6.170 Laboratory in Software Engineering
Spring 2005
Problem Set 1: Java and Coding to Specification
Due: Thursday, September 22, 2005 at 1:00 PM

Handout P1

Quick links:

- Specifications
- Provided source code (in the assignments section)
- Problem set submission (in the assignments section)

Contents:

We recommend that you read the entire problem set before you begin work.

Introduction

In this problem set, you will practice reading and interpreting specifications, as well as reading and writing Java source code. You will implement a pair of classes that will complete the implementation of a graphing polynomial calculator, and you will answer questions about both the code you are given and the code you have written.

To complete this problem set, you will need to know:

1. Basic algebra (rational and polynomial arithmetic)
2. How to read and write basic Java code

    - code structure and layout (class and method definition, field and variable declaration)
    - method calls
    - operators for:
        - object creation: `new`
        - field and method access: `.`
        - assignment: `=`
        - comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
        - arithmetic: `+`, `-`, `*`, `/`
    - control structure: loops (`while` and `for`) and conditional branches (`if`, `else`)
3. How to read procedural specifications (requires, modifies, effects)

Getting started

Checkout the `ps1` module from your repository. If you don't remember how to do this, take a look at the Problem Set Procedure document. Then, work through the problems below. Use the provided Specifications to help you fill in the missing methods.

By the end of the problem set, you should have the following files ready to submit in your ps1 directory:

- `doc/problem1.txt`
- `src/RatPoly.java`
- `src/RatPolyStack.java`
- `doc/problem4.txt`
- `doc/reflection.txt`

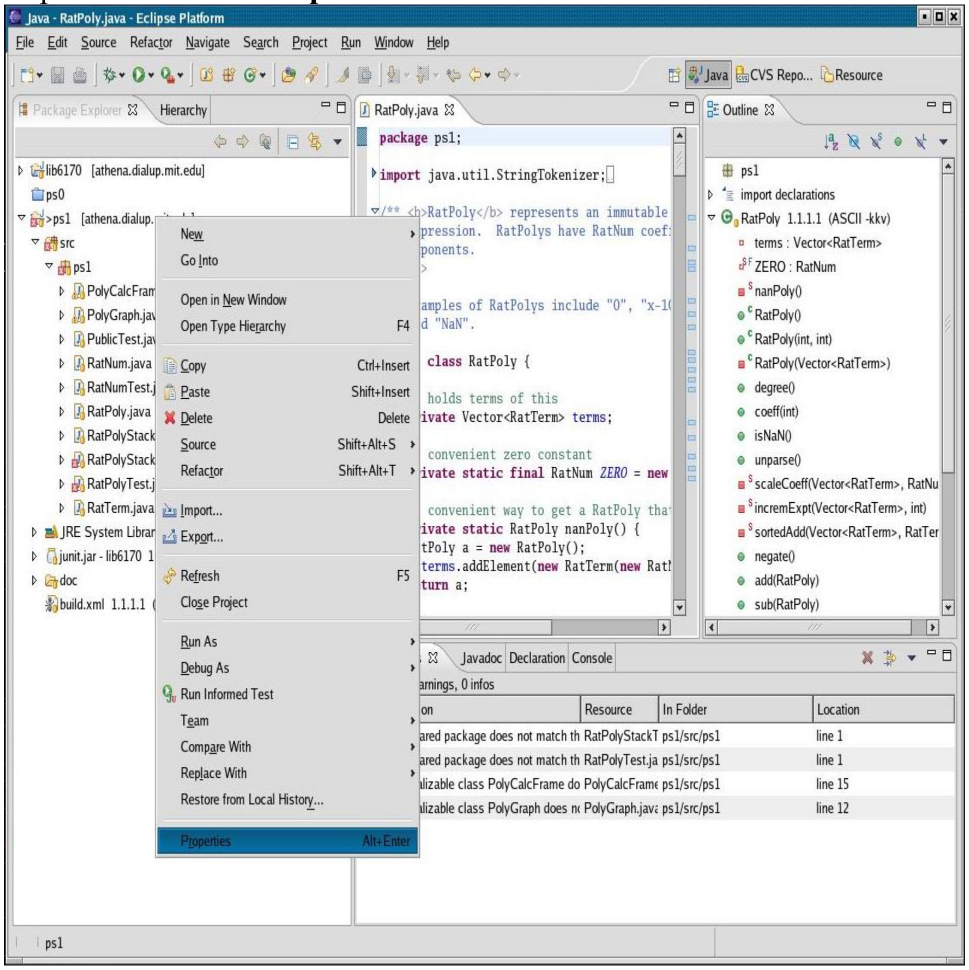
You are free to work from home if that makes your life easier, but keep in mind that your final code MUST run correctly on Athena. Once everything is in order, read the Problem Set Submission instructions, which will tell you how to run a final check of your code and turn it in.

Introducing Continuous Testing

In Problem Set 0, your were introduced to JUnit testing. In this problem set, we will introduce another tool that will make testing easier and more effective -- the Eclipse Continuous Testing plugin. This tool is already installed on Athena and all you need to do is to enable it.

Continuous testing builds on the automated developer support in Eclipse to make it even easier to keep your Java code well-tested. With continuous testing enabled, as you edit your code, Eclipse runs your tests quietly in the background, and notifies you if any of them fail or cause errors. It is most useful in situations where you would already have a test suite while you are changing code: when performing maintenance, refactoring, or using test-first development.

1. To enable Continuous Testing, right click on the project folder in the Package Explorer and select **Properties.**



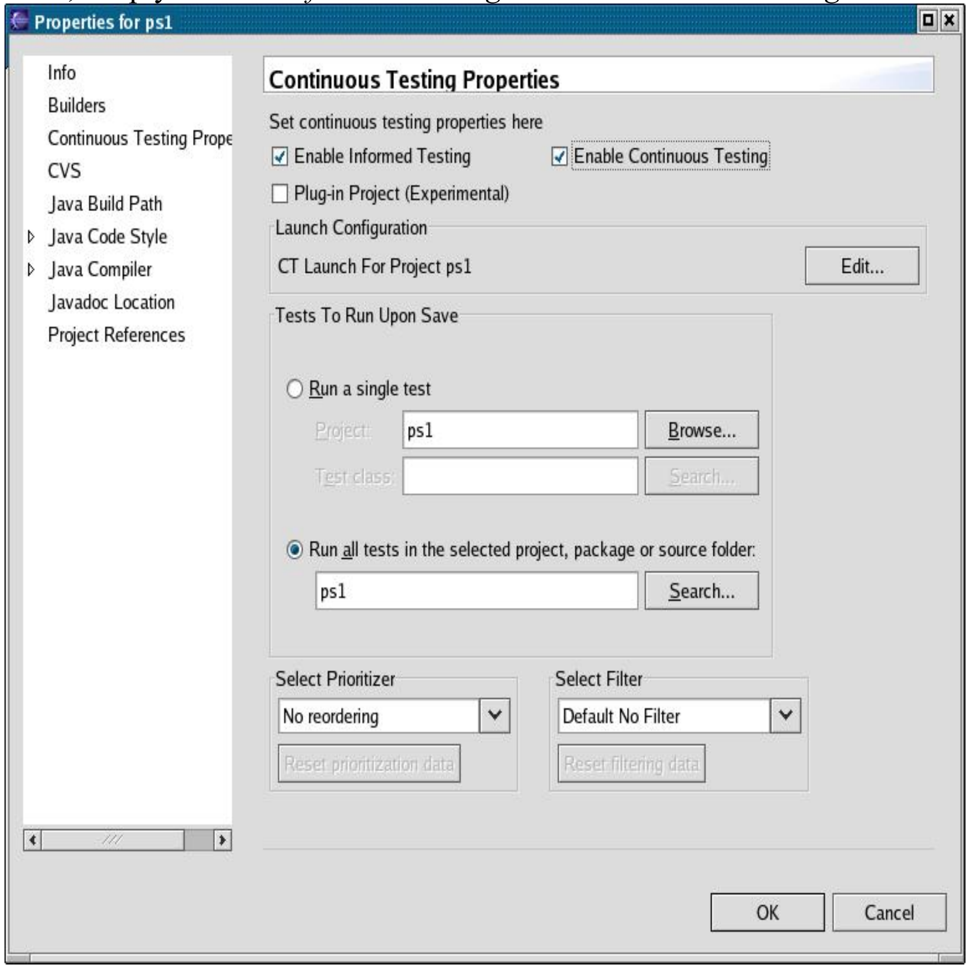
```
package ps1;

import java.util.StringTokenizer;

/** <b>RatPoly</b> represents an immutable
 pression.  RatPolys have RatNum coef
 ponents.
 >

 amples of RatPolys include "0", "x-1
 d "NaN".

 class RatPoly {

 holds terms of this
ivate Vector<RatTerm> terms;

 convenient zero constant
ivate static final RatNum ZERO = new

 convenient way to get a RatPoly tha
ivate static RatPoly nanPoly() {
tPoly a = new RatPoly();
terms.addElement(new RatTerm(new Rat
turn a;
```

arnings, 0 infos

on	Resource	In Folder	Location
ared package does not match th	RatPolyStackT	ps1/src/ps1	line 1
ared package does not match th	RatPolyTest.ja	ps1/src/ps1	line 1
lizable class PolyCalcFrame do	PolyCalcFram	ps1/src/ps1	line 15
lizable class PolyGraph does n	PolyGraph.jav	ps1/src/ps1	line 12

Courtesy of The Eclipse Foundation.

2. Next, simply enable *"Informed Testing"* and *"Continuous Testing"* and click **OK**.



Properties for ps1

Continuous Testing Properties

Set continuous testing properties here

[✓] Enable Informed Testing [✓] Enable Continuous Testing

[ ] Plug-in Project (Experimental)

Launch Configuration: CT Launch For Project ps1

Tests To Run Upon Save

( ) Run a single test
Project: ps1
Test class:

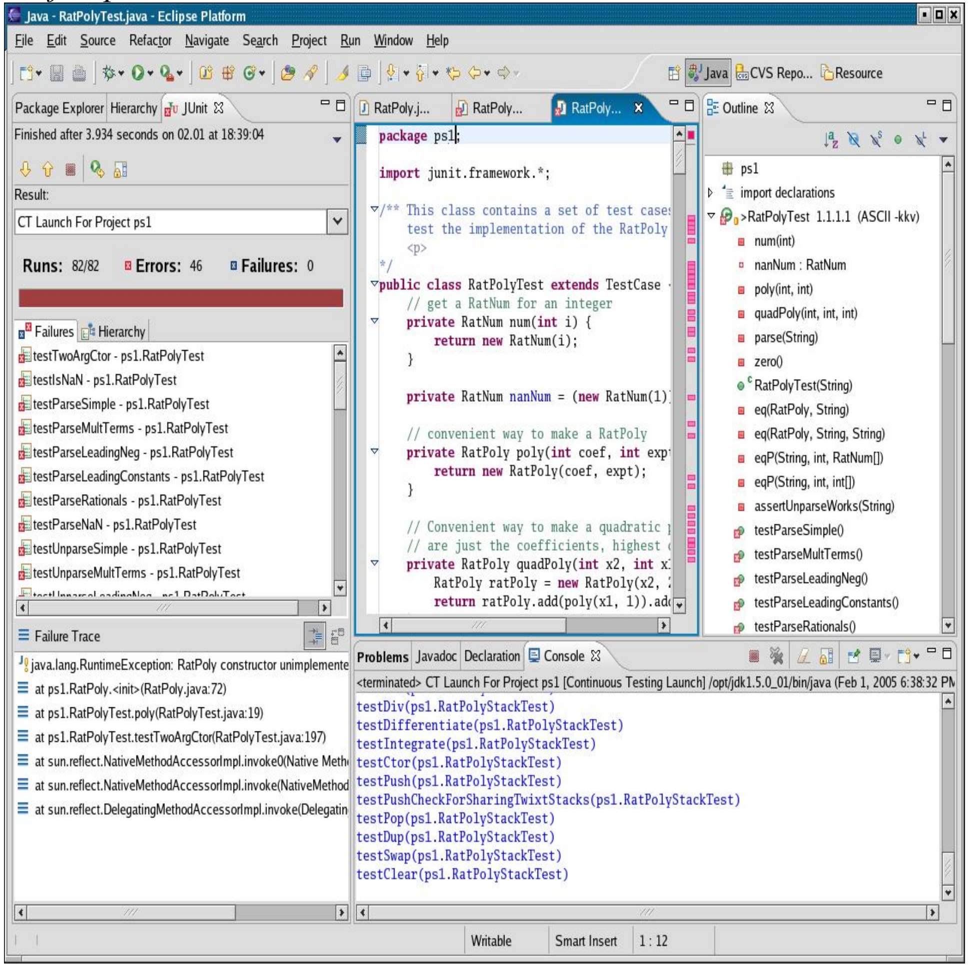
(●) Run all tests in the selected project, package or source folder: ps1

Select Prioritizer: No reordering

Select Filter: Default No Filter

Courtesy of The Eclipse Foundation.

3. When you next click on the **JUnit** view, you will see a test called *CT Launch for Project ps1*.


Java - RatPolyTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java CVS Repo... Resource
Package Explorer Hierarchy JUnit
RatPoly.j... RatPoly... RatPoly...
Outline
Failures Hierarchy
Failure Trace
Problems Javadoc Declaration Console
Writable Smart Insert 1 : 12

Finished after 3.934 seconds on 02.01 at 18:39:04

Result: CT Launch For Project ps1

Runs: 82/82 Errors: 46 Failures: 0

```
testTwoArgCtor - ps1.RatPolyTest
testIsNaN - ps1.RatPolyTest
testParseSimple - ps1.RatPolyTest
testParseMultTerms - ps1.RatPolyTest
testParseLeadingNeg - ps1.RatPolyTest
testParseLeadingConstants - ps1.RatPolyTest
testParseRationals - ps1.RatPolyTest
testParseNaN - ps1.RatPolyTest
testUnparseSimple - ps1.RatPolyTest
testUnparseMultTerms - ps1.RatPolyTest
```

```
java.lang.RuntimeException: RatPoly constructor unimplemente
at ps1.RatPoly.<init>(RatPoly.java:72)
at ps1.RatPolyTest.poly(RatPolyTest.java:19)
at ps1.RatPolyTest.testTwoArgCtor(RatPolyTest.java:197)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Meth
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethod
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Delegatin
```

```java
package ps1;

import junit.framework.*;

/** This class contains a set of test cases
    test the implementation of the RatPoly
    <p>
*/
public class RatPolyTest extends TestCase
    // get a RatNum for an integer
    private RatNum num(int i) {
        return new RatNum(i);
    }

    private RatNum nanNum = (new RatNum(1)

    // convenient way to make a RatPoly
    private RatPoly poly(int coef, int exp
        return new RatPoly(coef, expt);
    }

    // Convenient way to make a quadratic
    // are just the coefficients, highest
    private RatPoly quadPoly(int x2, int x
        RatPoly ratPoly = new RatPoly(x2,
        return ratPoly.add(poly(x1, 1)).ad
```

```
ps1
import declarations
>RatPolyTest 1.1.1.1 (ASCII -kkv)
    num(int)
    nanNum : RatNum
    poly(int, int)
    quadPoly(int, int, int)
    parse(String)
    zero()
    RatPolyTest(String)
    eq(RatPoly, String)
    eq(RatPoly, String, String)
    eqP(String, int, RatNum[])
    eqP(String, int, int[])
    assertUnparseWorks(String)
    testParseSimple()
    testParseMultTerms()
    testParseLeadingNeg()
    testParseLeadingConstants()
    testParseRationals()
```

```
<terminated> CT Launch For Project ps1 [Continuous Testing Launch] /opt/jdk1.5.0_01/bin/java (Feb 1, 2005 6:38:32 PM
testDiv(ps1.RatPolyStackTest)
testDifferentiate(ps1.RatPolyStackTest)
testIntegrate(ps1.RatPolyStackTest)
testCtor(ps1.RatPolyStackTest)
testPush(ps1.RatPolyStackTest)
testPushCheckForSharingTwixtStacks(ps1.RatPolyStackTest)
testPop(ps1.RatPolyStackTest)
testDup(ps1.RatPolyStackTest)
testSwap(ps1.RatPolyStackTest)
testClear(ps1.RatPolyStackTest)
```

Courtesy of The Eclipse Foundation.

Continuous testing will automatically run all the JUnit tests in the problem set and highlight all the tests that failed. You can also run specific tests like you did in Problem Set 0. The results for these tests can be accessed separately in the **Result** dropdown and Continuous Testing will not override them. Continuous Testing tries to use only spare cycles on your machine when you don't need them and should in general not have any effects on Eclipse's performance. It is possible however that for the older slower Athena machines, you might experience a noticeable slowdown in performance. If so, simply uncheck the two check boxes *"Informed Testing"* and *"Continuous Testing"* to disable Continuous Testing.

Problems

Problem 1: RatNum (24 points)

Read the specifications for **RatNum**, a class representing rational numbers. Then read over the staff-provided implementation, `RatNum.java`.
You may find it helpful to peruse the code in `RatNumTest.java` to see example usages of the RatNum class (albeit in the context of a test driver, rather than application code).

Answer the following questions, writing your answers in the file `doc/problem1.txt`

1. What is the point of the one-line comments inside the `add`, `sub`, `mul`, and `div` methods (right before the call to `checkRep()`)?
2. `add`, `sub`, `mul`, and `div` all require that "arg != null". This is because all of the methods access fields of 'arg' without checking if 'arg' is null first. But the methods also access fields of 'this' without checking for null; why is "this != null" absent from the requires-clause for the methods?
3. `RatNum.div(RatNum)` checks whether its argument is NaN (not-a-number). `RatNum.add(RatNum)` and `RatNum.mul(RatNum)` do not do that. Explain.
4. Why is `RatNum.parse(String)` a static method? What alternative to static methods would allow one to accomplish the same goal of generating a RatNum from an input String?
5. The `checkRep()` method is called at the beginning and end of most methods. Why is there no `checkRep()` call at the beginning of the constructor?
6. Imagine that the representation invariant were weakened so that we did not require that the numer and denom fields be stored in reduced form. This means that the method implementations could no longer assume this invariant held on entry to the method, but they also no longer were required to enforce the invariant on exit. The new rep invariant would then be:

   // Rep Invariant for every RatNum r: (r.denom >= 0)

   Which method or constructor implementations would have to change? Please list them. For each changed piece of code, describe the changes informally, and indicate how much more or less complex the result would be (both in terms of code clarity, and also in terms of execution efficiency). Note that the new implementations must still adhere to the given spec; in particular, `RatNum.unparse()` needs to output fractions in reduced form.

7. `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;
   ```

Give two reasons why the above changes would not meet the specifications of the function. (Hint: take a look at the @requires and @modified statements, or lack thereof.)

Problem 2: RatPoly (45 points)

Read over the specifications for the **RatTerm** and **RatPoly** classes. Make sure that you understand the overview for RatPoly and the specifications for the given methods.

You may also want to take a look at the specifications for the **java.util.Vector** class, especially the `get()`, `set()`, `insertElementAt()`, and `size()` methods.

Read through the provided skeletal implementation of `RatPoly.java` , especially the comments describing how you are to use the provided fields to implement this class. The Representation Invariant is an especially important comment to understand, because the invariants you define can have a drastic effect on which implementations will be legal for the methods of RatPoly.

Fill in an implementation for the methods in the specification of RatPoly. You may define new private helper methods as you like; we have suggested a few ourselves, complete with specifications, but you are not obligated to use them. (We believe that doing so will drastically simplify your implementation.) You may not add public methods; the external interface must remain the same.

We have provided a `checkRep()` method in RatPoly that tests whether or not a RatPoly instance violates the representation invariant. We highly recommend you use `checkRep()` in the code you write when implementing the unfinished methods. Take a look at the RatPoly.degree() function for a simple way to include `checkRep()` in your code. Simply place the entire main body of your code inside a `try { ... }` block, and append

```
finally {
    checkRep();
}
```

to your code.

We have provided a fairly rigorous test suite in `RatPolyTest.java`. You can run the given test suite with JUnit as you program and evaluate your progress and the correctness of your code.

The test suite depends heavily on the implementation of the `RatPoly.unparse()` method. Therefore, you should implement `unparse()` first, because you will not be able to do any testing until you do. Furthermore, you need to get `unparse()` correct, because errors in `unparse()` can be very confusing or misleading while you are debugging other methods. If the test suite claims that all or many of the tests are failing, the source of the problem could be a bug in your implementation of `unparse()`.

If you need help running test suites, take a look at the Problem Set Procedure document.

Problem 3: RatPolyStack (30 points)

Follow the same procedure given in Problem 2, but this time fill in the blanks for `RatPolyStack.java`. The same rules apply here (you may add private helper methods as you like). Since this problem depends on problem 2, you should not begin it until you have completed problem 2 (and the `ps1.RatPolyTest` test suite runs without any errors).

Make sure your code passes all the tests in `RatPolyStackTest.java`.

Problem 4: PolyCalc (1 point)

Now that you have implemented the two remaining classes in the system, you can run the PolyCalc application. This allows you to input polynomials and perform arithmetic operations on them, through a point-and-click user interface. The calculator graphs the resulting polynomials as well.

When you run ps1.PolyCalcFrame, a window will pop up with a stack on the left, a graph display on the right, a text area to input polynomials into, and a set of buttons along the bottom. Click the buttons to input polynomials and to perform manipulations of the polynomials on the stack. The graph display will update on the fly, graphing the top four elements of the stack.

Submit your four favorite polynomial equations, in the `RatPoly.unparse` format, in the file `doc/problem4.txt`.

Reflection (1 point)

Please answer the following questions in a file named `reflection.txt` in your `doc/` directory.

1. How many hours did you spend on each problem of this problem set?
2. In retrospect, what could you have done better to reduce the time you spent solving this problem set?
3. What could the 6.170 staff have done better to improve your learning experience in this problem set?

Provided classes

The following classes are all provided for you.

With source code also provided:

ps1.RatNum
ps1.RatNumTest
ps1.RatPolyTest
ps1.RatPolyStackTest
ps1.Cons (as part of the `RatPolyStack.java` starter code)
ps1.PublicTest
ps1.PolyGraph
ps1.PolyCalcFrame
ps1.RatTerm
ps1.RatTermVec

Hints

If you're having trouble, the Forums are a good place to look for help.

All of the unfinished methods in RatPoly and RatPolyStack throw RuntimeExceptions. When you implement a method, you should be sure to remove the `"throw new RuntimeException"` statement. For those of you who use Eclipse, we have also added a TODO: comment in each of these methods. The "Tasks" window will give you a list of all TODO comments, which will help you find and complete these methods.

See the problem set guidelines and advice in the assignments section.

The provided test suites in problem set 1 are the same ones we will use to grade your implementation; in later problem sets the staff will not provide such a thorough set of test cases to run on your implementations, but for this problem set you can consider the provided set of tests to be rigorous enough that **you do not need to write your tests**.

Errata

- The `div` code provided has a small bug. The current code has a line which reads:
-

-
  ```
  RatNum factor = remainder.get(0).coeff.div(p.coeff(degree()));
  ```

However, it should read:

```
RatNum factor = remainder.get(0).coeff.div(p.coeff(p.degree()));
```

- There is a small bug in PolyCalcFrame.java. To reproduce the error, you can do:
  1) Run PolyCalcFrame.
  2) Enter the polynomial "0" onto the stack.
  3) Clear the stack.
  4) Hit the integrate button.

  An exception should appear in the console of Eclipse indicating that an error occurred in the checkRep() method of RatPolyStack. The same error should appear when you follow the above direction but for step 4, hit the differentiate button.
  The problem is that the precondition of RatPolyStack.differentiate() and RatPolyStack.integrate() is not satisfied before calling the methods when the buttons are clicked. To fix this bug, you can change the following in your PolyCalcFrame.java.
-
- `void jButton_int_actionPerformed(ActionEvent e)`
- `if (stack != null)`
- ***`if (size > 0)`***
- `stack.integrate();`

  Similarly for

```
void jButton_dif_actionPerformed(ActionEvent e)
if (stack != null)
    if (size > 0)
            stack.differentiate();
```

Q & A

This section will list clarifications and answers to common questions about problem sets. We'll try to keep it as up-to-date as possible, so this should be the first place to look (after carefully rereading the problem set handout and the specifications) when you have a problem.