStream()));* ` to ` *new BufferedReader(new InputStreamReader(hTTPConnection.getInputStream(), StandardCharsets.UTF_8))* ` to explicitly specify how to read the file.
- Used try-with-resource statements as a better practice alternative to closing the buffer (` *try (BufferedReader bufferReader = new BufferedReader(new InputStreamReader(hTTPConnection.getInputStream(), StandardCharsets.UTF_8))){* `)
- Before I check the response code in the JSON response.
- So it seems it is good practice to display the body of the HTTP response when response code is not in the 2XX range (error response code) so I did that with `hTTPConnection.getErrorStream();`.
- ` hTTPConnection = (HttpURLConnection) query.openConnection(); ` to be executed per connection attempt , instead of once, to not carry over problem to the next try:

```
HttpURLConnection hTTPConnection;
try {
    hTTPConnection = (HttpURLConnection) query.openConnection();
    //^ Set up HTTP connection to the USGS api server.
    //^ According to '.openConnection()' doc, it only sets up connection object, not actually connecting (over network) yet.
    hTTPConnection.setRequestMethod("GET");
    //^ Setting request method to GET to inform server what we want to retrieving data.
}
catch (IOException e) { throw new IOException("Failed to open HTTP connection to USGS api."); }

boolean successfulConnection = false;
int responseHTTPCode = 0;
//^ Initialised to satisfy compiler; will be assigned actual response code upon successful connection.
for (int i = 0; i < 5; i++) {
    //* Allow 5 attempts to establish network connection and get response code from server.
    try { responseHTTPCode = hTTPConnection.getResponseCode(); }
    //^ Establish connection over network and get response code from server.
    //^ We expect '200 OK' response code for successful GET request;
    //^ Anything other code indicates failure of some sort such as 4xx client-side errors and 5xx server-side errors.
    catch (IOException e) {
        //* Very probable point of failure - failing to establish network connection.
        //* Not worth failing the entire request due to transient network issues.
        System.out.println("Attempt #" + (i+1) + " to establish network connection (to API) has failed. Retrying...");
        continue;
    }
    successfulConnection = true;
    break;
}
if (!successfulConnection) throw new IOException("Failed to establish network connection to USGS api after 5 attempts.");
//^ If fails after 5 attempts, assume network issues.
```

^ before



```
private HttpURLConnection sendToServer(URL query) throws IOException {  1 usage   jh1662 *
    HttpURLConnection hTTPConnection = null;

    boolean successfulConnection = false;
    int responseHTTPCode = 0;
    //^ Initialised to satisfy compiler; will be assigned actual response code upon successful connection.
    for (int i = 0; i < 5; i++) {
        //* Allow 5 attempts to establish network connection and get response code from server.
        try {
            //* Establish connection over network and get response code from server.
            //* We expect '200 OK' response code for successful GET request;
            //* Anything other code indicates failure of some sort such as 4xx client-side errors and 5xx server-side errors.
            hTTPConnection = (HttpURLConnection) query.openConnection();
            //^ Set up HTTP connection to the USGS api server.
            //^ According to '.openConnection()' doc, it only sets up connection object, not actually connecting (over network) yet.
            hTTPConnection.setRequestMethod("GET");
            //^ Setting request method to GET to inform server what we want to retrieving data.
            responseHTTPCode = hTTPConnection.getResponseCode();
        }
        catch (IOException e) {
            //* Very probable point of failure - failing to establish network connection.
            //* Not worth failing the entire request due to transient network issues.
            System.out.println("Attempt #" + (i+1) + " to establish network connection (to API) has failed. Retrying...");
            continue;
        }
        successfulConnection = true;
        break;
    }
    if (!successfulConnection) throw new IOException("Failed to establish network connection to USGS api after 5 attempts.");
    //^ If fails after 5 attempts, assume network issues.
```

^ after

- Added `hTTPConnection.disconnect();` for good practice (similar reason for flushing/closing the buffer) and to give the program a clean restart when retrying HTTP connections.
- So apparently, a connection attempt by last indefinitely due to being left hanging due to network issues. To prevent program from being stalled, timeouts are used by using the code `*hTTPConnection.setConnectTimeout(5_000);*` and `*hTTPConnection.setReadTimeout(10_000);*`.
- `Thread.sleep(1000 * (i+1));` to prevent error #429 and give time for server to recover from other server related errors.
- Added HTML hyperlink in Javadoc
- Seems the stream '.collect' can indeed be used on type JsonArray so it has been made so in 'JSON.filterEarthquakeInstances'.
- Changed type of 'time' component in 'EarthquakeEntry' from String to long.
- Revised README.md
- Rounded up the numbers in the report comparison

Another commit

Commit notes

Fully implemented `EarthquakeEntry` record and implemented the essential parts of the `EarthquakeReport` record.

This made changes to the project skeleton plan:
** Declared `viewRawDataSet` action (in `ClientActions` interface) to prevent the earthquake report string representation from being too big for user to take in. This causes declaration of `EarthquakeReport.renderRawDataSet()` and `Interpreter.viewRawDataSet`.*
** Declared and implemented `EarthquakeReport.findMeanDepth` and corresponding field.*
** Swapped `java.time.LocalDateTime` with `java.time.ZonedDateTime`, throughout entire project,*

to enforce the UTC timezone. Additional imports are used in for further use of `ZonedDateTime`.

Next will need to do javadoc comments and further addition of earthquake depth integration.

Added initial project structure of Earthquake data handling, API fetching, records, enums, and validation classes. This is used to generate an initial UML.

Development log is kept in a MS Word doc for now. Manual will be in README.md in a letter commit.

Added Java docs to the records, made new records, added new classes and methods as the plan comes into the write-up.

Due to the use of images, development doc will remain as MS Word and now be included in the resp.

More information on changes can be seen in the development log file.

Written up "RecordRetriever" which was too large so splitted init "HTTPRetriever" and "JSONToRecord".

Next will implement 'Interpreter'.

See notes, in Development log.docx, after mention of 3rd commit.

See notes, in `Development log.docx`, after mention of 4th commit.

Basic overview - implementation of program's necessaries, done Javadoc commenting on majority of the classes and their methods (including enums and records), and halfway through testing automation.

See notes, in Development log.docx, after mention of 5th commit.

Questions asked to Copilot (using "Smart (GPT-5)")

Question	**Response**
what do u call it when u generate cords for a point which is the avg point on the graph to all other points?	centroid
what do u call langtitude and langtitude together as one word?	coordinate
What import to use for ISO8601 and UTC format time and date	Too many to list, chose the 'java.time. LocalDateTime' one
What types of magnitude scales are there?	https://www.usgs.gov/programs/earthquake-hazards/magnitude-types
What is the closest thing to a universal magnitude type/scale?	Mw (Moment Magnitude)
"fatal: mmap failed: Invalid argument"	Use "rm .git\index -Force" then "git reset"

Is “EatthquakeEntry::latitude” a lambda	That is a method reference which is not the same as a lambda.
Why are ‘ZonedDateTime’ instances not comparable?	They don’t implement the 'Comparable' interface.
Can an earthquake’s depth be negative?	https://www.usgs.gov/faqs/what-does-it-mean-earthquake-occurred-a-depth-0-km-how-can-earthquake-have-a-negative-depth
How do I use streams on unsupported data structures?	‘java.util.stream.StreamSupport’
Why error message is just “null” in Junit assertions?	Error will be given as a ‘InvocationTargetException’, need to unwrap it to get the original error. To unwrap it, do ` Throwable cause = e.getCause();`
Can I put HTML tables in java docs?	Yes, Javadoc supports: <table>, <tr> ,<th> , and <td>
Why can’t I put `return` in enhanced switch?	You are supposed to put them before ‘switch’ statement declaration, not in the case scopes.
String of “E” characters but 512 characters long.	`“E”.repeat(512);`
what is used to expect thown checked errors in Junit.	assertThrows
HTML list in Javadocs	Use `<ul>` and `<li>`
` ZonedDateTime.parse` does not catch invalid dates like Feb 30th	It is tolerant, use `LocalDateTime.parse` as its more strict.
^ That changes nothing	Also use `.withResolverStyle(ResolverStyle.STRICT)`