# Etude 5:

Written by Ben Dutton and Francesco Lee

**Question 1:**

The problem occurs when we subsitute the number "2,000,000,000" as x for the equation (x + x)/2. This is a relatively simple issue, as all ints in Java are stored as a 32 bit integer by default. It is common knowledge (amongst computer scientists anyway) that a 32 bit integer can only store a value up to 2^31 - 1, or 2147483647. When we tried to add 2,000,000,000 with itself, it was supposed to produce four billion, but as it is over the value that a 32 bit integer can store, it overflowed and became a negative number. This number was then divided by two, leading to the final result of -147483648, which is obviously incorrect.

**Question 2:**

Everything goes fine here, as expected.

**Question 3:**

The lines that are incorrect in question three are:

```
System.out.println(midpt2(1, 2) + " " + midpt2(2, 1));

System.out.println(midpt2(1, 4) + " " + midpt2(4, 1));

System.out.println(midpt2(2, 3) + " " + midpt2(3, 2));

System.out.println(midpt2(3, 4) + " " + midpt2(4, 3));
```

The reason becomes clear when we subsititute it into the formula:

x + ((y - x)/2) we get:

1 + ((2 - 1)/2) = 1, because Java rounds down 1 / 2 to 0, which results in 1 + 0 = 1

2 + ((1 - 2)/2) = 2, and Java rounds down -1 / 2 to 0 again as well, which results in 2 + 0 = 2

The same sort of loss of precision results in the same logical error for all the other lines of

code that meets with this problem.

**Question 4:**

The definition `x1 <= x2 ? (x2 - x1) / 2 + x1 : (x1 - x2) / 2 + x2`

now works for all the lines that were incorrect before, as when we have

```
System.out.println(midpt3(1, 2) + " " + midpt3(2, 1));
```

The equation becomes:

1 <= 2 is true, therefore (2 - 1) / 2 + 1, which is equal to 1 with everything stored as ints and the second case would be:

2 <= 1 is false, therefore (2 - 1) / 2 + 1, which is also equal to 1 with everything stored as ints.

Therefore, since everything is arranged in exactly the same way, the situation in question 3 is avoided.

**Question 5:**

For Question 5, the problem was quite simple. First we initalized i to the max value,

```
Integer i = Integer.MAX_VALUE;
```

Then we ran the midpoint code on the values of the integer i like so:

```
System.out.println(midpoint(-i,i));

System.out.println(midpt2(-i,i));

System.out.println(midpt3(-i,i));
```

These were the functions that were called:

```
private static int midpoint(int x1, int x2){

    int midpointx = (x1+x2)/2;

    return midpointx;

}

private static int midpt2(int x1, int x2){
```

```
    return x1 + ((x2-x1)/2);
}

private static int midpt3(int x1, int x2){
    return x1 <= x2? (x2-x1)/2 + x1 : (x1-x2)/2 + x2;
}
```

The first line returned the correct value of 0, as expected, but however, the other two methods return the same wrong result of -2147483648. This is because in midpt2 and midpt3, there comes a moment where we have one number minus another, and since one of our numbers input into the function is a negative, we have siuations where we get:

-2147483648 -2147483648 (for midpt2) and 2147483648 - (-2147483648) (for midpt3)

These situations both result in overflow, as the answer is over four billion, but int max already represents the maximum value that one can store in an int.

**Question 6:**

If the numbers the program will deal with are guaranteed to be small and don't deal with decimals to a great extent or if the accuracy isn't all that important, then it would be okay to use a "unsafe" language like java. When creating programs which need very complex and very accurate calculation a "safe" language is necessary.

Our experimentation this etude shows us that when we are programming something that requires numerical precision, we should endeavour to use libraries from trusted sources (as discussed in the town hall). This is particularly important when we are dealing with real world examples, as overflow errors could potentially wipe out billions of dollars (in the case of bank software) or endangering critical defense systems (such as missile defense systems software, which do not have a perfect success rate in the first place).

**Question 7:**

Output:

0 0.000000e+00

1 0.000000e+00

2 0.000000e+00

3 0.000000e+00

4 0.000000e+00

5 0.000000e+00

6 0.000000e+00

7 0.000000e+00

8 0.000000e+00

9 0.000000e+00

10 0.000000e+00

11 0.000000e+00

12 0.000000e+00

13 0.000000e+00

14 0.000000e+00

15 1.073742e-07

16 0.000000e+00

17 0.000000e+00

18 0.000000e+00

19 inf

The last line reads inf, simply because the resulting answer from our calculations overflows even for float values. As for the 15th line, it was a victim of a rounding issue.

**Question 8:**

Output:

0 0.000000e+00

1 0.000000e+00

2 0.000000e+00

3 0.000000e+00

4 0.000000e+00

5 7.275958e-08

6 9.094948e-08

7 1.136869e-07

8 1.421086e-07

9 0.000000e+00

10 1.110223e-07

11 1.387779e-07

12 8.673618e-08

13 1.084202e-07

14 6.776264e-08

15 8.470330e-08

16 1.058791e-07

17 6.617446e-08

18 8.271807e-08

19 1.033976e-07

20 1.292470e-07

21 1.138989e-05

22 1.138383e-03

23 5.879122e-02

24 1.000000e+00

The output starts going wrong from i = 5 onwards (but i = 9 works fine). This is because

we have introduced a precision error, as a computer does not have the means to accurately store very small numbers inside a base 2 structure. These rounding errors are then magnified once we perform calculuations on these numbers that are just slightly off, resulting in incorrect answers. Test case nine worked fine because the numbers involved could be stored perfectly in a base 2 structure (i.e there is no way to represent 1/3 accurately in base 10, as it is 0.3333333 etc. However we can accurately represent 1/2 as 0.5, same thing happened at i = 9 and i = 12).

**Question 9:**

Multiply

0 0.000000e+00

1 0.000000e+00

2 0.000000e+00

3 0.000000e+00

4 0.000000e+00

5 0.000000e+00

6 0.000000e+00

7 0.000000e+00

8 0.000000e+00

9 0.000000e+00

10 0.000000e+00

11 0.000000e+00

12 0.000000e+00

13 0.000000e+00

14 0.000000e+00

15 0.000000e+00

16 0.000000e+00

17 0.000000e+00

18 0.000000e+00

19 0.000000e+00

Divide

0 0.000000e+00

1 0.000000e+00

2 7.450581e-08

3 0.000000e+00

4 1.164153e-07

5 7.275958e-08

6 9.094948e-08

7 1.136869e-07

8 1.421086e-07

9 8.881785e-08

10 1.110223e-07

11 1.387779e-07

12 8.673618e-08

13 1.084202e-07

14 6.776264e-08

15 8.470330e-08

16 1.058791e-07

17 1.323489e-07

18 1.654361e-07

19 1.033976e-07

20 6.462349e-08

21 8.077936e-08

22 1.009742e-07

23 1.262177e-07

24 7.888609e-08

With the new hyp function, all the cases in question 7 work successfully; this does not hold true for division however, as a computer just cannot store tiny numbers accurately without some dedicated library designed to deal with such tiny values. This shows us that if we wanted accurate answers from our programs, we need to use some tried and tested library in order to ensure that our answers were actually reliable. (Especially important if the program has something to do with spaceflight, as even the tiniest error could lead to a massive deviation in expected results).

**Question 10 (need to show the numbers):**

Output:

4.330126941204071044 92e-01 4.330126941204071044 92e-01

4.993746280670166015 62e+00 4.993745803833007812 50e+00

4.999937438964843750 00e+01 4.999937438964843750 00e+01

4.999999389648437500 00e+02 4.999999389648437500 00e+02

5.000000000000000000 00e+03 5.000000000000000000 00e+03

5.000000000000000000 00e+04 5.000000000000000000 00e+04

5.000000000000000000 00e+05 5.000000000000000000 00e+05

0.000000000000000000 00e+00 5.000000000000000000 00e+06

0.000000000000000000 00e+00 5.000000000000000000 00e+07

0.000000000000000000 00e+00 5.000000000000000000 00e+08

0.000000000000000000 00e+00 5.000000000000000000 00e+09

0.000000000000000000000e+00 4.999999897600000000000e+10

0.000000000000000000000e+00 4.999999979520000000000e+11

0.000000000000000000000e+00 4.999999913984000000000e+12

0.000000000000000000000e+00 5.000000018841600000000e+13

0.000000000000000000000e+00 4.999999934955520000000e+14

0.000000000000000000000e+00 5.000000136282112000000e+15

0.000000000000000000000e+00 4.999999921533747200000e+16

0.000000000000000000000e+00 4.999999921533747200000e+17

0.000000000000000000000e+00 4.999999990253223936000e+18

Heron's formula catastrophically fails when a = 10^6, this is because when s is calculated there is a 5 that is divided by 10 each time moving it rightward from the decimal point. Eventually this 5 is truncated when it gets small enough and so s begins to evaluate to exactly 1. As part of the formula has s-c where c is 1, having both s and c being 1 means this evaluates to 0 and so (s-a)*sqrt(s*(0)) = 0 and so instead of returning the right answer it returns 0 onwards from a = 10^6.