Solutions

Final Quiz

Study Guide / Practice Problems

Topics:

- Streams
  - **Resources:** Module 24
  - **You should know:**
    - How to convert to and from streams
    - How to reduce a stream to calculate an average
    - How to map streams and change each element in it(like in streams.zip)
- Polymorphism
  - **Resources:** module 25/27
  - **You should know:**
    - How the compiler and runtime environment will deal with:
      - Overloading
      - Overriding
      - Reference var types vs object types
    - How to complete the worksheets from module 25
- ~~Exceptions:~~
  - ~~**Resources:** Module 25~~
  - ~~**You should know:**~~
    - ~~How to trace through try / except / finally blocks~~
    - ~~How to trace through methods that have try /catch blocks and / or throw exceptions~~
    - ~~Be able to make your own exceptions~~
    - ~~Know the difference between checked and unchecked exceptions~~
- Review:
  - Quiz 2 Study Guide
    - Designing Classes and Interacting Classes
    - Object class and Overriding (toString / equals)
  - Quiz 4 Study Guide
    - Inheritance (overloading/overriding)
  - Quiz 5 Study Guide

        - Abstract Classes
    - Your projects:
        - Be able to make design choices and represent your code with UML
        - Be able to talk about your projects

Study Material:

- This practice quiz, the other practice quizzes listed above, your projects, all modules

Practice problems for new topics

1. Streams

1. Complete the following code given the fact that the Professor class is not Comparable and has the getters: `hasTenure()` which returns a `boolean` and `getMortgage()` which returns an int

```
public class Test {
      public static void main(String[] args) {
      ArrayList profs = new ArrayList<>();
      profs.add(new Professor(500000, false, "Julie", 0, 2));
      profs.add(new Professor(300000, false, "Paul", 0, 2));
      profs.add(new Professor(250001, true, "Phil", 0, 2));
      profs.add(new Professor(250000, true, "Hugh", 0, 2));
      profs.add(new Professor(200000, false, "Kurt", 0, 2));

      // TODO:
      // Write one line of code to calculate the average mortgage
      // of all professors with tenure.

 // Below -
//Several of many different ways to compute the average mortgage of
// tenured profs.
// 1. ------------------------------------------------

double avg = profs.stream().filter(Professor::hasTenure)
       .mapToDouble(Professor::getMortgage).average().getAsDouble();

// 2. ------------------------------------------------
//notice: this option you need to divide by the size of filtered list
```

```
double avg = profs.stream().filter(p -> p.hasTenure()) .mapToInt(p ->
     p.getMortgage() .sum() /
    (double)profs.stream().filter(Professor::hasTenure).count();

System.out.println(avg);
```

2. Also be able to use streams to filter and map and collect. See the lecture material from the day we covered streams for more coding examples you should be able to do.

Review:

Be able to:

- Talk about your projects
- Work with inheritance (overriding methods / pulling up common data)
- Make design choices (like project 4, lab 4) and describe with UML
- Work with polymorphism (be able to do both worksheets from module 25)

1. How would you represent a social media network with objects? Think about how to represent the fact that:
    a. A person in the system has data, including their list of friends / who they are in a relationship with / who they are related to

Think about if you will need different types of people or should just have one Person class

Some Other Practice Problems

Use the class **CelestialBody** and the interface **Orbits** to answer the questions 1-6.

```
public class CelestialBody
{
   private double mass;  // in kg
   private double velocity;
   private String name;

   public CelestialBody(double mass, double velocity, String name)
   {
      this.mass = mass;
      this.velocity = velocity;
      this.name = name;
   }
   public double mass() { return mass; }
   public double velocity() { return velocity; }
   public String name() { return name; }
}

public interface Orbits
{
   public double duration();
   public CelestialBody orbiting();
}
```

1. Write a class **Planet** that extends **CelestialBody** and implements **Orbits**. The constructor should take the mass, velocity, name, orbital duration, the celestial body being orbited, and an integer number of moons. Write the entire class. Each field of the **Planet** should have a way to be queried.

```
public class Planet extends CelestialBody implements Orbits
{
   private double duration;
   private CelestialBody cb;
   private int moons;

   public Planet(double mass, double velocity, String name,
                 double duration, CelestialBody cb, int moons)
   {
      super(mass, velocity, name);
      this.duration = duration;
      this.cb = cb;
      this.moons = moons;
   }
   public double duration() { return duration; }
   public CelestialBody orbiting() { return cb; }
   public int moons() { return moons; }
}
```

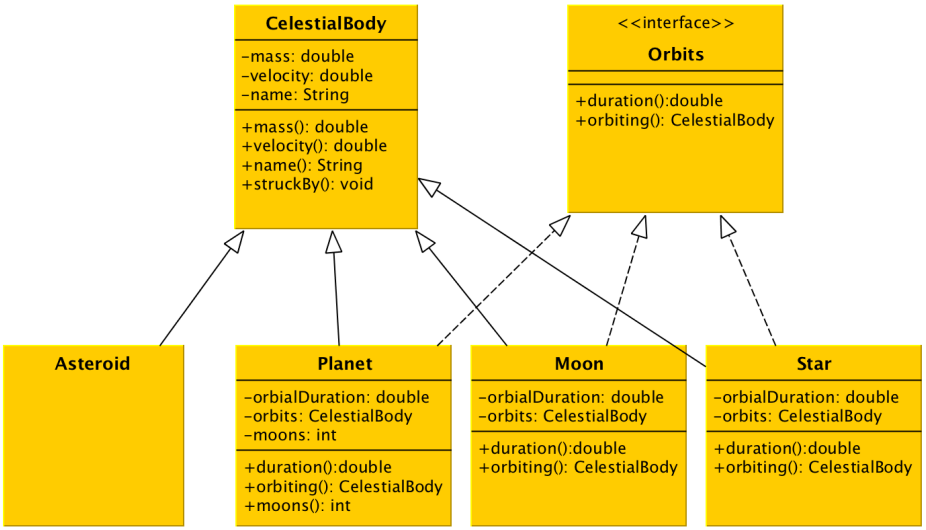
2. Override the `equals()` method from **Object** in both **CelestialBody** and **Planet**. Indicate which method should go in which class.

```
//In CelestialBody:
public boolean equals(Object other)
{
   if (other == null)
      return false;
   if (!getClass().equals(other.getClass()))
      return false;

   CelestialBody c = (CelestialBody)other;
   return mass == c.mass &&
          velocity == c.velocity &&
          name.equals(c.name);
}

//In Planet:
public boolean equals(Object other)
{
   if (!super.equals(other))
      return false;
   Planet p = (Planet)other;
   return duration == p.duration &&
          cb.equals(p.cb)) &&
          moons == p.moons;
}
```

3. Assume a **Moon** and **Star** class also extend **CelestialBody** and implement **Orbits**. Further assume that a class **Asteroid** extends **CelestialBody**. Finally assume that the **CelestialBody** class has a method `struckBy()` that takes an **Asteroid** as its input parameter. Draw a UML diagram of all the described classes and interfaces.



I'm okay if your **Moon** and **Star** don't have those instance vars listed. They technically aren't required to implement the **Orbits** interface. I just thought it made the most sense to have them.

4. Create a **Comparator<CelestialBody>** object, where larger masses should come first. Do this in 3 different ways – 1) By writing a separate class, 2) By using a Lambda expression, and 3) By using a key extractor.

```
public class CelestialBodyComp implements
Comparator<CelestialBody>
{
   public int compare(CelestialBody a, CelestialBody b)
   {
      if (a.mass() > b.mass())
         return -1;
      if (b.mass() > a.mass())
         return 1;
      return 0;
   }
}

Comparator<CelestialBody> comp = (a, b) -> {
      if (a.mass() > b.mass())
         return -1;
      if (b.mass() > a.mass())
         return 1;
      return 0;
   }

Comparator<CelestialBody> comp =
        Comparator.comparing(CelestialBody::mass).reversed();
```

5. Complete the following method that returns the total orbital duration of all the **CelestialBodies** in the input **ArrayList**. Note that not all the objects in the list orbit something. These objects should contribute nothing to the total.

```
public double totalDuration(ArrayList<CelestialBody> spaceThings)
{
   double duration = 0;
   for (CelestialBody c : spaceThings)
   {
      if (c instanceof Orbits)
         duration += ((Orbits)c).duration();
   }
   return duration;
}

// Another possible solution could use a stream
```

6. Write a method that accepts a **`List`** of **`Planet`** objects and returns a list of all **`Planet`** objects with masses between 50% and 200% of the mass of the earth (5.9736×10^{24}kg). Use a **`Stream`**.

```
public List<Planet> earthlike(List<Planet> planets)
{
   return planets.stream()
                 .filter(p -> p.mass() > 2.9868e24
                              && p.mass() < 1.19472e25)
                 .collect(Collectors.toList());
}
```