Problem 1

1. Classify each public method of RatNum as either a creator, observer, producer, or mutator
    - public RatNum(int n): **creator**
    - public RatNum(int n, int d): **creator**
    - public boolean isNaN(): **observer**
    - public boolean isNegative(): **observer**
    - public boolean isPositive(): **observer**
    - public int compareTo(RatNum rn): **observer**
    - public double doubleValue(): **observer**
    - public int intValue(): **observer**
    - public float floatValue(): **observer**
    - public long longValue(): **observer**
    - public RatNum negate(): **producer**
    - public RatNum add(RatNum arg): **producer**
    - public RatNum sub(RatNum arg): **producer**
    - public RatNum mul(RatNum arg): **producer**
    - public RatNum div(RatNum arg): **producer**
    - public int hashCode(): **observer**
    - public boolean equals(/*@Nullable*/ Object obj): **observer**
    - public String toString(): **observer**
    - public static RatNum valueOf(String ratStr): **producer**

2. add, sub, mul, and div all require that arg != null. This is because all of these methods access fields of arg without checking if arg is null first. But these methods also access fields of this without checking for null; why is this != null absent from the requires clause for these methods?

   Verifying whether "this" is null using the clause "this != null" is redundant because "this" has already been defined as either NaN, ZERO, or a RatNum, and none of the methods provided have the ability to convert "this" into null. Therefore, it is unnecessary to check for null.

3. Why is RatNum.valueOf(String) a static method? What alternative to static methods would allow someone to accomplish the same goal of generating a RatNum from an input String?

   To enable calling the function without an existing RatNum object, it is defined as static. An alternative approach could be to implement a new constructor that accepts a string input and performs parsing in a similar fashion.

4. add, sub, mul, and div all end with a statement of the form return new RatNum (numerExpr , denomExpr);. Imagine an implementation of the same function except the last statement is:

this.numer = numerExpr;
this.denom = denomExpr;
return this;

The absence of @modifies statements in the functions means that altering any of the variables in the function would violate the functions' specifications. Additionally, making changes to "numer" and "denom" contradicts the class specifications of RatNum, which requires it to be immutable. Thus, such modifications make "numer" and "denom" mutable objects.

5. Calls to checkRep are supposed to catch violations in the classes' invariants. In general, it is recommended to call checkRep at the beginning and end of every method. In the case of RatNum, why is it sufficient to call checkRep only at the end of constructors? (Hint: could a method ever modify a RatNum such that it violates its representation invariant? Could a method change a RatNum at all? How are changes to instances of RatNum prevented?)

   The reason why it is enough is that RatNum is declared as immutable, meaning that the variables' values remain the same after being initialized by the constructors. Whether they are initialized correctly or not, they will not undergo any modifications in any of the methods. Additionally, the use of "final" for the variable values provides an additional layer of protection against any changes.