Bug 4 Tracing

Legend: **blue** words are classes, **green** words are methods, and **purple** words are instance variables. (If a word is not coloured, that means it is being used in its natural English sense).

Contents

[Description 2](#_Toc463822057)

[Static Review 3](#_Toc463822058)

[Hypothesis testing 6](#_Toc463822059)

[Hypothesis 1 6](#_Toc463822060)

[Hypothesis 2 8](#_Toc463822061)

[Hypothesis 3 8](#_Toc463822062)

[First run 8](#_Toc463822063)

[Second run 9](#_Toc463822064)

[Third run 10](#_Toc463822065)

[Summary of testing results 11](#_Toc463822066)

[Resolution 12](#_Toc463822067)

[Risk assessment 12](#_Toc463822068)

[Testing the solution 13](#_Toc463822069)

[Conclusion 14](#_Toc463822070)

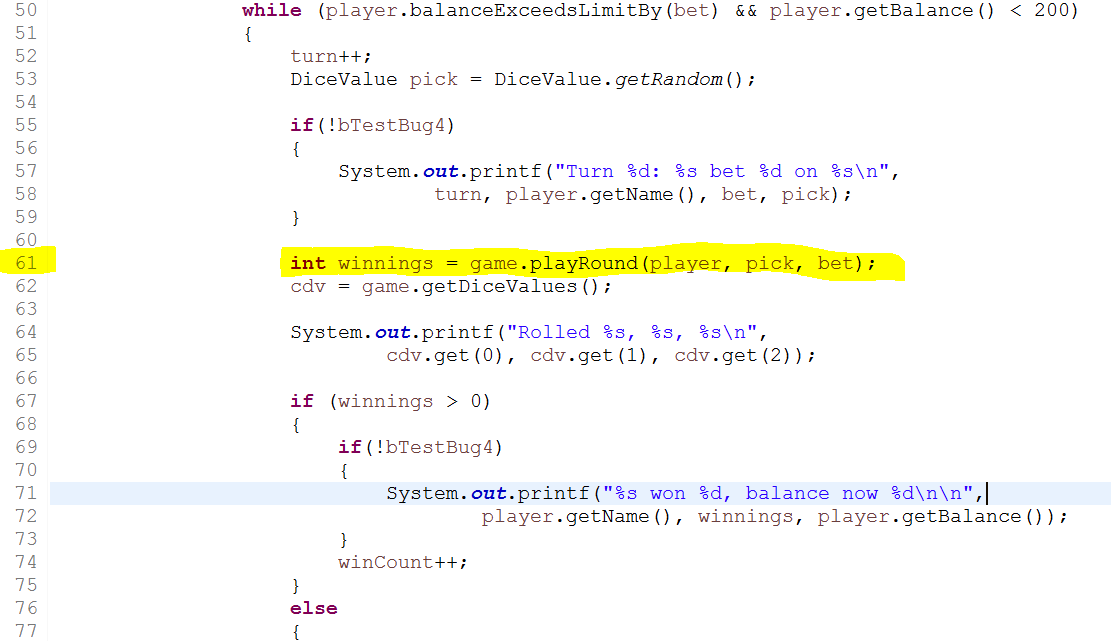
# Description

All rolls are identical in every game of each run through of **Main**. For example, it could be CROWN, CLUB, HEART repeated for every roll for every game. However, when the run is repeated, this roll can change. For example, in the next run it could be HEART, DIAMOND, DIAMOND repeated on every roll of every game.

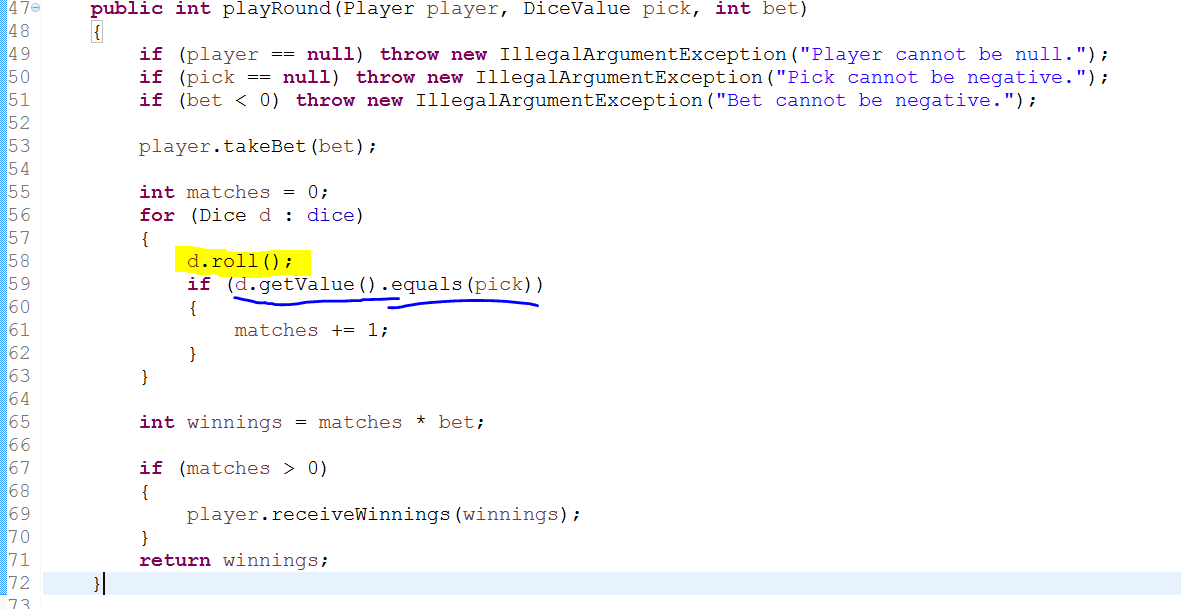
# Static Review

The method that should generate a new **DiceValue** to compare against the pick is **Dice**’s **roll**. This occurs in **Main**’s **main** method, in **Game**’s **playRound** method here:

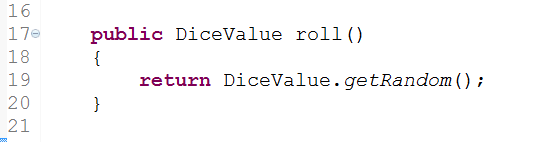
**Main**’s **main**: (highlighted in yellow)



**Game**’s **playRound**: (highlighted in yellow)



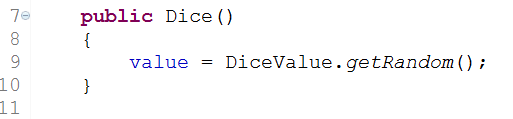
**Dice**’s **roll**:



We notice the following things:

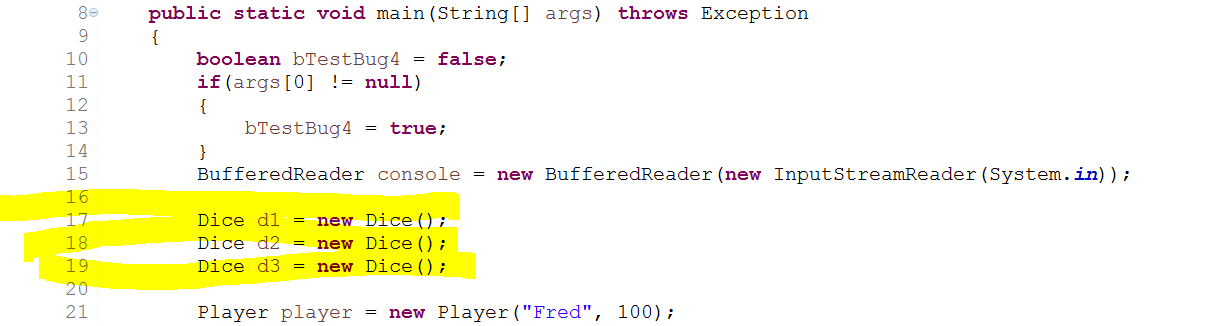
1. The **Dice** class has an instance variable **value** of type **DiceValue.**
2. It is this value that is getting compared against the pick (underlined in blue in **Game**’s **playRound** method, pictured above).
3. The roll method in **Dice** does not change the instance variable **value** (it returns a **DiceValue** object which is not captured – in **playRound** in **Game** it is called like a method with a void return).

If we look at the constructor of **Dice**:



We notice some additional things:

1. **DiceValue**’s **getRandom** is being called rather than the more natural **roll** method (given it is the same class). Although this shouldn’t matter because all **roll** does is call **getRandom**, and return exactly what **getRandom** returns.
2. The variable **value** is set from the return value of **getRandom**.
3. Nowhere else in the class **Dice** is **value** set in any way.
4. Conclusion from 5 and 6: **value** is invariant over the life of the instance.
5. If we look at the whole game loop in **Main**’s main we notice that here is the only place that new **Dice** are created:



From this we can guess that all **Dice** used in each run of the program are the same three objects. This will be tested to be sure (see *Testing Hypotheses* later in this document).

1. From 2, 7 and 8 (8 subject to testing as described later in this document) we can see why the rolls are always the same:
   * The **value** is invariant
   * The same **Dice** are used in every **Game** (therefore using the same invariant **value**)
   * It is the **value** that is displayed by the program as a roll
2. From 9: Obviously the roll will be identical every **Game**!

Therefore, our hypotheses are the following:

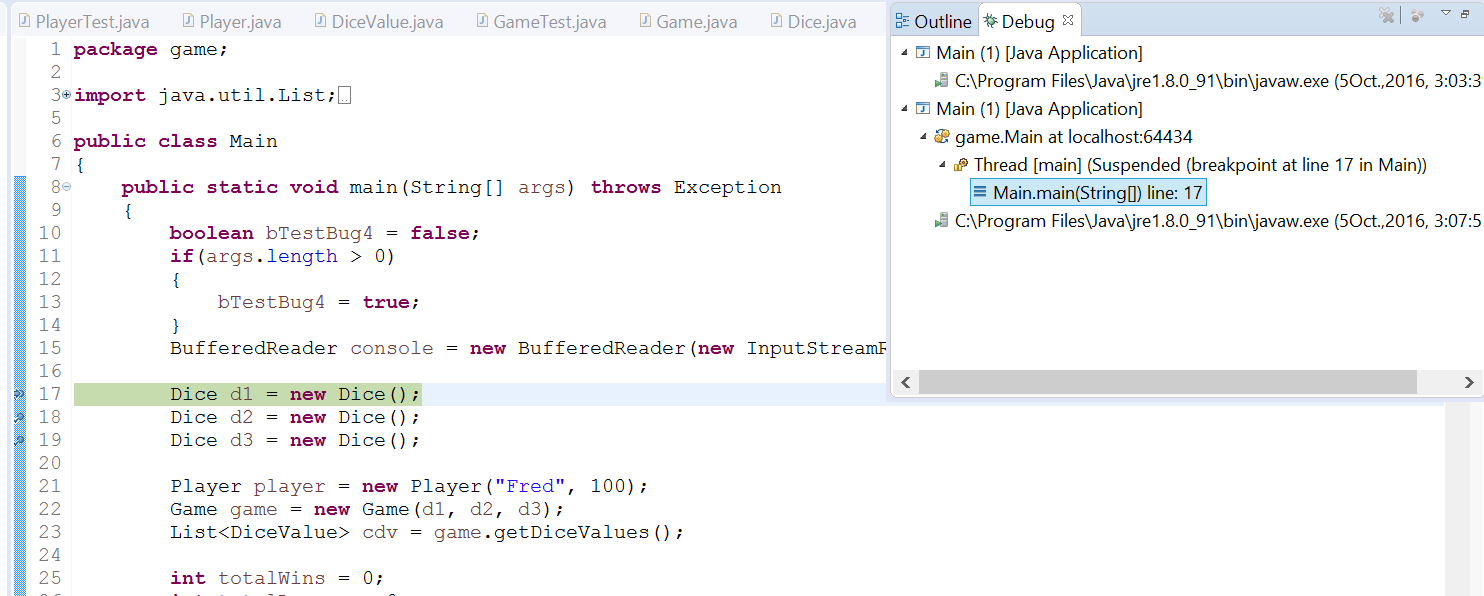
1. **Dice** are created only once per run of the program, and then reused for each **Game**.
2. The **value** is invariant over the life of any particular instance of **Dice**.
3. The **value** of the **Dice** is what is used as each roll & compared to the pick to determine if the player wins or not.

# Hypothesis testing

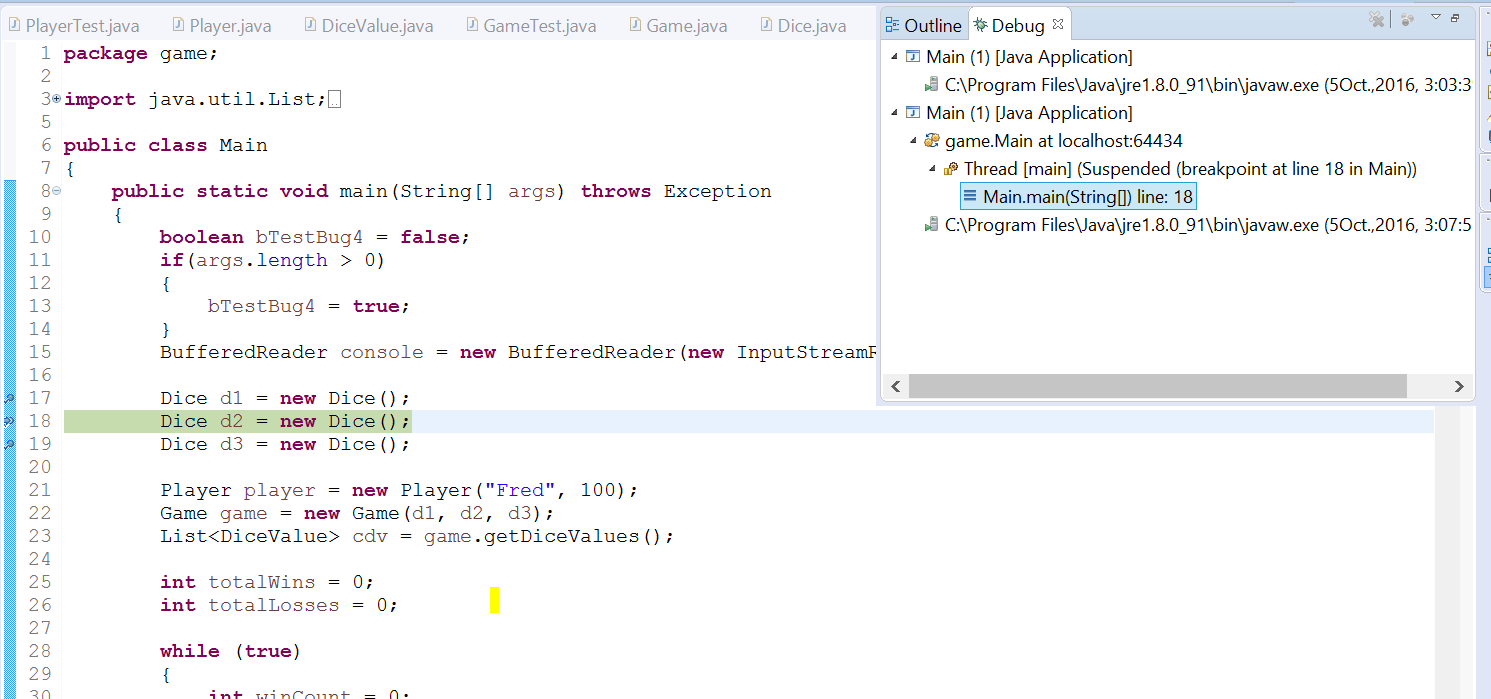
## Hypothesis 1

To test this one, we place breakpoints in **main** where the three **Dice** are created (as highlighted above) to prove that it isn’t orphan code that doesn’t get executed. We also place a breakpoint in **Dice**’s constructor, and run the program in debug mode and count how many times the point of execution enters the constructor. From the static review, we see that there are three **Dice** objects required in a **Game**. Therefore, we expect that the point of execution will enter **Dice**’s constructor exactly three times. If it does this, this will verify hypothesis 1 as true.

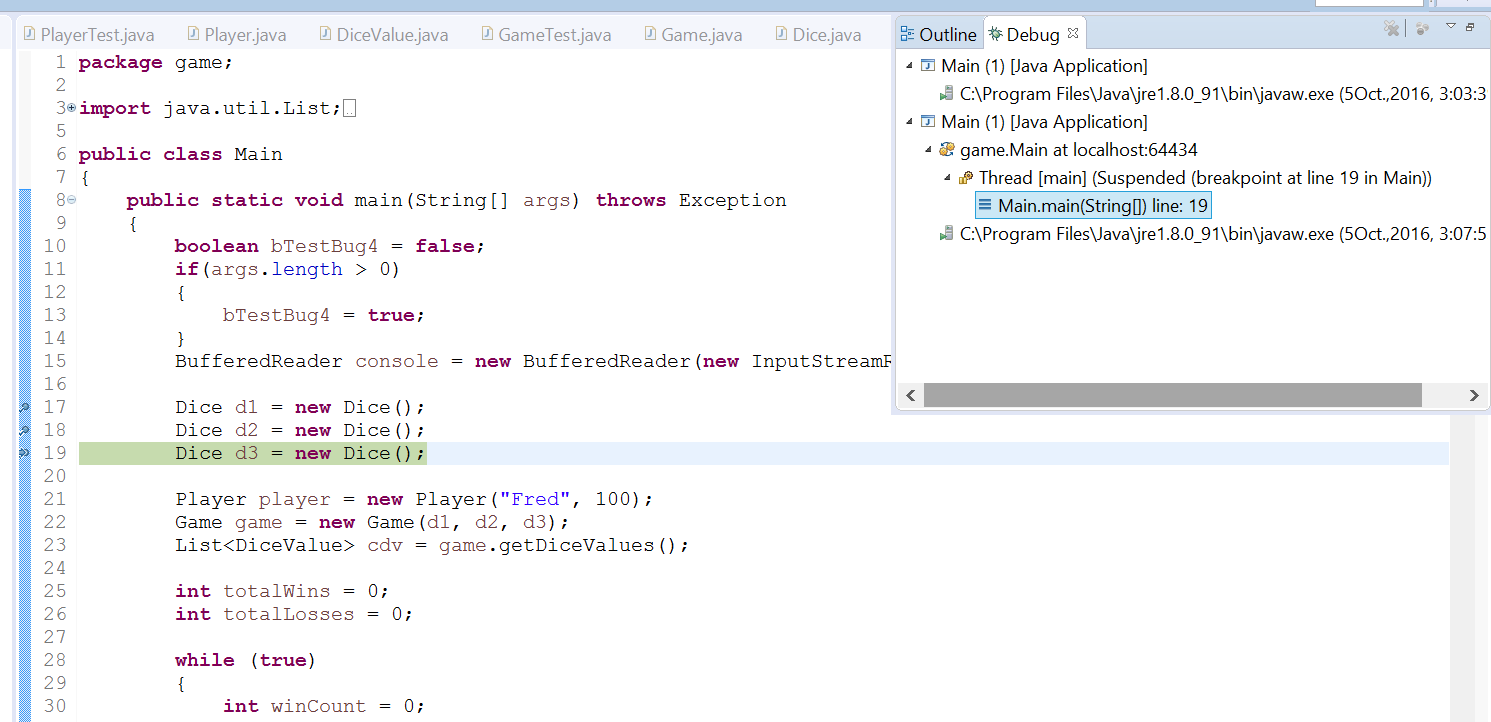
Hit creation of first **Dice**:



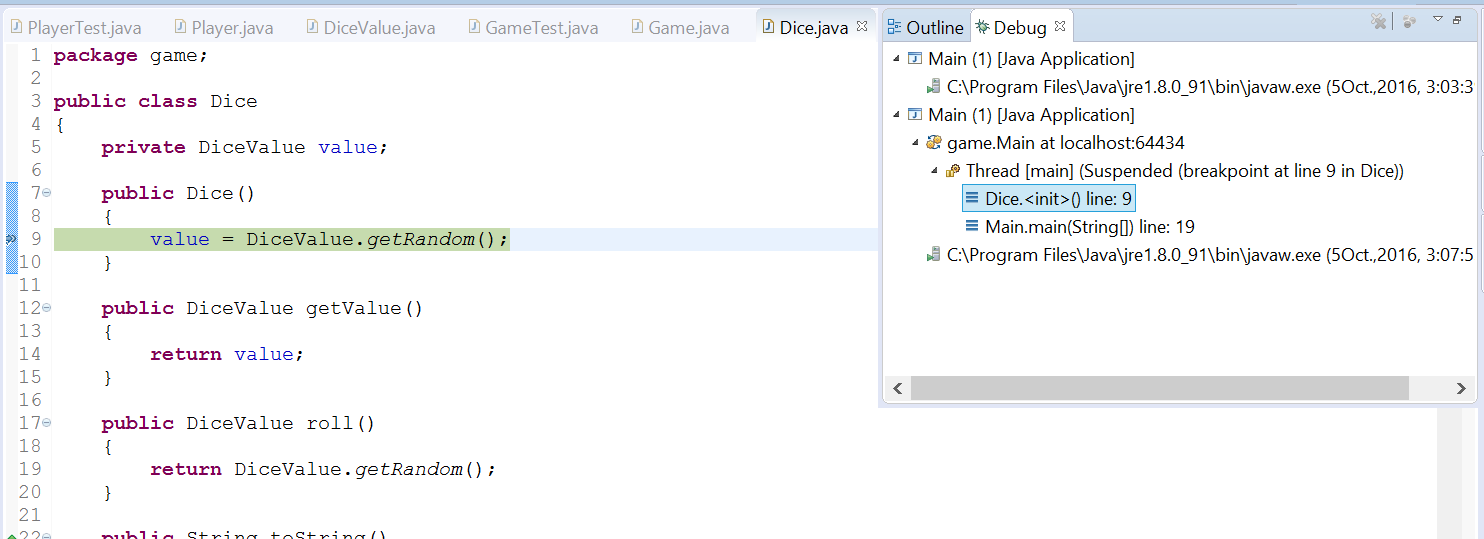
Hit creation of second **Dice**:



Hit creation of third **Dice**:



In between, it also hits the constructor (as expected, given it says “new Dice()”):



And most importantly, it only goes into the constructor three times (once after each “new Dice()” statement, as expected), and never again. This proves hypothesis 1: The three **Dice** created at the beginning of **main** are reused in each **Game**.

## Hypothesis 2

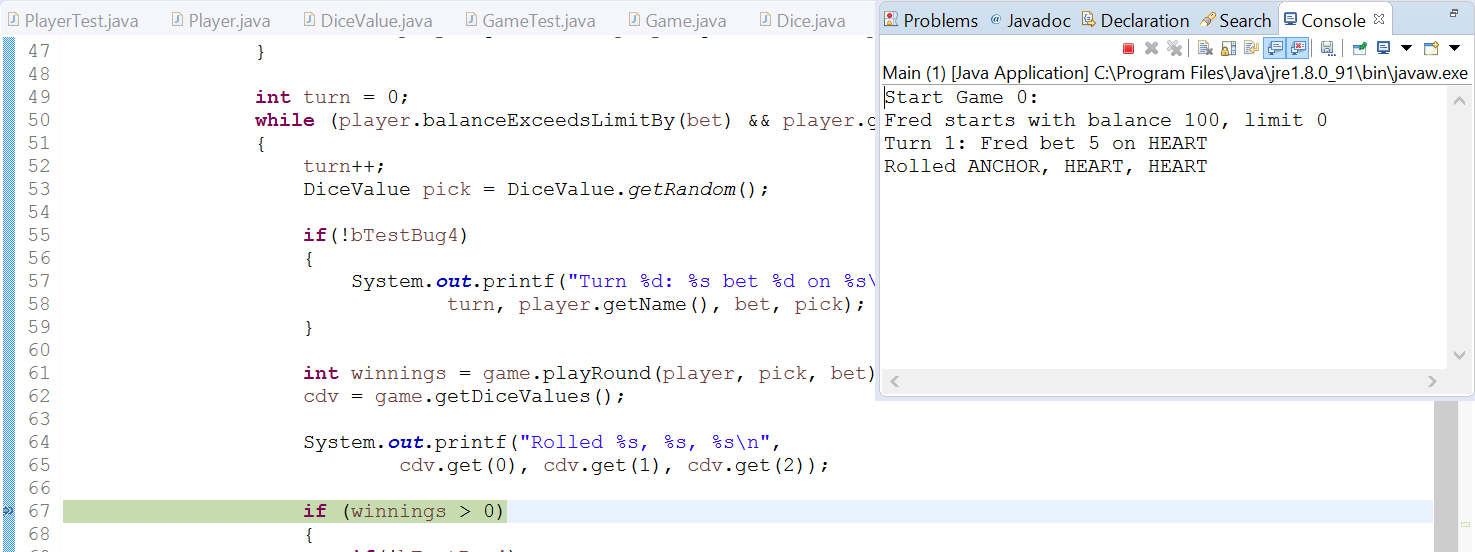
The instance variable **value** has a **private** access modifier meaning that the only way it can be changed is through methods in the class itself (there cannot be anything like: Dice dice = new Dice(); dice.value = DiceValue.SPADES; for example). In our static review we found that the only place in the **Dice** class that **value** is set is in the constructor (there is no setValue method for example). So, we would expect that **value** would only be set at creation and never changed again. However, maybe it is possible for a program to be sneaky and call the constructor after already being created? We don’t need to worry about this, as the previous test (hypothesis 1) has already tested for this, by putting breakpoints in the constructor, and noting that it only gets called three times (on creation of the three **Dice** objects used in **main**).

So, we don’t need to do any tests. Hypothesis 2 is proven during testing of hypothesis 1.

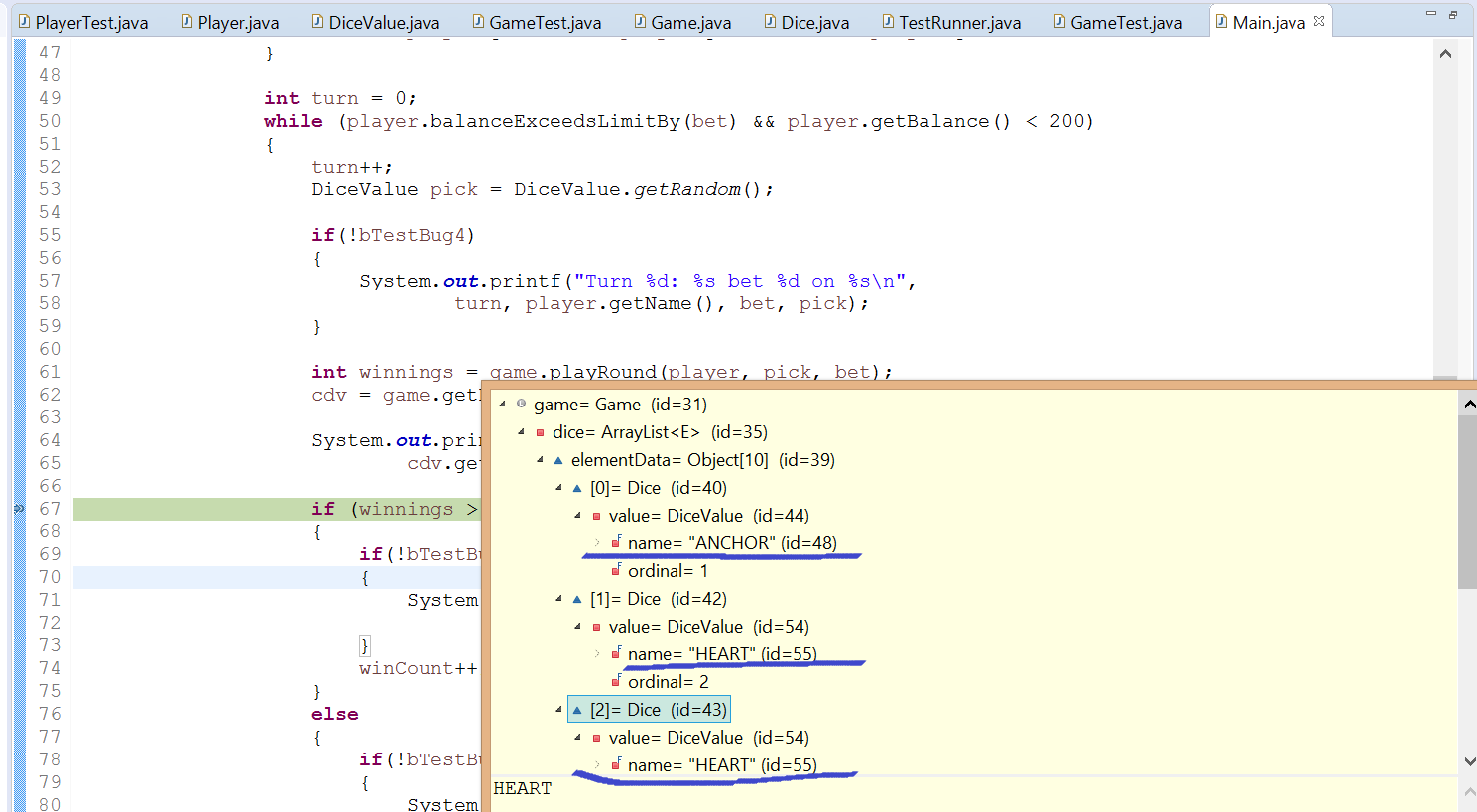
## Hypothesis 3

This one is quite clear from the static review, but we can still test it. We will put a breakpoint just after the rolls are displayed in the console window, and have a look at what the value of the corresponding **Dice** object’s **value** is at that moment. To assure our audience that it is not a coincidence, we will do this three times (for separate runs of the program).

First run

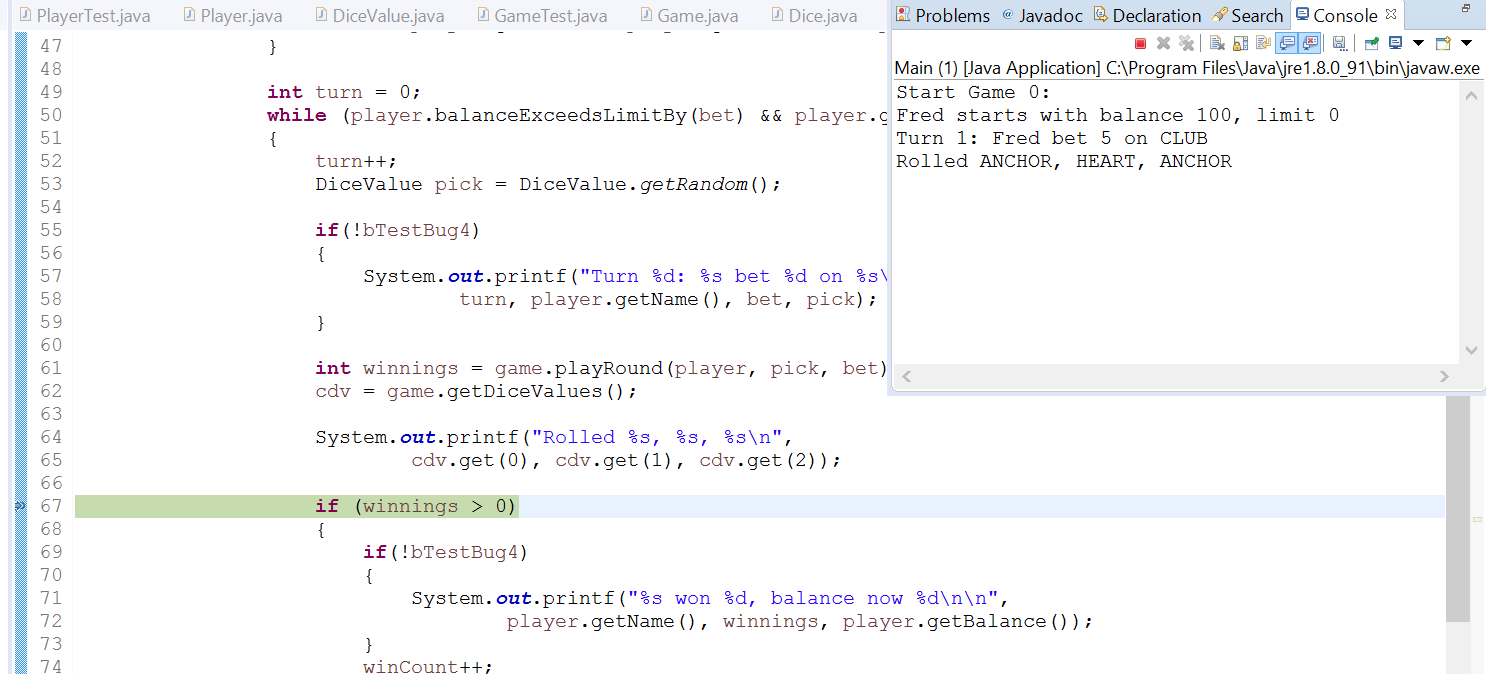
Showing the console window (Rolled ANCHOR, HEART, HEART):

Showing the values of **value** (underlined in blue) for each of the **Dice** in **Game** (values are ANCHOR, HEART, and HEART same as in the console window):

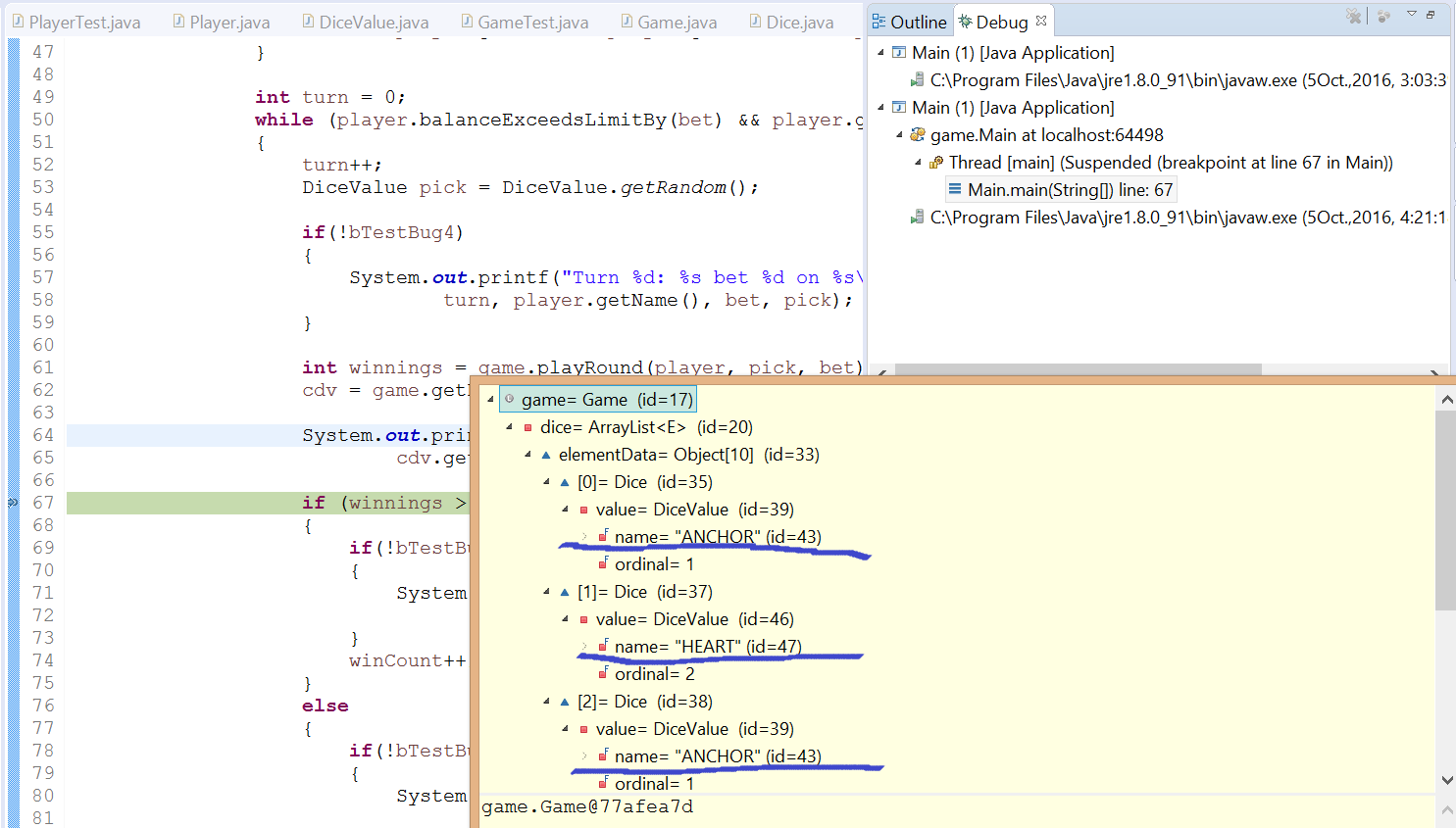


### Second run

Showing the console window (Rolled ANCHOR, HEART, ANCHOR):

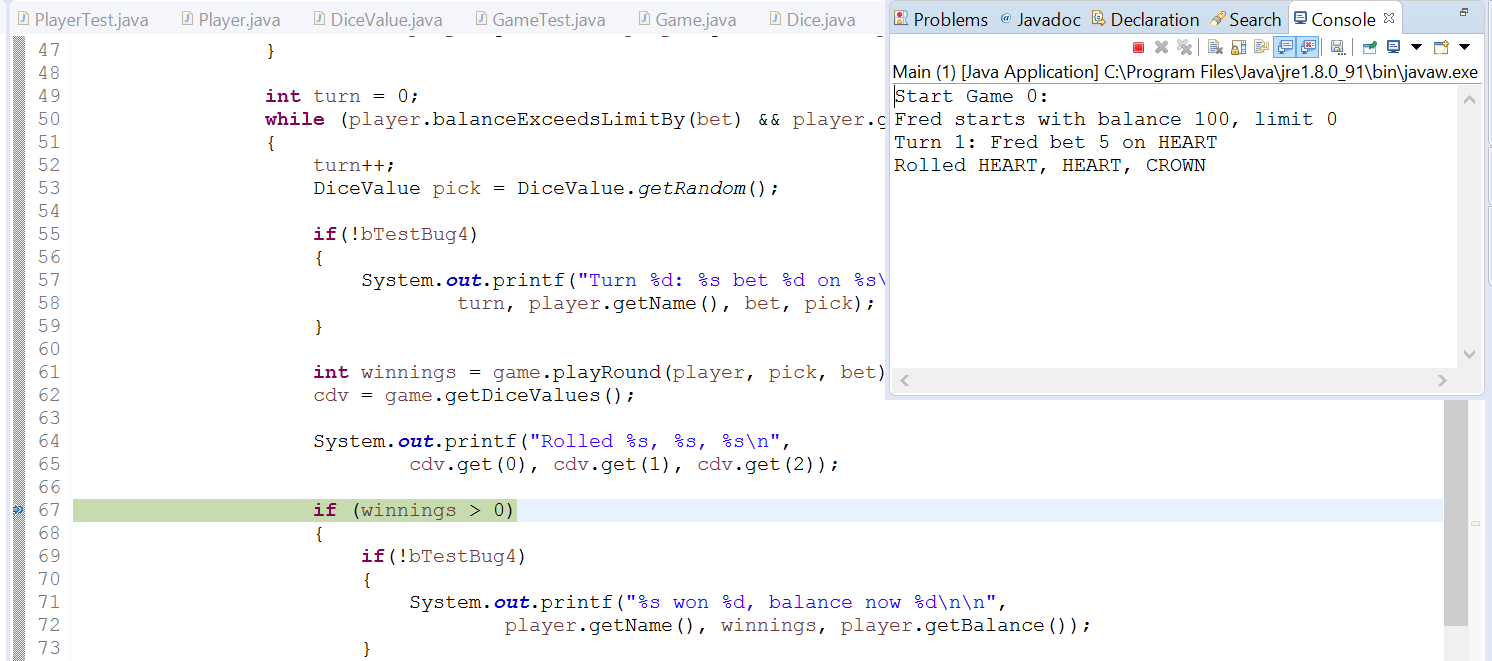


Showing the values of **value** (underlined in blue) for each of the **Dice** in **Game** (values are ANCHOR, HEART, and ANCHOR same as in the console window):

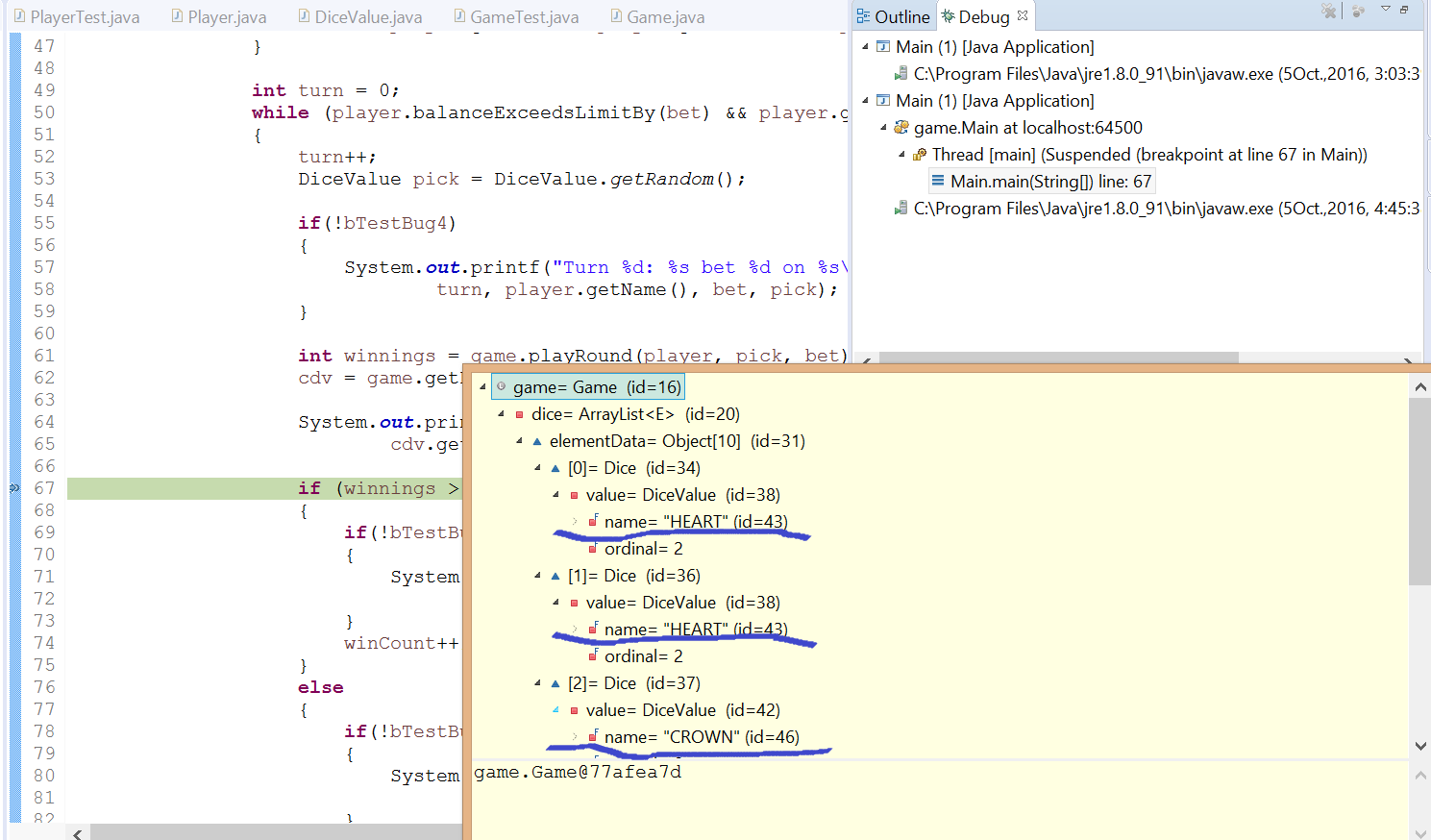


### Third run

Showing the console window (Rolled HEART, HEART, CROWN):



Showing the values of **value** (underlined in blue) for each of the **Dice** in **Game** (values are HEART, HEART, and CROWN same as in the console window):



In all three runs of the program, the output to the console was identical to the **value** of the **Dice** in **Game**. It could be a massive coincidence, but having done it three times and with the static review corroborating this conclusion, I think we can say hypothesis 3 is proven.

## Summary of testing results

The variable value of **Dice** was infected when the return value of getRandom was not saved in roll.

# Resolution

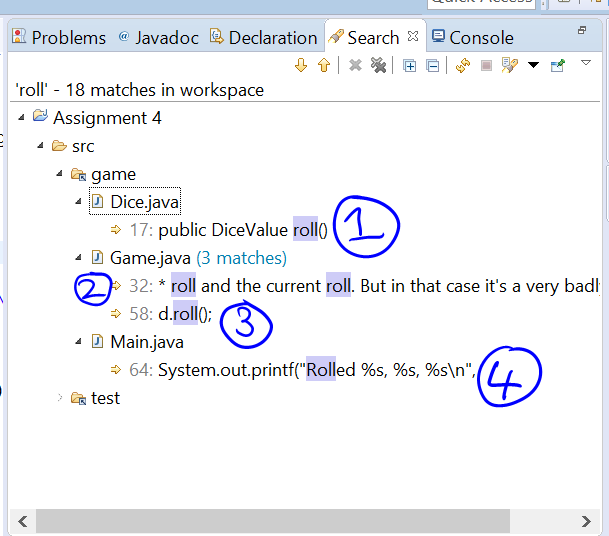
The cause of the bug is that the result of **DiceValue**’s **getRandom** is not saved anywhere when being called by **roll** (**roll** returns it, but is not captured, and the value of **Dice** is used as the roll instead). The solution should then be to save the result of **DiceValue**’s **getRandom** to **Dice**’s **value** in the **roll** method.

## Risk assessment

Could this solution introduce any new defects?

It is possible that somewhere in the program, perhaps **roll** is being called (and the result captured to be used for something), and that section relies on this result being different to the value of **value**. This would mean something bad could happen if we made **value** be equal to the result of **roll**.

We can check for this by running a search on “roll”. If **roll** is not being called anywhere in the program (besides in **playRound**, where we know the result is not being captured), then we know that this proposed solution is safe.



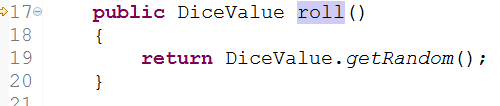
Here are all the possibly relevant locations of the word “roll” in the code (test package is not expanded as this won’t be relevant to the main running of the program).

1. This is the declaration of roll()
2. This is just a comment
3. This is where it is being called in **Game**’s **playRound**
4. This is where the results of the roll are printed to the console

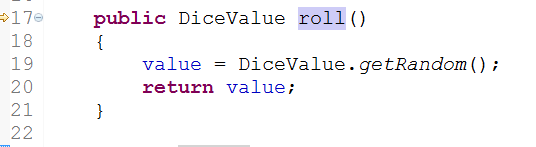
As we can see, there are no places where **roll** is called outside of what we have already investigated. Therefore, this solution is safe.

## Testing the solution

We are going from this:

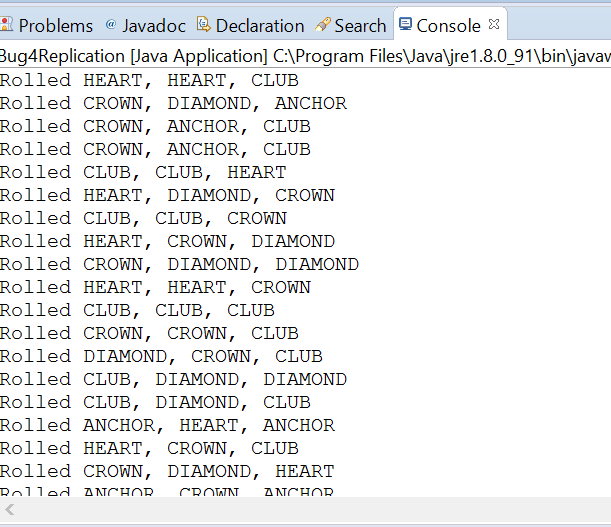


To this:



The reason we are not making it a void method, which would be the logical thing to do (as we don’t need to capture the result outside of the instance), is because this would break GameTest.java (as it has some mocking of this method). Since there is plenty of room for improvement in the code, but we are not improving them all, and only going after the bugs, this should be OK. (Otherwise we should be losing marks every time we spot an inefficiency or strange code choice and don’t fix it – but the assignment is only about investigating and resolving bugs, and willy-nilly changing of code for reasons other than this has been discouraged. So it would be completely illogical to lose marks for leaving this inefficiency in, unless the assignment is being hypocritical).

We originally simplified the problem by writing Bug4Replication.java. This ran **main** as normal, but stripped out all the status messages, just leaving the rolls. If we refer to either Bug4ReplicationSample1.txt, Bug4ReplicationSample2.txt or Bug4ReplicationSample3.txt, we can see that the rolls for each run are all identical. Let’s see what happens now that we have implemented this proposed solution.



As we can see, we get a good mix of roll results now (besides the lack of spades – but that is addressed by another bug).

# Conclusion

The bug is now resolved, and no new defects have been introduced by the resolution. Sample successful results can be seen in Bug4Solution.txt.