1 Problem 1

1.1 Classify each public method of RatNum as either a creator, observer, producer, or mutator.

Creators:

- `public RatNum(int n)`
- `RatNum(int n, int d)`
- `public static RatNum valueOf(String ratStr)`

Observers:

- `public boolean isNaN()`
- `public boolean isNegative()`
- `public boolean isPositive()`
- `@Override public int compareTo(RatNum rn)`
- `@Override public double doubleValue()`
- `@Override public int intValue()`
- `@Override public float floatValue()`
- `@Override public long longValue()`
- `@Override public int hashCode()`
- `@Override public boolean equals(/*@Nullable*/ Object obj)`
- `@Override public String toString()`

Producers:

- `public RatNum negate()`
- `RatNum add(RatNum arg)`
- `sub(RatNum arg)`
- `mul(RatNum arg)`
- `RatNum div(RatNum arg)`

Mutators: None (states in lecture that Poly is an immutable class, so it has no mutators)

1.2 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?

These methods directly access fields of the argument (RatNum) without prior null checks since the input is anticipated to be fully formed, and none of the constructors introduce nullable values. Therefore, it's unlikely for individual fields within the object to be null without the entire object being null itself.

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

The static modifier is applied to the valueOf(String) method because it operates solely on the input string to produce a new RatNum instance, without relying on any instance-specific information from the class. An alternative approach to using class methods is to create a utility class with equivalent functionality.

1.4 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;
```

For this question, pretend that the this.numer and this.denom fields are not declared as final so that these assignments compile properly. How would the above changes fail to meet the specifications of the function (hint: take a look at the @requires and @modifies clauses, or lack thereof) and fail to meet the specifications of the RatNum class?

The modifications wouldn't meet the function's requirements because both the @requires and @modifies clauses specify that 'this' and 'arg' must not be null. Likewise, they wouldn't meet the RatNum class specifications due to the same conditions. Also the @modifies would also need additional statements stating that we are modifying the numer and denom.

1.5 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?)

Calling checkRep() only at the end of constructors is enough because the RatNum class is immutable. This immutability stems from the fact that the fields are final and the methods do not alter them. The purpose of invoking checkRep() at the end of constructors is to guarantee that the RatNum instance is initialized with the accurate values. When we do change values of RatNum, we are actually creating a new RatNum object with the new values and returning that. This is why we can call checkRep() at the end of the constructors and not have to worry about it in the other methods.