PSOFT hw3 problem 1

Brian Wang-Chen

February 2024

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 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() : producers
   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(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

   No need to check if this != null because the current instance RatNum object is already ensured to not be null from our constructor, otherwise it would have thrown a exception when the method runs

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? for these methods?

   RatNum.valueOf(string) is a class method with a static modifier because valueOf does not access a instance variable of RatNum. A alternative would to create a RatNum object using a new constructor that takes in a input string

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?

   arithmetic functions specifies they are returning a RatNum after the performed operation. New implementation fails the specification as it is only modifying the instance variables of RatNum and not returning a new RatNum, which also cannot be done as their are no mutators, so the instance variables are final once intialized.

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?)

There are no mutators in RatNum so once a RatNum object is initialized, the instance variables are final. There are producer methods that produce new RatNum objects but never modify the original object therefore there is no need to have checkrep at the beginning.