# Stargazers

Ryan Nett, Wilson Leong, and Andrew Pirondini

CPE 365: Intro to Databases

Final Class Project

June 8th, 2018

# Team Information

**Team Name:** Stargazers
**Team Members:** Ryan Nett, Wilson Leong, and Andrew Pirondini

## Log File

The team log file for tasks completed during the project:

| **Task Description** | **Time (hours)** | **Completed By** |
|---|---|---|
| Gathering Data | Completed before project. | Ryan |
| Parsing Data | 3 hours | Ryan |
| Making the Database | 2 hours | Ryan |
| Java data classes | 2 hours | Ryan |
| Frontend Design | 6 hours | Andrew |
| Event Handlers | 2 hours | Andrew |
| Backend and database access | 6 hours | Wilson |
| Integration | 2 hours | Ryan |
| JDBC Connections & SSH Tunneling | 1 hour | Ryan & Wilson |
| Written Report | 1 hour | Ryan & Wilson |
| Presentation | 1 hour | Ryan & Wilson |

# Project Goal

Import and display Kepler exoplanet data, giving users the ability to use useful filters to find interesting solar systems.

# Installation and Usage Instructions

This section contains information about the different stages of the project an running to executables on the provided data.

## Use

Run the jar file.
To import data, click the blue “Import External Data” button in the top right corner.  Select the planets.csv file or another file downloaded from https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=planets.  Make sure that the data file you use has the same columns as the planets.csv file.

```
#0 id
#1  COLUMN pl_hostname:    Host Name
#2  COLUMN pl_letter:      Planet Letter
#3  COLUMN pl_discmethod:  Discovery Method
#4  COLUMN pl_pnum:        Number of Planets in System
#5  COLUMN pl_orbper:      Orbital Period [days]
#6  COLUMN pl_orbsmax:     Orbit Semi-Major Axis [AU])
#7  COLUMN pl_orbeccen:    Eccentricity
#8  COLUMN pl_orbincl:     Inclination [deg]
#9  COLUMN pl_bmassj:      Planet Mass or M*sin(i)[Jupiter mass]
#10 COLUMN pl_bmassprov:   Planet Mass or M*sin(i) Provenance
#11 COLUMN pl_radj:        Planet Radius [Jupiter radii]
#12 COLUMN pl_dens:        Planet Density [g/cm**3]
#13 COLUMN pl_ttvflag:     TTV Flag
#14 COLUMN pl_kepflag:     Kepler Field Flag
#15 COLUMN pl_k2flag:      K2 Mission Flag
#16 COLUMN pl_nnotes:      Number of Notes
#17 COLUMN ra_str:         RA [sexagesimal]
#18 COLUMN ra:             RA [decimal degrees]
#19 COLUMN dec_str:        Dec [sexagesimal]
#20 COLUMN dec:            Dec [decimal degrees]
#21 COLUMN st_dist:        Distance [pc]
#22 COLUMN st_optmag:      Optical Magnitude [mag]
#23 COLUMN st_optband:     Optical Magnitude Band
#24 COLUMN gaia_gmag:      G-band (Gaia) [mag]
#25 COLUMN st_teff:        Effective Temperature [K]
#26 COLUMN st_mass:        Stellar Mass [Solar mass]
#27 COLUMN st_rad:         Stellar Radius [Solar radii]
#28 COLUMN rowupdate:      Date of Last Update
#29 COLUMN pl_name:        Planet Name
#30 COLUMN pl_massj:       Planet Mass [Jupiter mass]
#31 COLUMN hd_name:        HD Name
#32 COLUMN hip_name:       HIP Name
#33 COLUMN st_sp:          Spectral Type
```

```
#34 COLUMN st_spstr:       Spectral Type
```

Not all of these attributes are used, but the parser is based on column indexes, so be sure to include them.

# Description of the Internals

## UI

The UI is created using JavaFX and has much of the view information stored in FXML files. It consists of one large view and controller with multiple panes. The leftmost pane contains a table that gets data from the backend about all the solar systems and adds filters for this table. These filters are handled through event handlers any time the input is changed. The middle pane shows information about whatever the selected solar system is and updates whenever a selection is changed in the left pane. The rightmost pane has a drawing to visualize the solar system with the star and several rings of planets around it.

## Backend

The backend connects to the database and runs a select query with filters. The query was fairly complex, because we want to filter by the number of goldilocks planets per star, which is not a field of stars. We had to sum the goldilocks flags for the planets of each star, and then join it (again) with planets. Instead of doing this in the query, we decided to use a view (see the Database section). We also added the other filter conditions to the where statement. This is done dynamically using lists to make filtering easier.

## Database

We only needed two tables: planets and stars. Stars's primary key is the star name, and Planet's primary key is the star name and planet letter. Planets has a foreign key to stars on the star name.

These are the create table statements:

```
create table stars (
  starName varchar(30) primary key,
   hipName varchar(30),
   class varchar(20),
   type enum('unknown', 'supergiant', 'bright giant', 'giant', 'subgiant', 'main
sequence'),
   color enum('unknown', 'blue', 'blue white', 'white', 'yellow white', 'yellow',
'light orange', 'orange red'),
   starMass real,
   starRadius real,
   temp real,
   goldilocksInner real,
```

```
   goldilocksOuter real,
   planets int,
   distance real
);

create table planets (
  starName varchar(30) not null,
   letter varchar(5) not null,
   orbitalRadius real,
   orbitalPeriod real,
   orbitalEccentricity real,
   orbitalInclination real,
   planetMass real,
   planetRadius real,
   density real,
   goldilocks bit,
   primary key (starName, letter),
   foreign key (starName) references stars(starName)
);
```

I also created a view using:

```
select * from
planets P join
(
select S1.starName, sum(P1.goldilocks) as golds from
planets P1 join stars S1 using(starName)
group by P1.starName
) G
using(starName)
join stars S using(starName);
```

The view made it significantly easier to query the database.

## Parser

The parser takes a csv file of planet data, as described in the Installation section, and creates and runs two SQL insert statements, one for the planets and one for the stars.

It calculates lumosity for each star, and then uses that to calculate the goldilocks range (where a planet can have liquid water, and thus potentially support life). Each planet is given a flag of whether it is in the goldilocks zone or not. I use two methods to calculate luminosity: using the temperature and radius of the star, and using the magnitude. If the data is available for the first method, I save it, because it is the more accurate of the two.

The star class is also converted into color and type (from a string), so that we can use enums.

All of the star and planets tuples are chained together, and added two one of the two insert queries.

# What We Learned and Project Challenges

There are lots of things to parse, and the data format was not particularly friendly.  Many planets were missing one or more values, this is because they were discovered using different methods.

During development, program will sometimes throw exceptions, but will still run in the background. Scaling the size of the star on display relative to the radius of the orbits without making the star too small for the user to see was also quite a challenge.

Connecting to the database from home was tricky, we had to tunnel through a ssh server with port forwarding.  Luckily, the Jsch library does this.  We used a lazy method for connections, allowing us to re-use a connection and to save the arguments.  We also had to save the ssh connection and close it after we were done, otherwise the program would not quit when the window was closed.

Creating the UI in JavaFX turned out to also be a bit challenging. JavaFX seems designed for much simpler views and did not have tools to handle this one complex view very well. It worked alright but the controller ended up with very cluttered code and layout changes became a large amount of work after the whole view was created. It seems like JavaFX is very good at doing simple views, such as a table with a title and a link to some other view, but much worse for the purpose we were using it for.