Use the class **CelestialBody** and the interface **Orbits** to answer the questions 1-10.

```
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.

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

3. Assume that it doesn't make sense to direcly instantiate an object of type **CelestialBody**. What should we change in the definition of the class?

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

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

6. 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)
{
```

7. What do you need to do to the above method so that it can accept a `List<Moon>` or a `List<Planet>` or a list of any other kind of **CelestialBody**.

8. 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 ($5.9736 \text{x} 10^{24}$kg).

9. Write a method that accepts a **`Map<Planet, List<CelestialBody>>`**. Each **`Map`** entry represents a planet and all celestial bodies that orbit that planet. The method also accepts a **`CelestialBody`** object and will return a **`List`** of all the **`Planet`** objects that are orbited by the **`CelestialBody`**.

10. Consider the following code:

```
public class Meteor extends CelestialBody
{
   public Meteor(double mass, double velocity)
   {
      super(mass, velocity, null)
   }
   @Override
   public String name() { throw RuntimeException("not implemented"); }
}
```

What object-oriented design principle does this subclass violate?

Exceptions - Examine the following Java code and fill out the table below by completing the output to System.out produced in each scenario or Stack Trace if the scenario causes a stack trace to print to the screen. Note that neither ***NullPointerException*** or ***IllegalArgument*** exception is a sub-class of the other.

```
System.out.print("A, ");
methodOne();
System.out.print("B, ");

try {
   System.out.print("C, ");
   methodTwo(x);
   System.out.print("D, ");
}
catch (IllegalArgumentException e) {
   System.out.print("E, ");
}
catch (Exception e){
   System.out.print("F, ");
   return;
} finally {
   System.out.print("G, ");
}
System.out.print("H, ");
```

Scenario	**Description**	**Write the output and "Stack Trace" if one is printed.**
1	Call to ***methodOne*** throws a NullPointerException and, if called, ***methodTwo*** throws a NullPointerException	
2	Call to ***methodOne*** completes normally and, if called, ***methodTwo*** completes normally	
3	Call to ***methodOne*** throws an IllegalArgumentException and, if called, ***methodTwo*** completes normally	
4	Call to ***methodOne*** completes normally and, if called, ***methodTwo*** throws an IllegalArgumentException	
5	Call to ***methodOne*** completes normally and, if called, ***methodTwo*** throws a NullPointerException	

Design – Refactor the following code to minimize code duplication, using the Template Method design pattern. You may modify it in place.

```
abstract public class WritingInstrument {

    abstract public void writeNovelWith(Author author, Paper paper);

}

public class Pen extends WritingInstrument {

    @Override
    public void writeNovelWith(Author author, Paper paper) {
        author.refillInkIfNeeded(this);
        author.createCharacters(this, paper);
        author.writeIntroduction(this, paper);
        author.developPlot(this, paper);
        author.writeMiddle(this, paper);
        author.writeCliffhangerEnding(this, paper);
    }

}
```

```
public class Pencil extends WritingInstrument {

    @Override
    public void writeNovelWith(Author author, Paper paper) {
        author.sharpen(this);
        author.createCharacters(this, paper);
        author.writeIntroduction(this, paper);
        author.developPlot(this, paper);
        author.writeMiddle(this, paper);
        author.writeCliffhangerEnding(this, paper);
    }















}

public class Author {
    public void sharpen(Pencil pencil) { ... }
    public void refillInkIfNeeded(Pen pen) { ... }
    public void createCharacters(WritingInstrument w, Paper p) { ... }
    public void writeIntroduction(WritingInstrument w, Paper p) {
        ...
    }
    public void developPlot(WritingInstrument w, Paper p) { ... }
    public void writeMiddle(WritingInstrument w, Paper p) { ... }
    public void writeCliffhangerEnding(WritingInstrument w, Paper p) {
        ...
    }
}
```