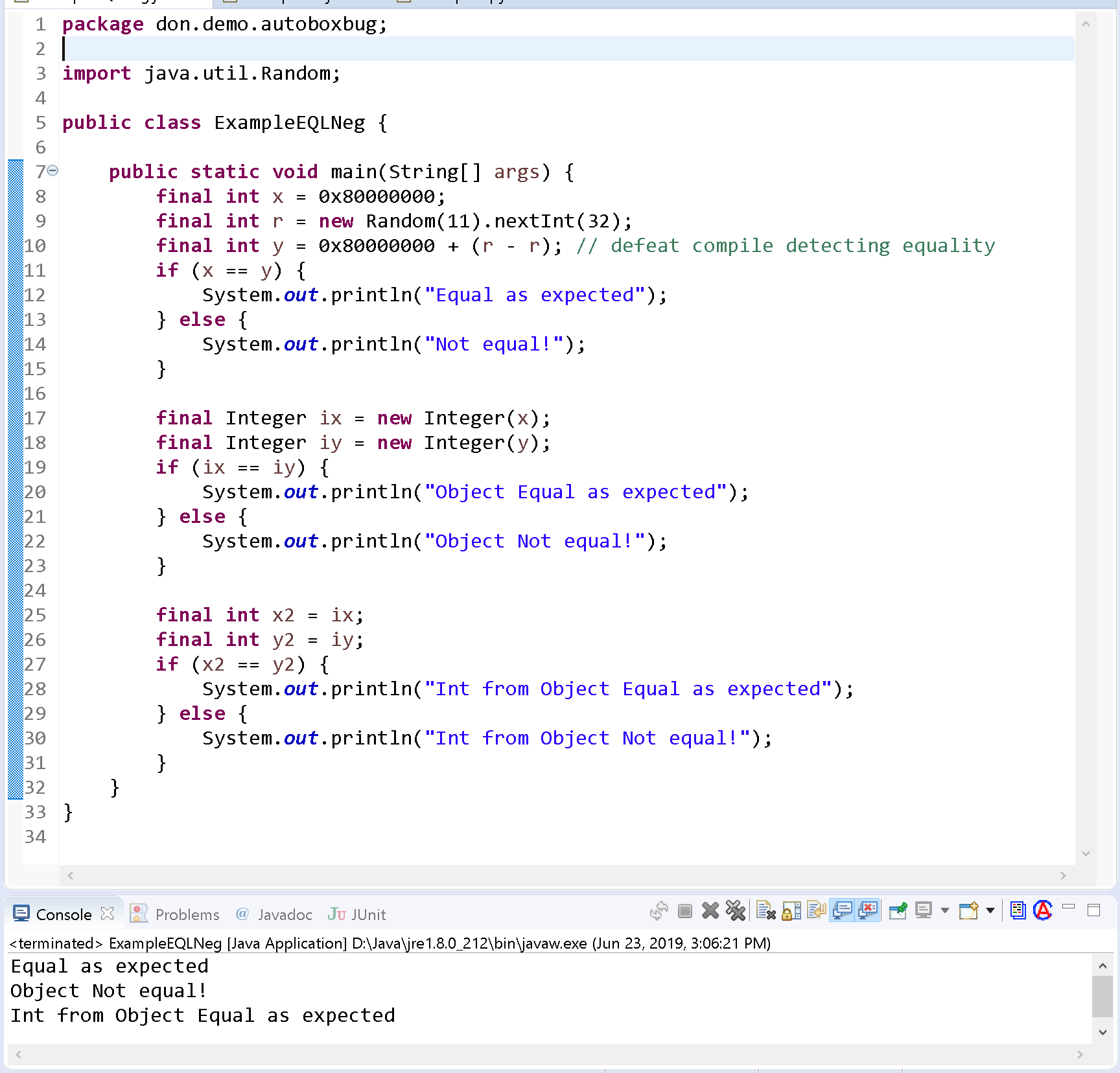
# Pilot Error

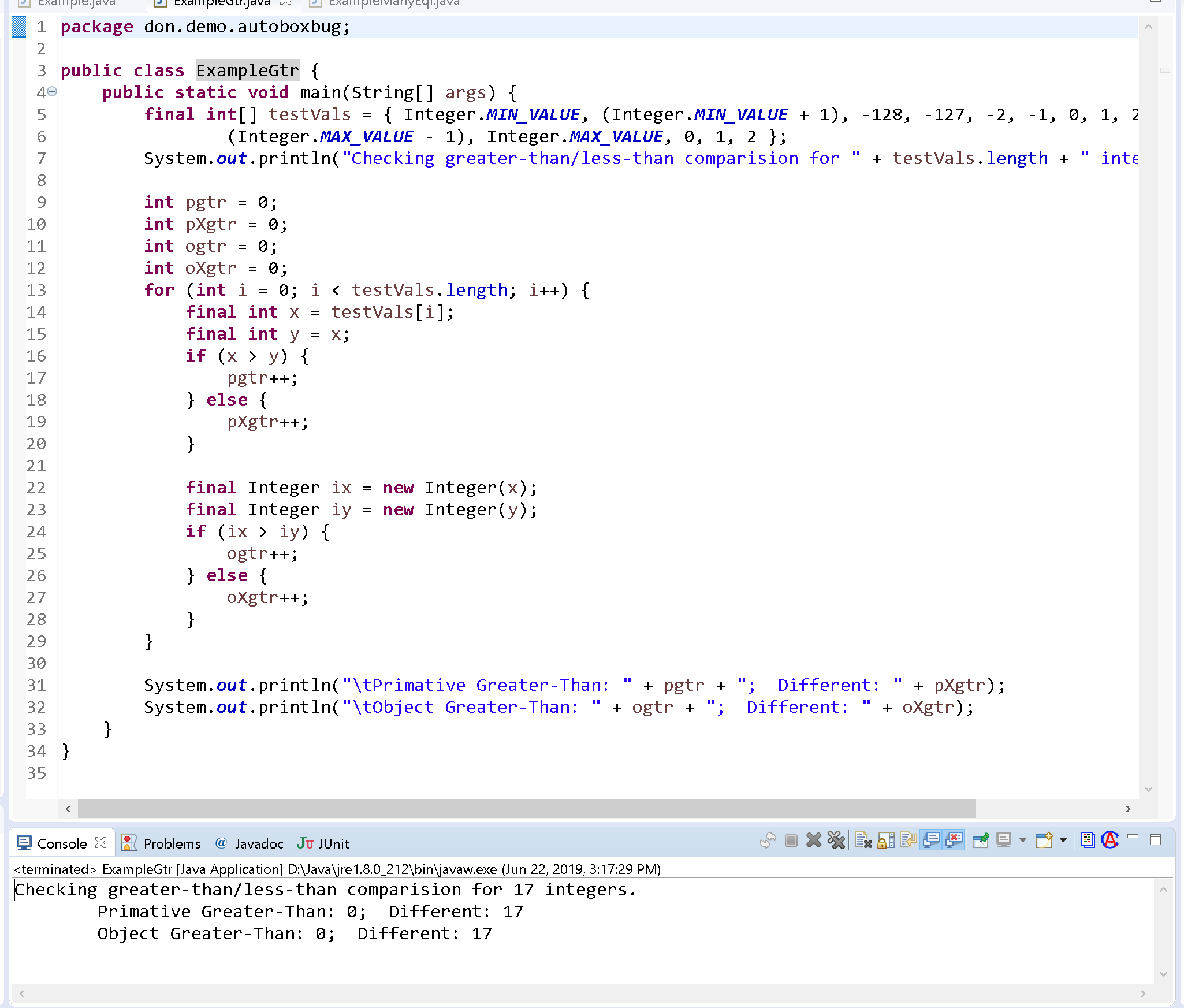
Programmers are constantly rediscovering conceptual errors in their coding and they suffer to correct their code. I wanted to capture a mistake I recently made. I once again had a “pilot error” experience in that Java appeared to not work work as expected. Just so you can enjoy my surprise, that one can still make a bone-head error after coding Java for 23 years, just follow along with the detective story. There are short in-line code snippets to review, but no links to follow. As a hint in this detective story, my programming background was in C++ prior to Java.

Recently, in the context of an interview question, I proposed sorting a collection of 32 bit unsigned integers (“bit strings”) to improve later searches. Sorting required creating a comparator to achieve unsigned Java ***Integer*** comparison because Java does not support unsigned numerical types. Java array and ***List*** instances store ***Integer*** values that are passed into the sorting comparator as parameters, and the comparator results are used to sort the list.

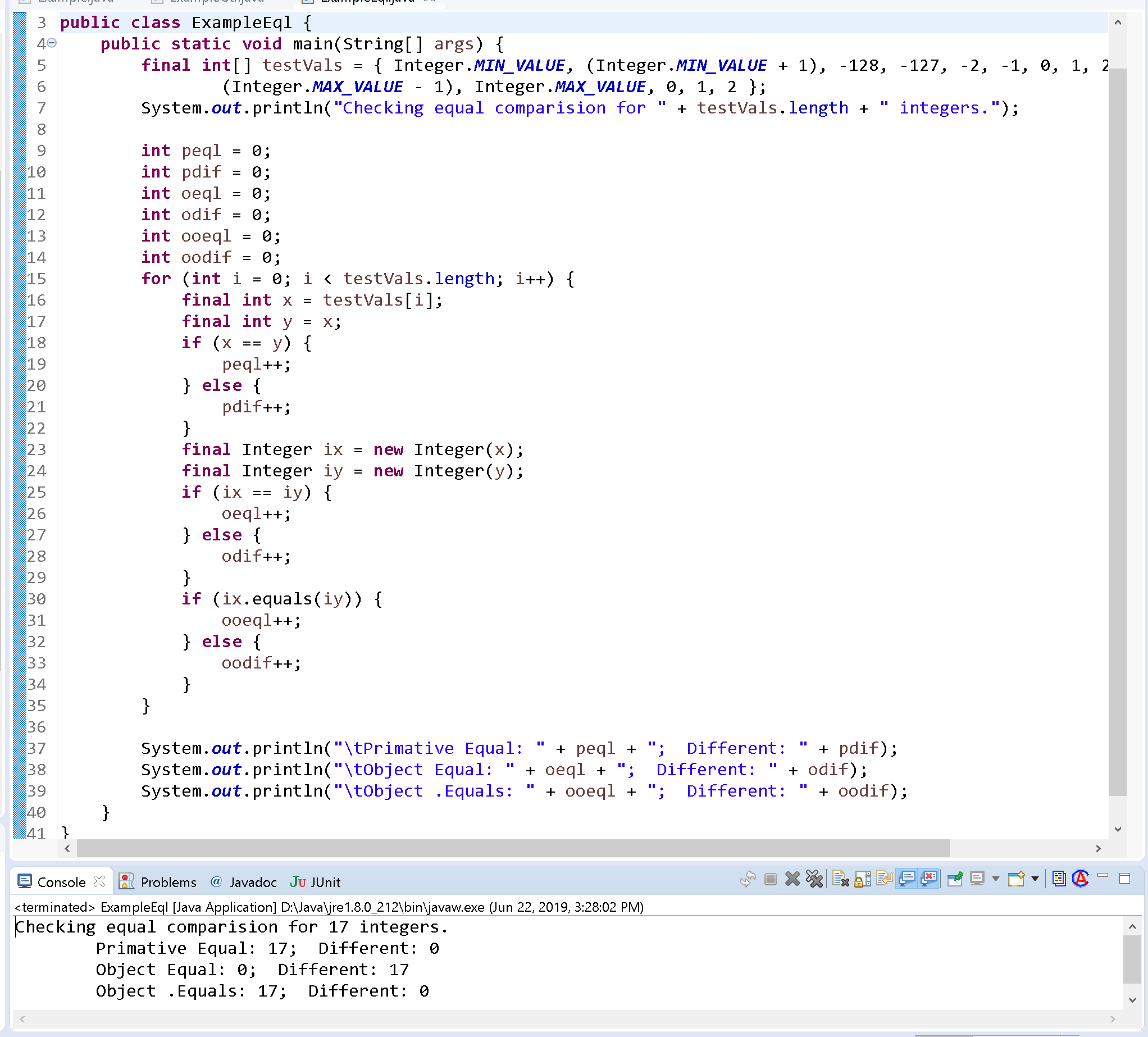
My comparator seemed to oddly fail around testing equality of some ***Integer*** instances, but succeed for other Integer instances, and succeeded for all other non-equality comparisons. Here is one such failure (the first example in my TDD approach.) We have set it up in a debugging example:



As you can see, the particular value (***Integer.MIN\_VALUE***) did not correctly compare equal when referencing objects, but did correctly compare when viewed as primitives. To add to the mystery, the other comparison operators worked as expected. Here is an example of GREATER-THAN success (e.g.  ***!(x > y)***):



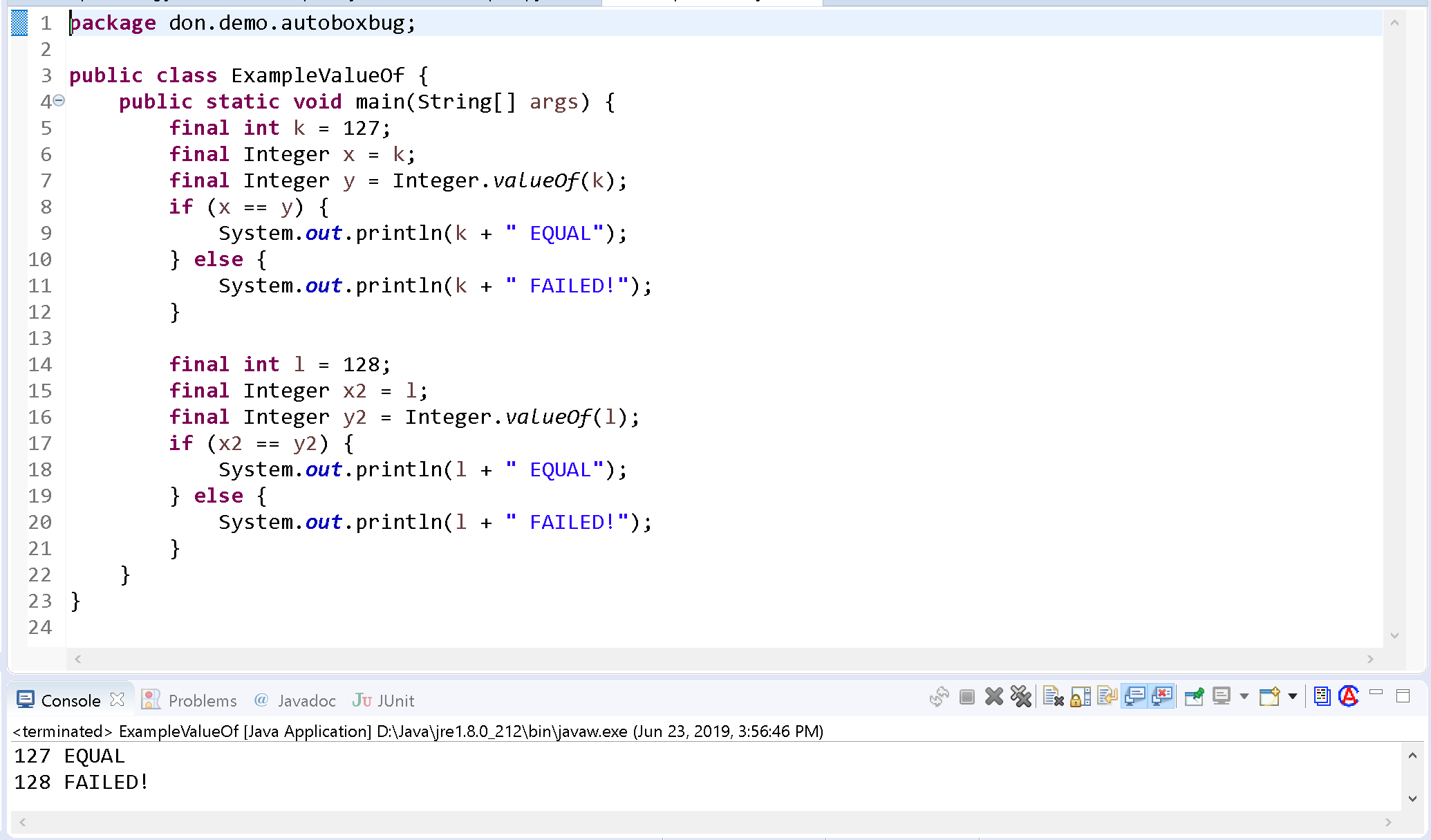
Seeing equality testing was the problem, I expanded the list of test cases for equality beyond the single test value shown above, and found that EQUAL would consistently fail on all of these additional test values:



Again, we can see that primitive equality comparison and comparison of extracted ***int*** values succeed, but object equality of ***Integer*** values fails. The next step was some research.

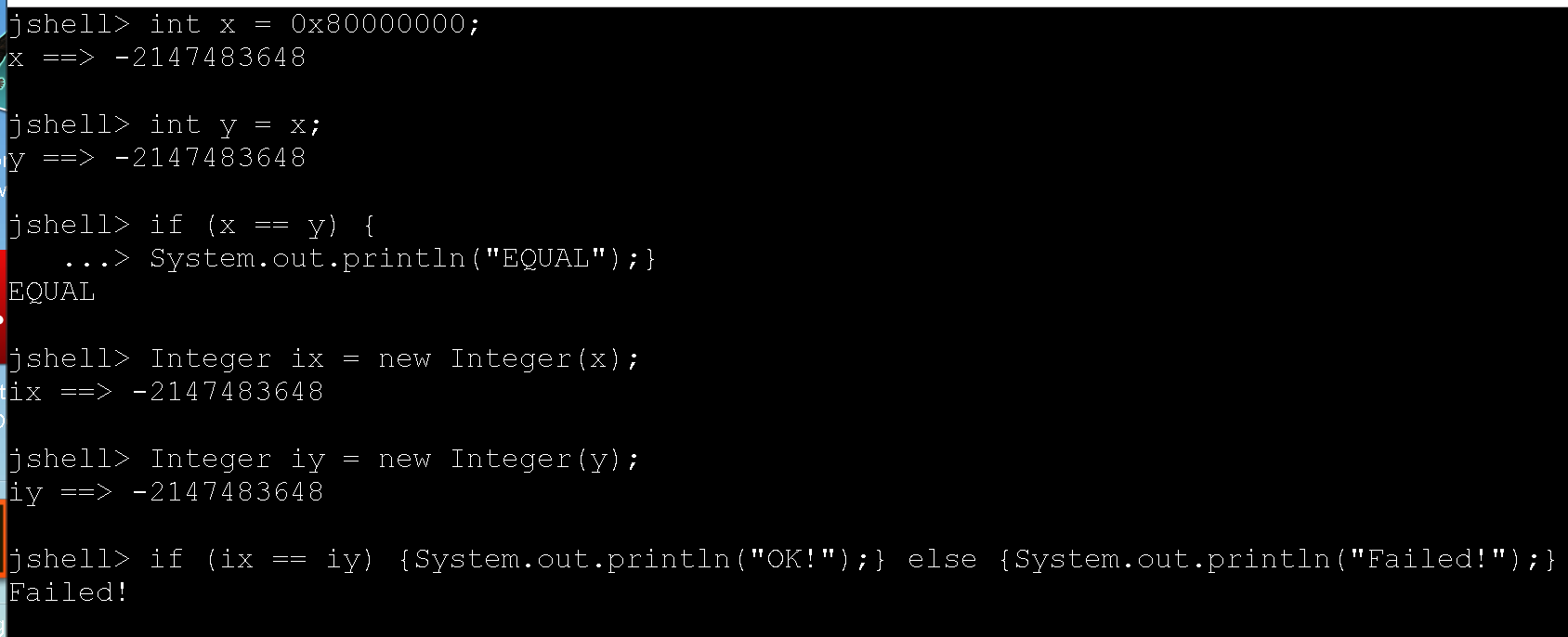
I discovered an historic but not widely perceived behavior in the Java “autoboxing” mechanism between the primitive ***int*** type and the object ***Integer*** wrapper. This issue should be more widely known because of the many potential situations in which it might occur. There were four Google references to autoboxing dating between 2011 and 2018, where I found using the search phrase “Java autoboxing errors”.

None of these references fully explained the behavior, and some pointed to the use of the ***Integer.valueOf()*** method in autoboxing causing comparison issues. *StackOverflow* had additional discussions on the interaction between ***Integer*** instances and the Java “*Constant Pool*”. Still, I found no real explanation for this specific problem. For illustration, here is an example of the interaction between the ***valueOf*** method and the *Constant Pool* and equality testing:



As you can see for ***Integer*** instance 127, equality compares correctly; but for ***Integer*** instance 128 it fails! So, the plot thickens. Autoboxing can cause unexpected differences in equality testing.

This conundrum is also present in Java 9, but there was a problem testing on my machine. I am working in Java 8 (for AWS), but there is an on-line version of Java 9 with ***JShell*** available for short tests. You may run Java 9 from <https://tryjshell.org>.) The results for this test are shown below:



## Problem Summary

Here is what we have so far:

* Integer instances sometimes correctly compare for equality, but often fail.
* This behavior is verified in Java 8 and Java 9
* The behavior may be related to issues in Autoboxing, introduced in Java 1.5
* All other instance comparisons (e.g., ***>*** and ***<***) work as expected.

## The Culprit is Revealed

Do you think you have the solution? Remembering back to my early days learning Java, we were told that the “***==***” operator tests for *identity*, and the object ***equals*** method tests for *content* equality. For some integer constants, specifically in the range -127 <= x <= 127, Java caches them in the *Constant Pool* using the ***Integer.valueOf*** method. That means that for a small set of ***Integer*** instances, object content and object identity testing get the same results. All other comparison operators extract the value of the integer instance and then perform the comparison. Ah, so that explains it!

Ω