Feedback on triangle assignment

Your Triangle program passes 50 out of 50 tests. That's 100%.

General feedback

My `classify` function looks like this:

```
// Classify a triangle from the lengths, in string format.
// A triangle is taken to be scalene if it fails all other checks.
String classify(String a, String b, String c) {
    long x = number(a), y = number(b), z = number(c);
    String type;
    if (x == 0 || y == 0 || z == 0) type = "Illegal";
    else if (impossible(x, y, z)) type = "Impossible";
    else if (flat(x, y, z)) type = "Flat";
    else if (equilateral(x, y, z)) type = "Equilateral";
    else if (isosceles(x, y, z)) type = "Isosceles";
    else if (right(x, y, z)) type = "Right";
    else type = "Scalene";
    return type;
}
```

In order to avoid repetitive code for checking the three strings for validity and converting them to numbers, I have invented a function `number` to do that for one string, then called it three times. I've written a separate function to test each type.

There are going to be issues to do with overflow later, so the lengths are stored in `long`variables, and the `number` function returns a `long`. If the string is invalid or out of range,`number` returns `0`. Here's the function:

```
// Convert a string to a long, returning 0 if the string is invalid, e.g.
// starts with an unnecessary leading zero or is out of range.
long number(String s) {
    long x, m = Integer.MAX_VALUE;
    try { x = Long.parseLong(s); }
    catch (Exception e) { x = 0; }
    if (s.startsWith("0")) x = 0;
    if (x < 1 || x > m) x = 0;
    return x;
}
```

I've taken the approach here (discussed and illustrated in the feedback for the grade assignment!) of calling a parse function and catching any exception it produces in order to detect validity. It would be possible to check validity first before calling the function, to avoid exceptions, but it would be quite messy.

To keep the `classify` function short and readable, I've invented a function for each triangle type. The checks are carefully ordered, with the more unusual cases first. Then each function can be written assuming that the previous checks have happened. I've documented the assumptions in the comments for maximum clarity. The functions are:

```
// Is a triangle impossible?  Assume it is legal.
boolean impossible(long x, long y, long z) {
    return x + y < z || x + z < y || y + z < x;
}

// Is a triangle flat?  Assume it is legal.
boolean flat(long x, long y, long z) {
    return x + y == z || x + z == y || y + z == x;
}

// A 'normal' triangle is not illegal or impossible or flat.

// Is a normal triangle equilateral?
boolean equilateral(long x, long y, long z) {
    return x == y && y == z;
}

// Is a normal, non-equilateral triangle isosceles?
boolean isosceles(long x, long y, long z) {
    return x == y || y == z || x == z;
}

// Is a normal triangle right-angled?
boolean right(long x, long y, long z) {
    return x*x + y*y == z*z || x*x + z*z == y*y || y*y + z*z == x*x;
}
```

The Tests

Some of the tests are about the range of numbers allowed. Assuming that you are not going to use 'big integers', there must be some limit beyond which your program won't work. It is important to specify exactly what that limit is, and to reject numbers beyond that limit, and to make sure that everything works properly right up to that limit. The tests reveal that the maximum value of an int, 2147483647, is to be the largest allowable length, and that is just enough to make sure that `long` works to do all the calculations without overflow.

The last three tests 48, 49, 50 check for overflow and precision, and are quite subtle. The thing is that if you use the `int` type then overflow can happen. When you check `x+y == z` to look for flat triangles, the `x+y` calculation can overflow and become negative. When you check `x*x + y*y = z*z` for right-angled triangles, the `x*x + y*y` calculation or `z*z`calculation can overflow. This is because computer arithmetic is conventionally done modulo 2^{32}, so that all numbers are in the range -2^{31} to $2^{31}-1$, and if you have an overflow such as adding one to $2^{31}-1$, the result has 2^{32} subtracted to give -2^{31}. This is called "wrapping round".

The easiest way to avoid the overflow problem in Java is to use type `long`. These are 64-bit numbers which go up to $2^{63}-1$. You can check that there are no overflow problems, like this: the worst case is `x*x + y*y` with x and y both $2^{31}-1$; this gives a total less that 2^{63}, so all is OK.

Test 48 uses the numbers 1100000000 1705032704 1805032704 which should give the answer `Scalene`. However, those numbers satisfy the right-angled test modulo 2^{32}, so you

get the wrong answer if your program overflows. (This test was hard to devise. Mathematicians might be interested in the note about it at the bottom of this page.)

One way that some people might choose to avoid the overflow problem is to switch to using floating point numbers. This is the wrong way to solve the problem, so test 49 uses the numbers 2000000001 2000000002 2000000003, to show why. The problem is that floating point numbers are approximate, so if you use the float type, the last digit of these three numbers is lost, they all look the same, and you get the wrong answer `Equilateral`.

Using the double type is another way to try to avoid the overflow problem. It also has precision problems, but they are not so easy to illustrate. The last test is one which gives the wrong answer `Equilateral` if you use doubles, because of the precision lost in calculating `x*x + y*y`. A double has 52 significant bits of precision, which is not enough to handle the product of two ints, which needs about 62 bits. I found the test using a bit of maths, a bit of intuition, and by writing a little search program. Interestingly, people sometimes check this result using a calculator, but the trouble is that calculators often use the double type, and so they also get it wrong!

Computers have overflow detection, but high level languages conventionally turn it off because 'it is too expensive', even though that is not really true any more. The C language allows you to switch detection on, but programmers typically only use it during development then switch it off for production. As one of my colleagues says, this attitude is like wearing a life jacket on land, then taking it off when you go to sea!

The Overflow Test

The challenge was to come up with a test which would distinguish between using `int`which is limited to 32 bits or `long` which is big enough to avoid overflow.

Basing it on the right-angled triangle formula, I wanted to find three numbers `x, y, z`between 1 and `2^31-1` where

```
1)  x^2 + y^2 = z^2 + k*2^32
```

Then, a program based on 32-bit integers would do the arithmetic modulo `2^32` and would decide incorrectly that `x^2 + y^2 = z^2`and would incorrectly report the triangle as right-angled.

Formula 1 has a lot of leeway in it, so it should be possible to find a solution, but it needs a bit of care. One approach is to rewrite it as:

```
2)  x^2 - k*2^32 = z^2 - y^2
```

A little bit of classical algebra gives:

```
3)  x^2 - k*2^32 = (z + y) * (z - y)
```

These numbers are limited by `2^31` which is about `2*10^9` so we would expect all the numbers `x, y, z` to be around `10^9`, and differences to be around `10^8`, say. Let's arbitrarily choose `z - y = 10^8`.

```
4)  z - y = 10^8
```

That means we want `10^8` to divide the left hand side of formula 3. The easiest way to get that to happen is for `10^8` to divide both `x^2`and `k*2^32`. That means `10^4` divides `x` and `10^8` divides `k`, so let's set:

```
5)  x = a * 10^4
6)  k = b * 10^8
```

Dividing formula 3 through by `10^8` gives:

```
7)  a^2 - b*2^32 = z + y = 2*y + 10^8
```

Now there is more than enough leeway to finish the job. Choose `a` more or less arbitrarily and fiddle about until all the numbers come out with the right sort of size. It turns out that `a = 110000` and `b = 2` does fine.