CSE214

HOMEWORK 7 - Spring 2023

HOMEWORK 7 - due Tuesday, May 2nd no later than 7:00PM

Reminders:

- **Submit your files only via CodeGrade in the Content page on Brightspace. Access CodeGrade by clicking "HW7 - CODEGRADE SUBMISSION LINK" for your submission.**
- **Use of a package is optional. If you wish to use it, make sure to name it "hw7" (all in lower case). Otherwise, you will lose points.**
- **Be sure your code follows the coding style for CSE214.**
- **Make sure you read the warnings about academic dishonesty.** ***Remember, all work you submit for homework or exams MUST be your own work.***
- **You are allowed to use any Java API Data Structure classes such as LinkedList, ArrayList or Vector to implement this assignment.**
- **You may use Scanner, InputStreamReader, or any other class that you wish for keyboard input.**

Assignment

In this assignment, you will create an application which views and sorts a database of Near Earth Objects (i.e. asteroids) that are provided live via a public NASA API. To make the assignment simpler, you will be able to use a special `BigData` library to parse the text file returned by the API directly into Java objects.

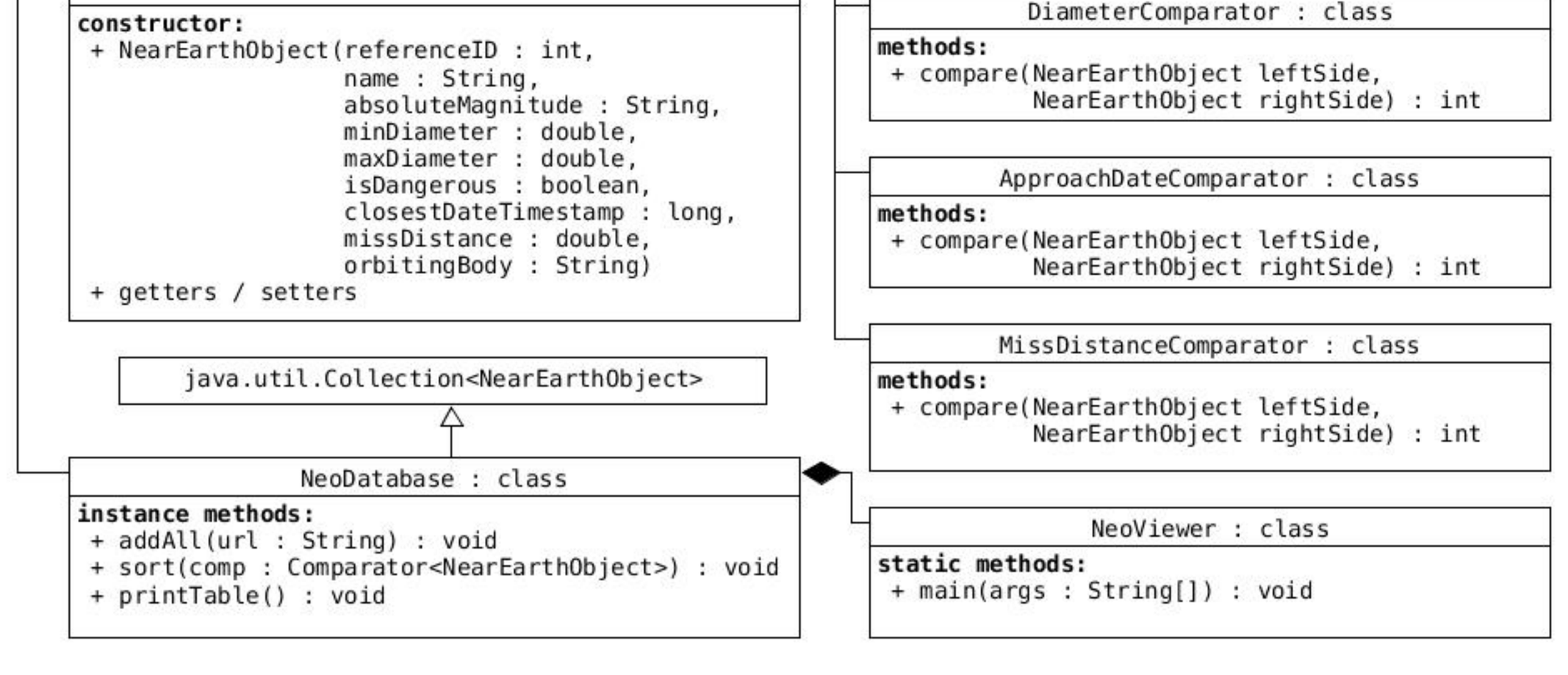
The data structure used for this assignment is up to you - **you may use any data structure you think is best to implement the database**. Before settling on a data structure, however, be sure to consider how the database will be used, and which data structures we've studied are best suited for these purposes.

This assignment is unlike others you've worked on so far - the data you will be retrieving is coming from a live web server via a REST API. REST, or REpresentational State Transfer, is a simple yet popular system for exchanging data over a network (*many real-time applications use RESTful services to retrieve the most up-to-date data*). A RESTful service provides data to clients using simple HTTP requests which return data files instead of HTML documents, typically in the form of JSON or XML.

Note: Although this assignment requires you make web requests and parse text data, nearly all of this work is done by the `BigData` library. Be sure to read the entire specification before beginning the assignment, as there are certain steps you can do which will greatly reduce the work required for this assignment.

Required Classes

The instructions detailed in this specifcation should be considered more as a guidelines than as fixed rules. Please feel free to add additional classes, methods, member variables as you see fit. As long as the requirements are satisfied, i.e. all operations in the grading key are supported, you may be as flexible as you like when implementing the assignment. Please keep in mind that standard OOP practices should still be followed, and points will be deducted for poor software design (*e.g. using `public` instance variables*).



UML Diagram

1. `NearEarthObject` Class

Write a fully documented class named `NearEarthObject`. This class represents a record in the database of asteroids currently tracked by NASA. It should be noted that this class will **always** be constructed by the `BigData` library, and serves as a data container for the information that is already hosted by the NeoW API. Please refer to the **`BigData`** section below for more information on how the library will use this constructor to extract `NearEarthObjects` out of the JSON document returned by the NASA NeoW API.

- **`private int referenceID`**
- **`private String name`**
- **`private double absoluteMagnitude`**
- **`private double averageDiameter`**
- **`private boolean isDangerous`**
- **`private Date closestApproachDate`**
- **`private double missDistance`**
- **`private String orbitingBody`**
- **`public NearEarthObject(int referenceID, String name, double absoluteMagnitude, double minDiameter, double maxDiameter, boolean isDangerous, long closestDateTimestamp, double missDistance, String orbitingBody)`**
  - *Brief:*
    - Default Constructor.
    - Note: It is very important that your constructor exactly match this signature. This constructor will be used by the BigData library to fetch the `NearEarthObject` records from the NeoW API.
  - *Parameters:*
    - `referenceID`
      - Unique the ID of the NEO.
      - Fetched using the "`near_earth_objects/neo_reference_id`" identifier.
    - `name`
      - Unique name of the asteroid or orbital body.
      - Fetched using the "`near_earth_objects/name`" identifier.
    - `absoluteMagnitude`
      - Absolute brightness of the asteroid or orbital body in the sky.
      - Fetched using the "`near_earth_objects/absolute_magnitude_h`" identifier.
    - `minDiameter`
      - Estimated minimum diameter of the asteroid or orbital body in kilometers. This parameter should be used in conjunction with the `maxDiameter` parameter to calculate and initialize the `averageDiameter` member variable.
      - Fetched using the "`near_earth_objects/estimated_diameter/kilometers/estimated_diameter_min`" identifier.
    - `maxDiameter`
      - Estimated maximum diameter of the asteroid or orbital body in kilometers. This parameter should be used in conjunction with the `minDiameter` parameter to calculate and initialize the `averageDiameter` member variable.
      - Fetched using the "`near_earth_objects/estimated_diameter/kilometers/estimated_diameter_max`" identifier.
    - `isDangerous`
      - Boolean value indicating whether the astroid or orbital body is a potential impact threat.
      - Fetched using the "`near_earth_objects/is_potentially_hazardous_asteroid`" identifier.
    - `closestDateTimestamp`
      - Unix timestamp representing the date of closest approach. Note that this can be used to directly construct the `closestApproachDate` member variable, as the `Date` class provides a constructor taking a timestamp as a parameter.
      - Fetched using the "`near_earth_objects/close_approach_data/epoch_date_close_approach`" identifier.
    - `missDistance`
      - Distance in kilometers at which the asteroid or orbital body will pass by the Earth on the date of it's closest approach.
      - Fetched using the "`near_earth_objects/close_approach_data/miss_distance/kilometers`" identifier.
    - `orbitingBody`
      - Planet or other orbital body which this NEO orbits.
      - Fetched using the "`near_earth_objects/close_approach_data/orbiting_body`" identifier.
- **Getters and setters for all member variables.**

2. `Comparator` classes

The assignment requires `NearEarthObject` instances to be sorted by **four** different member variables: `referenceID`, `averageDiameter`, `closestApproachDate`, and `missDistance`. To accomplish this, you should create four different classes which implment the `Comparator` interface which allow the `NearEarthObjects` to be arranged in sorted order based on the value of these member variables.

1. **`ReferenceIDComparator implements java.util.Comparator<NearEarthObject>`**
2. **`DiameterComparator implements java.util.Comparator<NearEarthObject>`**
3. **`ApproachDateComparator implements java.util.Comparator<NearEarthObject>`**
4. **`MissDistanceComparator implements java.util.Comparator<NearEarthObject>`**

Each of these classes should implement a **`int compare(NearEarthObject left, NearEarthObject right)`** method. Each method in these classes should compare the two arguments based on the value of the desired member variable. For example, `ReferenceIDComparator` would compare two `NearEarthObjects` based on the values of their `referenceID` member variables, `DiameterComparator` would compare them based on the values of their `averageDiameter` member variables, etc. For more information, please refer to the documentation on the `Comparator` interface provided by Oracle.

Comparable/Comparator Example:

```
/*
 * An example of type abstraction that implements Comparable
 * and Comparator interfaces.
 *
 * The Comparable/Comparator interfaces provide a standard means
 * for communication with yet unknown types of objects.
 */

public class CollectionsTester {
    public static void main(String[] args) {
        ArrayList<Employee> staff = new ArrayList<Employee>();

        staff.add(new Employee("Joe",100000, 177700010));
        staff.add(new Employee("Jane",200000, 111100010));
        staff.add(new Employee("Bob",66666, 1999000010));
        staff.add(new Employee("Andy",77777, 188800010));

        Collections.sort(staff);                                  // Sort by salary
        System.out.println("Lowest paid employee: "+staff.get(0));    // Prints Bob

        Collections.sort(staff, new NameComparator());            // Sort by alphabetical order
        System.out.println("First employee in list: "+staff.get(0));  // Prints Andy

        Collections.sort(staff, new IdComparator());              // Sort by ID number
        System.out.println("Employee with lowest ID: "+staff.get(0)); // Prints Jane
    }
}

public class Employee implements Comparable {
    private String name;
    private int salary;
    private int id;
    public Employee(String initName, int initSal, int initId) {
        id     = initId;
        name = initName;
        salary = initSal;
    }
    public String getName(){ return name; }
    public int getSalary() { return salary; }
    public int getId(){ return id; }
    public void setSalary(int newSalary) {
        salary = newSalary;
    }
    public int compareTo(Object o) {
        Employee otherEmp = (Employee)o;
        if (this.salary == otherEmp.salary)
            return 0;
        else if (this.salary > otherEmp.salary)
            return 1;
        else
            return -1;
    }
    public String toString() {
        return name + ", $" + salary + ", "+ id;
    }
}

public class NameComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Employee e1 = (Employee) o1;
        Employee e2 = (Employee) o2;
        return (e1.getName().compareTo(e2.getName()));
    }
}

public class IdComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Employee e1 = (Employee) o1;
        Employee e2 = (Employee) o2;
        if (e1.getId() == e2.getId())
            return 0;
        else if (e1.getId() > e2.getId())
            return 1;
        else
            return -1;
    }
}
```

3. `NeoDatabase` class

Write a fully documented class named `NeoDatabase` which will contain and manage the `NearEarthObject` records which have been downloaded from the online dataset. The specific data structure used to implment this database is up to you - any Java API class may be used or customized for this purpose. While deciding which structure to use, be sure to consider the operations which will be performed on this database, as well as the pros and cons for using various structures such as `LinkedLists`, `Trees`, `Hashtables`, and the like (*keep in mind that `Collections.sort()` can only operate on a class which implements the `List` interface*).

- **`public static final String API_KEY = /* Unique Key here. */`**
  - API Key specific to this application used to perform queries to the NASA NeoW API. You must register for one at this page in order to make queries to the dataset.
  - Warning: The API limits requests to 1,000 per hour per unique API_KEY. Though it is unlikely that you should ever reach this request number, it should be noted that there is a limit to how many requests you may make.
- **`public static final String API_ROOT = "https://api.nasa.gov/neo/rest/v1/neo/browse?"`**
  - URL of the REST API used to conduct queries. The query parameters will be appended to the end of this string, which will indicate the page number requested by the user along with the program's uniqe `API_KEY` described above (*see `buildQueryURL()` below for further information*).
- **`public NeoDatabase()`**
  - *Brief:*
    - Default Constructor.
  - *Postconditions:*
    - The database has been constructed and is empty.
- **`public String buildQueryURL(int pageNumber) throws IllegalArgumentException`**
  - *Brief:*
    - Builds a query URL given a page number. This should be a simple method which returns `(API_ROOT + "page=" + pageNumber + "&api_key=" + API_KEY)`
  - *Parameters:*
    - `pageNumber`
      - Integer ranging from 0 to 715 indicating the page the user wishes to load.
  - *Preconditions:*
    - `0 ≤ page ≤ 715`.
  - *Throws:*
    - `IllegalArgumentException`
      - If `pageNumber` is not in the valid range.
- **`public void addAll(String queryURL) throws IllegalArgumentException`**
  - *Brief:*
    - Opens a connection to the data source indicated by `queryURL` and adds all `NearEarthObjects` found in the dataset.
  - *Parameters:*
    - `queryURL`
      - String containing the URL requesting a dataset from the NASA NeoW service (*should be generated by `buildQueryURL()` above*).
  - *Preconditions:*
    - `queryURL` is a non-null string representing a valid API request to the NASA NeoW service.
  - *Postconditions:*
    - All `NearEarthObject` records returned have been added to the database, or else a `IllegalArgumentException` has been thrown.
  - *Throws:*
    - `IllegalArgumentException`
      - If `queryURL` is `null` or cound not be resolved by the server.
- **`public void sort(Comparator<NearEarthObject> comp) throws IllegalArgumentException`**
  - *Brief:*
    - Sorts the database using the specified `Comparator` of `NearEarthObjects`.
  - *Parameters:*
    - `comp`
      - `Comparator` of `NearEarthObjects` which will be used to sort the database. This parameter can be any of the required `Comparator` classes listed above.
  - *Preconditions:*
    - `comp` is not null.
  - *Postconditions:*
    - The database has been sorted based on the order specified by the inidcated `Comparator` of `NearEarthObjects`.
  - *Throws:*
    - `IllegalArgumentException`
      - If `comp` is `null`.
  - *Note:*
    - This method can utilize the `Arrays.sort(T[] array, Comparator<T> comp)` or `Collections.sort(List<T> array, Comparator<T> comp)` functions to sort the database (*note that these functions only take an array of objects and a `List` of objects, repectively*).
- **`public void printTable()`**
  - *Brief:*
    - Displays the database in a neat, tabular form, listing all member variables for each `NearEarthObject`. Note the table should be printed in the order specified by the last **sort()** call.
  - *Preconditions:*
    - This `NeoDatabase` is initialized and not null.
  - *Postconditions:*
    - The table has been printed to the console but remains unchanged.

4. `NeoViewer` class

Write a fully-documented class named `NeoViewer` which allows a user to view datasets obtained from the NASA NeoW API. This class should contain a `main()` method which creates a database and prompts the user to add a page to the database, sort the current database, and display the database.

- **`public static void main(String[] args)`**
  - *Brief:*
    - The main method runs a menu driven application which creates a `NeoDatabase` instance and then prompts the user for a menu command selecting the operation. The required information is then requested from the user based on the selected operation. Following is the list of menu options and their required information:

A sample menu is provided below indicating the functionality your `NeoViewer` program should support:

```
A) Add a page to the database <page>
S) Sort the database
// Sub-menu
    R) Sort by referenceID
    D) Sort by diameter
    A) Sort by approach date
    M) Sort by miss distance
P) Print the database as a table.
Q) Quit
```

A few notes for the above operations:

- When the user wishes to add a page to the database, `<page>` parameter should be used to construct a query string to be used to construct a `DataSource`.
- The sort operation should display the indicated sub-menu for selecting which variable to sort on.

Big Data Library:

This assignment requires information to be extracted from JSON files provided by the NASA NeoW API service. In order to simplify the data extraction process, you can use the `BigData` library to parse `NearEarthObject` records from the datasets. This library provides several powerful data extraction tools for XML and JSON formatted data, and is capable of transforming structured text into Java objects using a special constructor convention.

In order to utilize the `BigData` library, you must include `bigdata.jar` in to your project. You can include a `JAR` file in the following manner, depending on your IDE: (DO NOT SUBMIT THE JAR FILE WITH YOUR JAVA FILES. OTHERWISE 10 POINTS WILL BE DEDUCTED)

Using a JAR in Eclipse:

- Right click the project name in the "Package Explorer" tab (on the left, by default) - Select "Build Path" - Select "Add External Archives..." - Navigate to where you saved `bigdata.jar` and select it.

Using a JAR NetBeans:

- Right click the project name in the "Package Explorer" tab (on the left, by default) - Click on "Properties" - Select "Libraries" on the left - Click on "Add JAR/Folder" on the right - Navigate to where you saved `bigdata.jar` and select it.

Using a JAR in IntelliJ: See here

Using a JAR on the command line: See here

Now you can "`import big.data.DataSource`" in your source code (or any other class from the `big.data` library that you need).

Note: The NASA NeoW service returns a page of records in the form of a JSON file. JSON is a popular syntax structure used to store and exchange data between applications, similar to XML or YML. For this assignment, **you do not need to know the inner workings of JSON documents** - the `BigData` library will do all the complex parsing and data extraction for you.

You can connect a `DataSource` object to a JSON file by using the static `connectJson()` method (*there are also methods which connect to other data file schemas, such as XML, but this assignment will focus on JSON*). Once the `DataSource` is connected, you can begin extracting data from it. The `DataSource` class contains a useful method called `fetchArray()`, which automatically constructs instances of a class given the class name as a String and a list of identifiers to be used in the constructor (*see the code below for an example*). The following code shows an example of how you can use the `DataSource` class in your code to extract data from a JSON file and construct an array of `MyData` instances.

```
// JSON file returned by http://www.myRemoteDataSource.com/myDataRecords.json
{
    "my_data" : [ {
        "my_data_string" : "Hello",       // Identifier = "my_data/my_data_string"
        "my_data_double" : 3.1415,        // Identifier = "my_data/my_data_double"
        "my_data_timestamp" : 1460913343  // Identifier = "my_data/my_data_timestamp"
    }, {
        "my_data_string" : "World",
        "my_data_double" : 6.2831,
        "my_data_timestamp" : 1460913343
    }, {
        "my_data_string" : "CSE 214",
        "my_data_double" : 14.333,
        "my_data_timestamp" : 1460913343
    } ]
}


// MyData.java
import java.util.Date;
class MyData {
    private String myDataString;
    private double myDataDouble;
    private Date myDataDate;
    // Constructor used by fetchArray()
    public MyData(String initString, double initDouble, long initTimestamp) {
        this.myDataString = initString;
        this.myDataDouble = initDouble;
        this.myDataDate = new Date(initTimestamp);
    }
    public String toString() {
        return myDataString + ", " + myDataDouble + ", " + myDataDate.toString();
    }

}


// in main():
DataSource ds = DataSource.connectJSON("http://www.myRemoteDataSource.com/myDataRecords.json");
ds.load();

// Constructs and returns an array of 3 MyData instances.
// fetchArray() takes the name of the class as a String as the first parameter, then a
// list of identifiers showing it where to find the values for the constructor parameters.
MyData[] myDatas = ds.fetchArray(
    "MyData",                     // Name of the class as a String.
    "my_data/my_data_string",     // Identifier for constructor parameter 1 (initString).
    "my_data/my_data_double",     // Identifier for constructor parameter 2 (initDouble).
    "my_data/my_data_timestamp"   // Identifier for constructor parameter 3 (initTimestamp).
);

for(int i = 0; i < myDatas.length; ++i) {
    System.out.println(myDatas[i].toString());
}

// Output
Hello, 3.1415, Sun Apr 16 17:15:43 EET 2016
World, 6.2831, Sun Apr 16 17:15:43 EET 2016
CSE 214, 14.333, Sun Apr 16 17:15:43 EET 2016
```

Alternative method to read from JSON files:

```
// in main():
import org.json.JSONException;
import org.json.JSONObject;
import org.json.JSONTokener;
import java.net.URL;
import java.io.*;

...

String req = "http://www.myRemoteDataSource.com/myDataRecords.json";
try {
    URL getRequest = new URL(req); //Creates a URL object from the URL string
    JSONTokener tokener = new JSONTokener(getRequest.openStream());
    JSONObject root = new JSONObject(tokener);

    //look into the JSONObject methods to figure out how to extract each piece of the JSON

    //prints all the NearEarthObjects as a string
    System.out.println(root.getJSONArray("near_earth_objects").toString());

} catch(IOException ex) {
      ...
} catch(JSONException ex) {
      ...
}
```

Input Format:

Each menu operation is entered on its own line and should be case insensitive (i.e. q and Q are the same). If the user selects `S` to sort the database, the user should be prompted to select the sort key using the indicated sub-menu (see sample I/O for examples).

The server accepts requests using a query convention over traditional HTTP GET requests. Requests can be made using the following form:

`https://api.nasa.gov/neo/rest/v1/neo/browse?page=0&api_key=DEMO_KEY`

The request above asks for page number `0` using the `DEMO_KEY` (*your program should replace `DEMO_KEY` with your specific `API_KEY`*). This query returns a JSON file contining an array of NEO records formatted in the following manner (*Note that unused fields have been filtered out*). Keep in mind that you do not need to read this file directly - `BigData` will do that for you. It is important however to understand where the information is coming form, so that you may better understand how the program works.

```
{
    "links" : {
        // Link info (ignore)...
    },
    "page" : {
        // Page info (ignore)...
    }
    "near_earth_objects" : [ { // Array of NEO records.
        "neo_reference_id" : "3644696,"
        "name" : "(2016 DK)",
        "absolute_magnitue_h" : 22.4.
        "estimated_diameter" : {
            "kilometers" : {
                "estimated_diameter_min" : 0.0880146521,
                "estimated_diameter_max" : 0.1968067451
            }
        },
        "is_potentially_hazardous_asteroid" : false,
        "close_approach_data" : [ {
            "epoch_date_close_approach" : 1455696000000,
            "miss_distance" : {
                "kilometers" : "13781428"
            },
            "orbiting_body" : "Earth"
        }],

    }, {
        // Next NEO record.
    }, // ...

    ]
}
```

Output Format:

All lists must be printed in a nice and tabular form as shown in the sample output. You may use C style formatting as shown in the following example. The example below shows two different ways of displaying the name and address at pre-specified positions 21, 26, 19, and 6 spaces wide. If the '-' flag is given, then it will be left-justified (padding will be on the right), else the region is right-justified. The 's' identifier is for strings, the 'd' identifier is for integers. Giving the additional '0' flag pads an integer with additional zeroes in front.

```
String name = "Doe Jane";
String address = "32 Bayview Dr.";
String city = "Fishers Island, NY";
int zip = 6390;

System.out.println(String.format("%-21s%-26s%19s%06d", name, address, city, zip));
System.out.printf("%-21s%-26s%19s%06d", name, address, city, zip);

// Output
Doe Jane             32 Bayview Dr.            Fishers Island, NY 06390
Doe Jane             32 Bayview Dr.            Fishers Island, NY 06390
```

Sample Input/Output:

```
// Comment in green, input in red, output in black

Welcome to NEO Viewer!

Option Menu:
  A) Add a page to the database
  S) Sort the database
  P) Print the database as a table.
  Q) Quit

Select a menu option: A

Enter the page to load: 1

Page loaded successfully!

// menu not shown in the sample input/output
Select a menu option: P

 ID   |           Name            | Mag. | Diameter | Danger | Close Date | Miss Dist | Orbits
==============================================================================================
2001981  1981 Midas (1973 EA)        15.3    3.780     true     04-04-1902   56101523    Earth
2002059  2059 Baboquivari (1963 UA)  16.0    2.739     false    10-24-1907   54636908    Earth
2002061  2061 Anza (1960 UA)         16.5    2.175     false    08-18-1926   33258270    Earth
 ... // more records are included. Print the first 26 letters of Name. Keep all spacing exactly the same.

// menu not shown in the sample input/output
Select a menu option: S

R) Sort by referenceID
D) Sort by diameter
A) Sort by approach date
M) Sort by miss distance

Select a menu option: D

Table sorted on diameter.

// menu not shown in the sample input/output
Select a menu option: P

 ID   |           Name            | Mag. | Diameter | Danger | Close Date | Miss Dist | Orbits
==============================================================================================
2002061  2061 Anza (1960 UA)         16.5    2.175     false    08-18-1926   33258270    Earth
2002059  2059 Baboquivari (1963 UA)  16.0    2.739     false    10-24-1907   54636908    Earth
2001981  1981 Midas (1973 EA)        15.3    3.780     true     04-04-1902   56101523    Earth
 ... // more records are included. Print the first 26 letters of Name. Keep all spacing exactly the same.

// menu not shown in the sample input/output
Select a menu option: S

R) Sort by referenceID
D) Sort by diameter
A) Sort by approach date
M) Sort by miss distance

Select a menu option: A

Table sorted on approach date.

// menu not shown in the sample input/output
Select a menu option: P

 ID   |           Name            | Mag. | Diameter | Danger | Close Date | Miss Dist | Orbits
==============================================================================================
2001981  1981 Midas (1973 EA)        15.3    3.780     true     04-04-1902   56101523    Earth
2002059  2059 Baboquivari (1963 UA)  16.0    2.739     false    10-24-1907   54636908    Earth
2002061  2061 Anza (1960 UA)         16.5    2.175     false    08-18-1926   33258270    Earth
 ... // more records are included. Print the first 26 letters of Name. Keep all spacing exactly the same.

// menu not shown in the sample input/output
Select a menu option: S

R) Sort by referenceID
D) Sort by diameter
A) Sort by approach date
M) Sort by miss distance

Select a menu option: M

Table sorted on miss distance.

// menu not shown in the sample input/output
Select a menu option: P

 ID   |           Name            | Mag. | Diameter | Danger | Close Date | Miss Dist | Orbits
==============================================================================================
2002061  2061 Anza (1960 UA)         16.5    2.175     false    08-18-1926   33258270    Earth
2002059  2059 Baboquivari (1963 UA)  16.0    2.739     false    10-24-1907   54636908    Earth
2001981  1981 Midas (1973 EA)        15.3    3.780     true     04-04-1902   56101523    Earth
 ... // more records are included. Print the first 26 letters of Name. Keep all spacing exactly the same.
// menu not shown in the sample input/output
Select a menu option: Q

Program terminating normally...
```
