<http://javarevisited.blogspot.sg/2012/02/java-mistake-1-using-float-and-double.html>

Java Mistake 1 - Using float and double for monetary or financial calculation

**Java** is considered very safe programming language compared to C and C++ as it doesn't have free() and malloc() to directly do memory allocation and deallocation, You don't need to worry of array overrun in Java as they are bounded and there is pointer arithmetic in Java. Still there are some sharp edges in [Java programming language](http://javarevisited.blogspot.com/2011/11/run-java-program-from-command-prompt.html) which you need to be aware of while writing enterprise application. Many of us make subtle mistake in Java which looks correct in first place but turn out to be buggy when looked carefully. In this series of java articles I will be sharing some of common Java mistake programmers make while programming application in Java. As I have said earlier every day we learn new things but we forget something equally important. This again highlight importance of [code review](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) and following [best practices in Java](http://javarevisited.blogspot.com/2011/08/code-comments-java-best-practices.html). In this part we will discuss why double and float should not be used in monetary or financial calculation where exact result of calculation is expected.

## Using double and float for exact calculation

[http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)This is one of **common mistake Java programmer** make until they are familiar with BigDecimal class. When we learn Java programming we have been told that use float and double to represent decimal numbers its not been told that result of floating point number is not exact, which makes them unsuitable for any financial calculation which requires exact result and not approximation. float and double are designed for engineering and scientific calculation and many times doesn’t produce exact result also result of floating point calculation may vary from [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) to JVM. Look at below example of BigDecimal and double primitive which is used to represent money value, its quite clear that floating point calculation may not be exact and **one should use BigDecimal for financial calculations**.

public class **BigDecimalExample** {

    public static void main(String args[]) throws IOException {

      /**/floating point calculation**

      double amount1 = 2.15;

      double amount2 = 1.10;

      System.out.println("difference between 2.15 and 1.0 using double is: " + (amount1 - amount2));

**//Use BigDecimal for financial calculation**

      BigDecimal amount3 = new BigDecimal("2.15");

      BigDecimal amount4 = new BigDecimal("1.10") ;

      System.out.println("difference between 2.15 and 1.0 using BigDecimal is: " + (amount3.subtract(amount4)));

    }

}

**Output:**

difference between 2.15 and 1.0 using double is: 1.0499999999999998

difference between 2.15 and 1.0 using BigDecmial is: 1.05

From above example of **floating point calculation** is pretty clear that result of floating point calculation may not be exact at all time and it should not be used in places where exact result is expected.

## Using Incorrect BigDecimal constructor

Another mistake Java Programmers make is **using wrong constructor of BigDecmial**. BigDecimal has [overloaded constructor](http://javarevisited.blogspot.com/2012/01/what-is-constructor-overloading-in-java.html) and if you use the one which accept double as argument you will get same result as you do while operating with double. So always use BigDecimal with String constructor. here is an example of using BigDecmial constructed with double values:

**//Creating BigDecimal from double values**

BigDecimal amount3 = new BigDecimal(2.15);

BigDecimal amount4 = new BigDecimal(1.10) ;

System.out.println("difference between 2.15 and 1.0 using BigDecmial is: " + (amount3.subtract(amount4)));

**Output:**

difference between 2.15 and 1.0 using double is: 1.0499999999999998

difference between 2.15 and 1.0 using BigDecmial is: 1.049999999999999822364316059974953532218933105468750

I agree there is not much difference between these two constructor but you got to remember this.

## Using result of floating point calculation in loop condition

One more mistake from Java programmer can be **using result of floating point calculation for determining conditions on loop**. Though this may work some time it may result  in infinite loop another time. See below example where your Java program will get locked inside infinite while loop:

double amount1 = 2.15;

double amount2 = 1.10;

while((amount1 - amount2) != 1.05){

  System.out.println("We are stuck in infinite loop due to comparing with floating point numbers");

}

**Output:**

We are stuck in infinite loop due to comparing with floating point numbers

We are stuck in infinite loop due to comparing with floating point numbers

……………

…………..

This code will result in infinite loop because result of subtraction of amount1 and amount 2 will not be 1.5 instead it would be "1.0499999999999998" which make boolean condition true.

That’s all on this part of learning from mistakes in Java, bottom line is :

* Don’t use float and double on monetary calculation.
* Use BigDecimal, long or int for monetary calculation.
* Use BigDecimal with String constructor and avoid double one.
* Don’t use floating point result for comparing loop conditions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html>

Mixing static and non static synchronized method - Java mistake 2

**Using static and non static synchronized method** for protecting shared resource is another Java mistake we are going to discuss in this part of  our series “learning from mistakes in Java”. In last article we have seen [why double and float should not be used for monetary calculation](http://javarevisited.blogspot.com/2012/02/java-mistake-1-using-float-and-double.html) , In this tutorial we will find out why using static and non static synchronized method together for protecting same shared resource is not advisable.

I have seen some times Java  programmer mix [static synchronized method](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and instance synchronized method to protect same shared resource. They either don't know or failed to realize that **static synchronized** and **non static synchronized** method **lock on two different object** which breaks purpose of synchronizing shared resource as two thread can concurrently execute these two method breaking mutual exclusive access, which can corrupt status of mutable object or even cause subtle [race condition in Java](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) or even more horrible [deadlock in java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html).

## Static and non static synchronized method Java

[static and non static synchronized method in java](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html)For those who are not familiar [static](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) synchronized method locked on class object e.g. for string class its String.class while instance synchronized method locks on current instance of Object denoted by “[this” keyword in Java](http://javarevisited.blogspot.com/2012/01/this-keyword-java-example-tutorial.html). Since both of these object are different they have different lock so while one thread is executing static synchronized method , other [thread in java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) doesn’t need to wait for that thread to return instead it will acquire separate lock denoted byte .class literal and enter into static synchronized method. This is even a [popular multi-threading interview questions](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html) where interviewer asked on which lock a particular method gets locked, some time also appear in Java test papers.

Bottom line is that  *never mix static and non static synchronized method for protecting same resource*.

## Example of Mixing instance and static synchronized methods

Here is an **example of multithreading code which is using static and non static synchronized method** to protect same shared resource:

**public** **class** SynchornizationMistakes {  
    **private** static int [count](http://www.php.net/count) = 0;  
    
    *//locking on this object lock*  
    **public** synchronized int getCount(){  
        return [count](http://www.php.net/count);  
    }  
    
    *//locking on .class object lock*  
    **public** static synchronized void increment(){  
        [count](http://www.php.net/count)++;  
    }  
      
}

here shared count is not accessed in mutual exclusive fashion which may result in passing incorrect count to caller of getCount() while another thread is incrementing count using static increment() method.

That’s all on this part of learning from mistakes in Java. Now we know that **static and non static synchronized method are locked on different locks** and should not be used to protect same shared object.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*