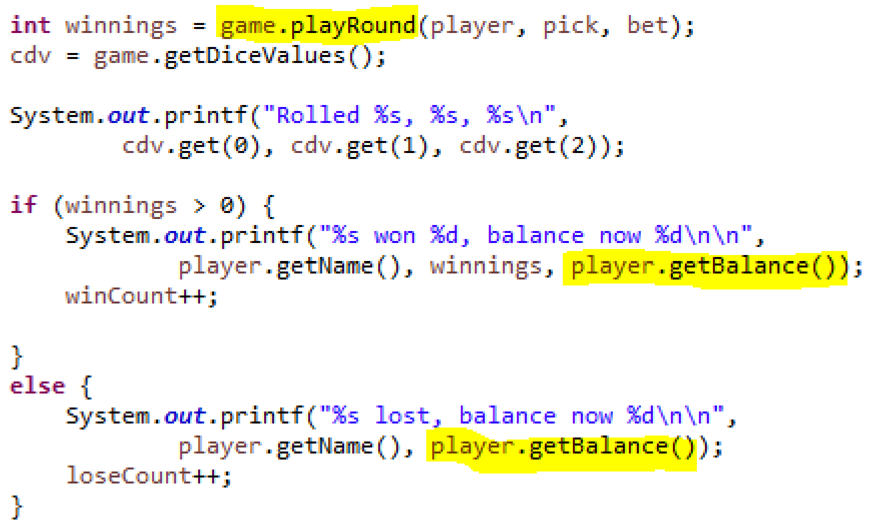
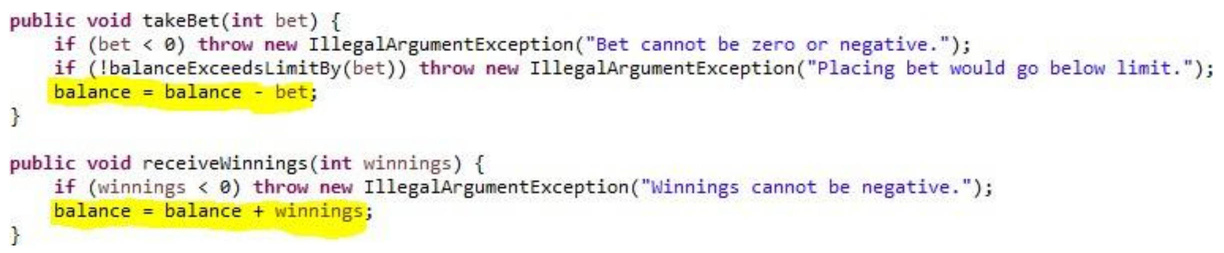
1. Tracing

From **Main.java**, it can be traced that the winning is calculated in the method *playRound* in **Game.java.** Method *getBalance* in **Player.java** is used to display winning.



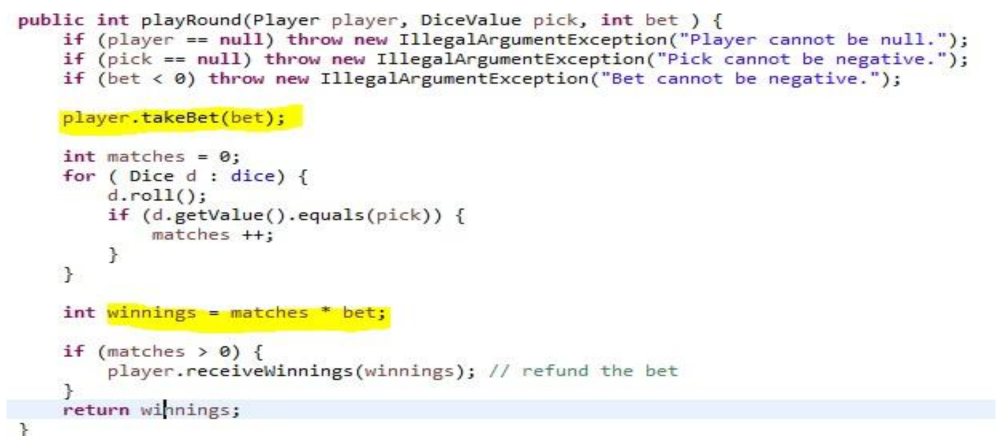
```java
int winnings = game.playRound(player, pick, bet);
cdv = game.getDiceValues();

System.out.printf("Rolled %s, %s, %s\n",
        cdv.get(0), cdv.get(1), cdv.get(2));

if (winnings > 0) {
    System.out.printf("%s won %d, balance now %d\n\n",
            player.getName(), winnings, player.getBalance());
    winCount++;

}
else {
    System.out.printf("%s lost, balance now %d\n\n",
            player.getName(), player.getBalance());
    loseCount++;
}
```

Tracing from the **Player.java**, Balance is calculated as:



```java
public void takeBet(int bet) {
    if (bet < 0) throw new IllegalArgumentException("Bet cannot be zero or negative.");
    if (!balanceExceedsLimitBy(bet)) throw new IllegalArgumentException("Placing bet would go below limit.");
    balance = balance - bet;
}

public void receiveWinnings(int winnings) {
    if (winnings < 0) throw new IllegalArgumentException("Winnings cannot be negative.");
    balance = balance + winnings;
}
```

Tracing from the **Game.java**, the *playRound* method is calculated as



```java
public int playRound(Player player, DiceValue pick, int bet ) {
    if (player == null) throw new IllegalArgumentException("Player cannot be null.");
    if (pick == null) throw new IllegalArgumentException("Pick cannot be negative.");
    if (bet < 0) throw new IllegalArgumentException("Bet cannot be negative.");

    player.takeBet(bet);

    int matches = 0;
    for ( Dice d : dice) {
        d.roll();
        if (d.getValue().equals(pick)) {
            matches ++;
        }
    }

    int winnings = matches * bet;

    if (matches > 0) {
        player.receiveWinnings(winnings); // refund the bet
    }
    return winnings;
}
```

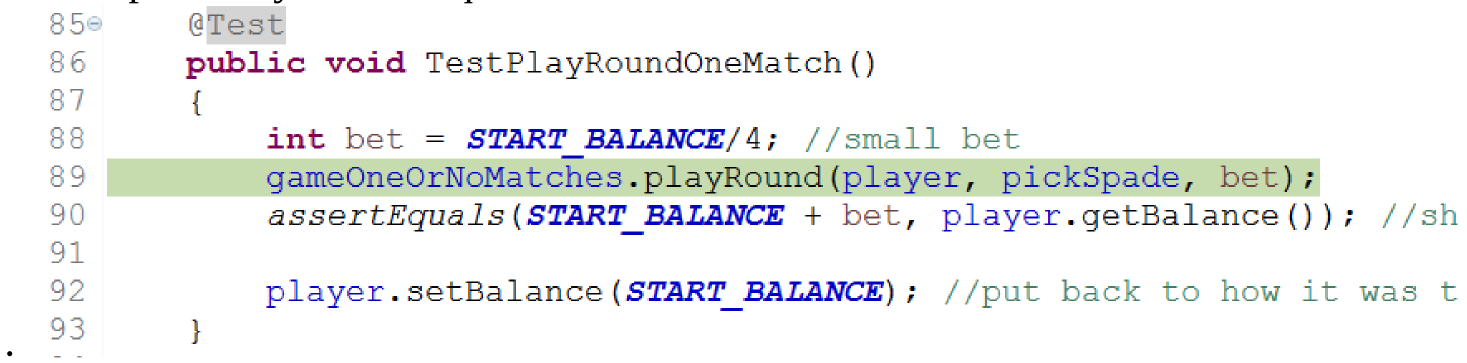
1 Hypothesis

There are three hypotheses in this case that needs to be verified:

1.The value of the dice is compared successfully to pick and the correct number of matches is calculated.
2.The player’s balance correctly increases by the winnings amount after **receiveWinnings** is executed.
3.The balance is incorrect at the time of executing **receiveWinnings**.

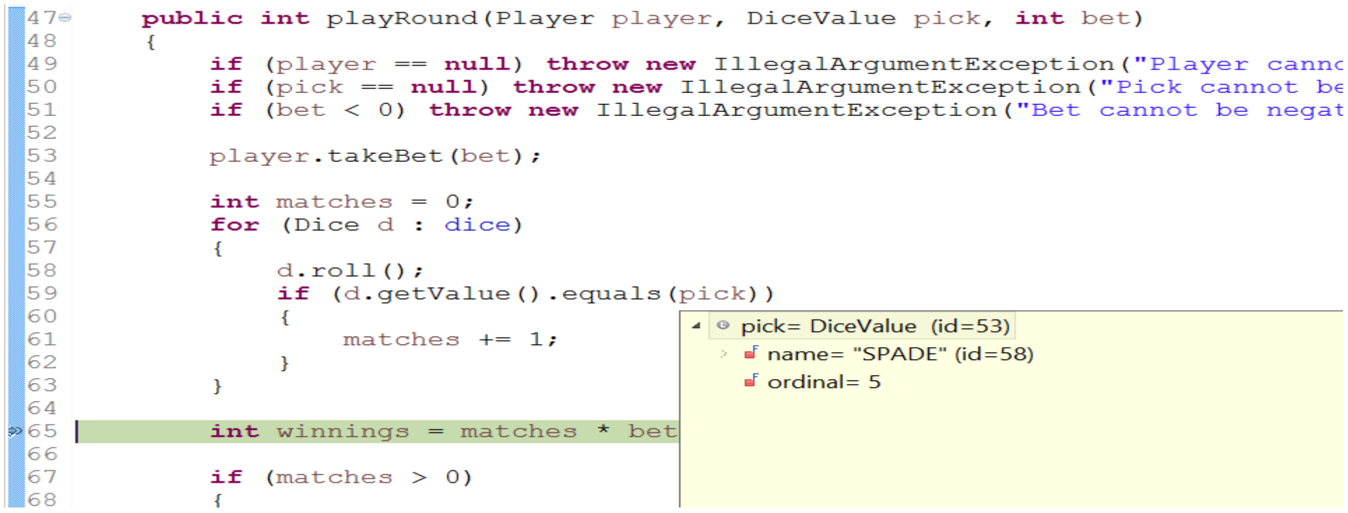
Hypothesis 1:

For this one, the unit test of the **Game** class is used, and place a breakpoint in the appropriate test (TestPlayRoundOneMatch, etc). This way the dice are guaranteed to come up the way that is required



```
    @Test
    public void TestPlayRoundOneMatch()
    {
        int bet = START_BALANCE/4; //small bet
        gameOneOrNoMatches.playRound(player, pickSpade, bet);
        assertEquals(START_BALANCE + bet, player.getBalance()); //sh

        player.setBalance(START_BALANCE); //put back to how it was t
    }
```

Testing one match



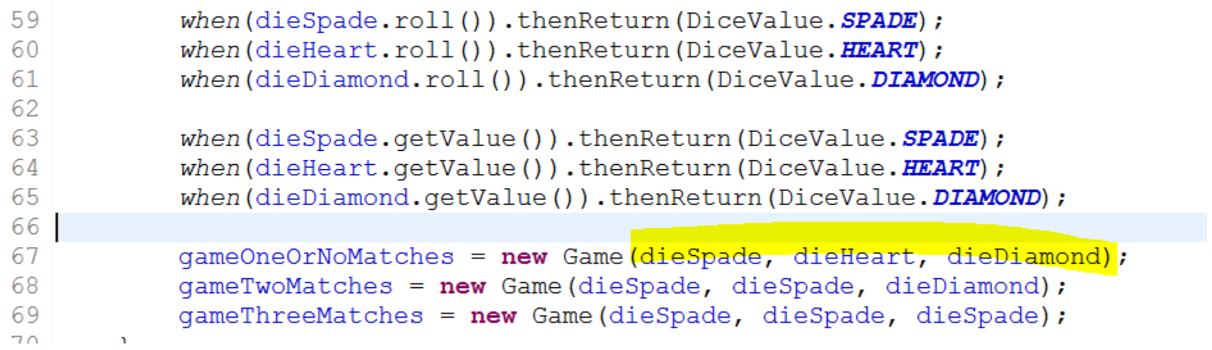
```
    public int playRound(Player player, DiceValue pick, int bet)
    {
        if (player == null) throw new IllegalArgumentException("Player cannc
        if (pick == null) throw new IllegalArgumentException("Pick cannot be
        if (bet < 0) throw new IllegalArgumentException("Bet cannot be negat

        player.takeBet(bet);

        int matches = 0;
        for (Dice d : dice)
        {
            d.roll();
            if (d.getValue().equals(pick))
            {
                matches += 1;
            }
        }

        int winnings = matches * bet

        if (matches > 0)
        {
```

```
pick= DiceValue  (id=53)
    name= "SPADE" (id=58)
    ordinal= 5
```

Going into the right test:

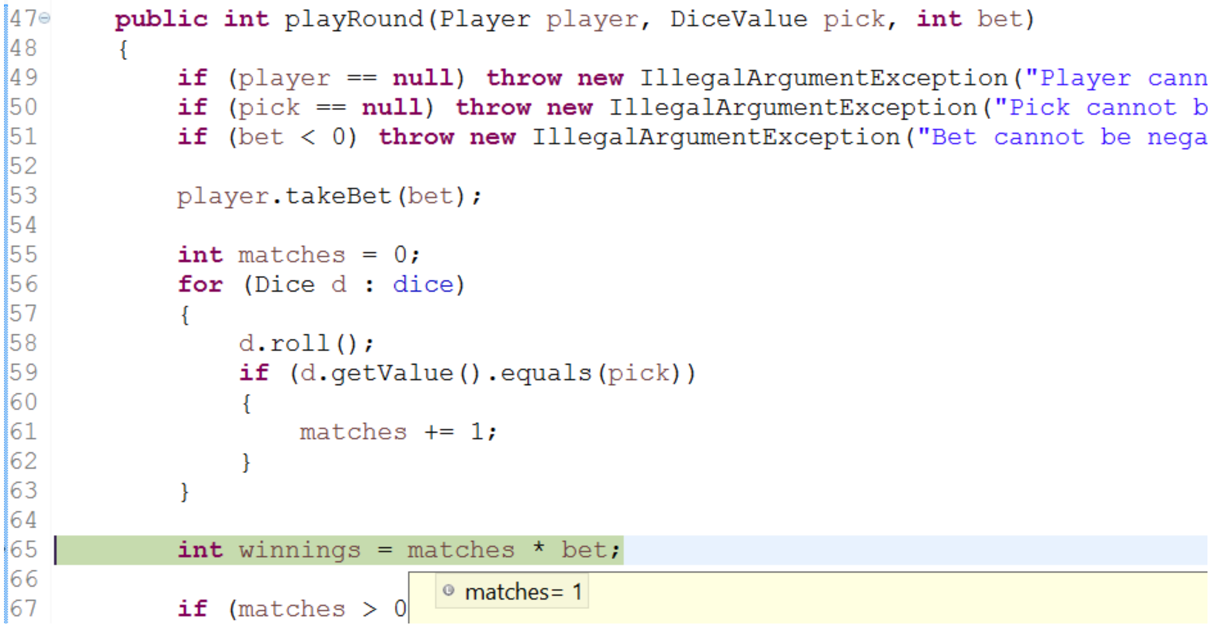
The pick is a Spade.

We can’t actually show the values of dice because they are mocked objects. But in the constructor of GameTest:



```
        when(dieSpade.roll()).thenReturn(DiceValue.SPADE);
        when(dieHeart.roll()).thenReturn(DiceValue.HEART);
        when(dieDiamond.roll()).thenReturn(DiceValue.DIAMOND);

        when(dieSpade.getValue()).thenReturn(DiceValue.SPADE);
        when(dieHeart.getValue()).thenReturn(DiceValue.HEART);
        when(dieDiamond.getValue()).thenReturn(DiceValue.DIAMOND);

        gameOneOrNoMatches = new Game(dieSpade, dieHeart, dieDiamond);
        gameTwoMatches = new Game(dieSpade, dieSpade, dieDiamond);
        gameThreeMatches = new Game(dieSpade, dieSpade, dieSpade);
```

You can see that gameOneOrNoMatches will roll a spade, a heart, and a diamond, and the player picks spade. So, we should get one match here.



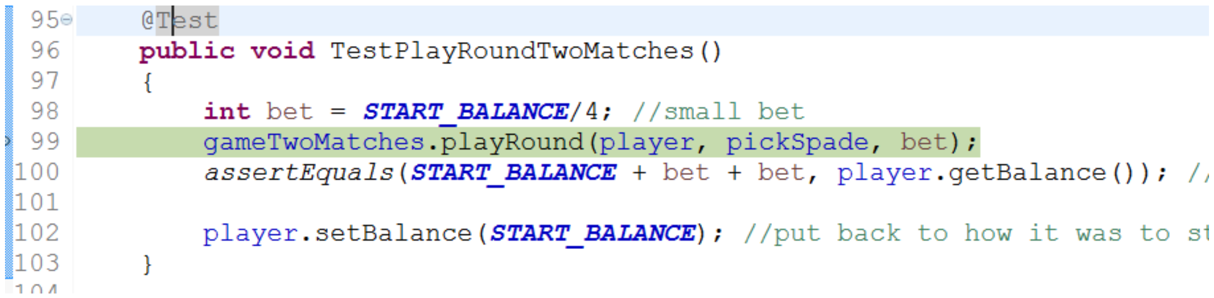
```
    public int playRound(Player player, DiceValue pick, int bet)
    {
        if (player == null) throw new IllegalArgumentException("Player cann
        if (pick == null) throw new IllegalArgumentException("Pick cannot b
        if (bet < 0) throw new IllegalArgumentException("Bet cannot be nega

        player.takeBet(bet);

        int matches = 0;
        for (Dice d : dice)
        {
            d.roll();
            if (d.getValue().equals(pick))
            {
                matches += 1;
            }
        }

        int winnings = matches * bet;

        if (matches > 0
```

```
matches= 1
```

And indeed we do get one match.

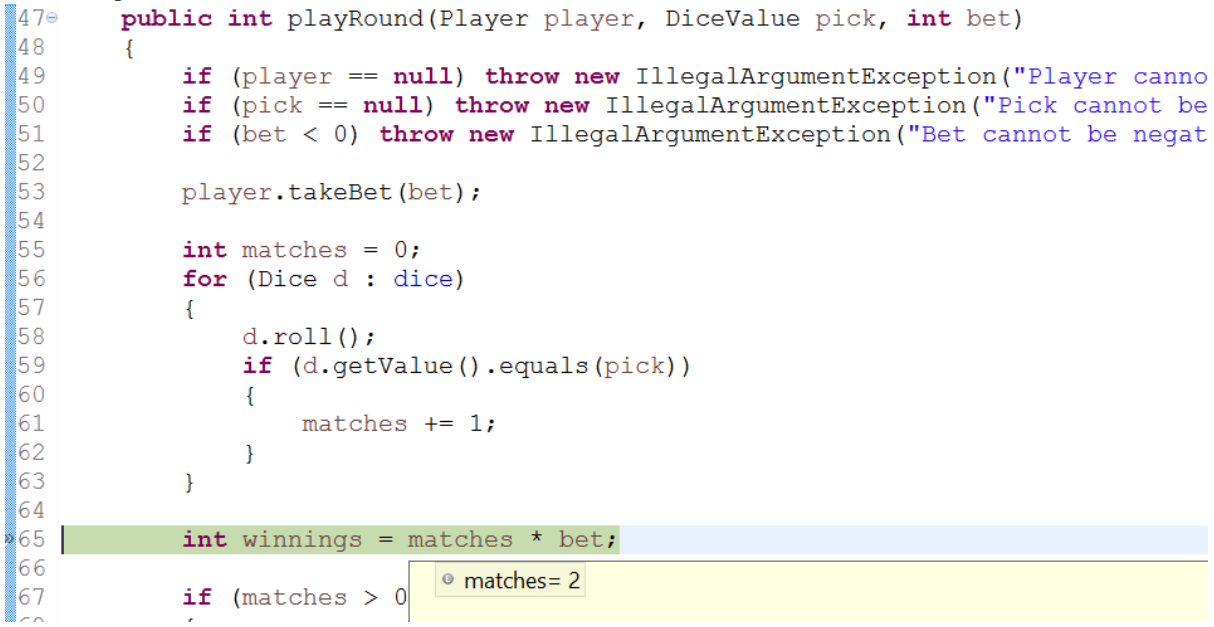
Testing two matches

Going into the right test:



```
    @Test
    public void TestPlayRoundTwoMatches()
    {
        int bet = START_BALANCE/4; //small bet
        gameTwoMatches.playRound(player, pickSpade, bet);
        assertEquals(START_BALANCE + bet + bet, player.getBalance()); /

        player.setBalance(START_BALANCE); //put back to how it was to st
    }
```

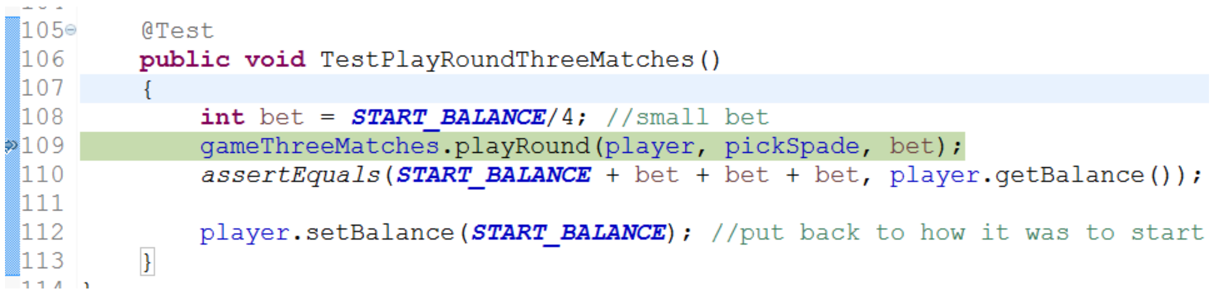
The right number of matches:



```
    public int playRound(Player player, DiceValue pick, int bet)
    {
        if (player == null) throw new IllegalArgumentException("Player canno
        if (pick == null) throw new IllegalArgumentException("Pick cannot be
        if (bet < 0) throw new IllegalArgumentException("Bet cannot be negat

        player.takeBet(bet);

        int matches = 0;
        for (Dice d : dice)
        {
            d.roll();
            if (d.getValue().equals(pick))
            {
                matches += 1;
            }
        }

        int winnings = matches * bet;

        if (matches > 0
```

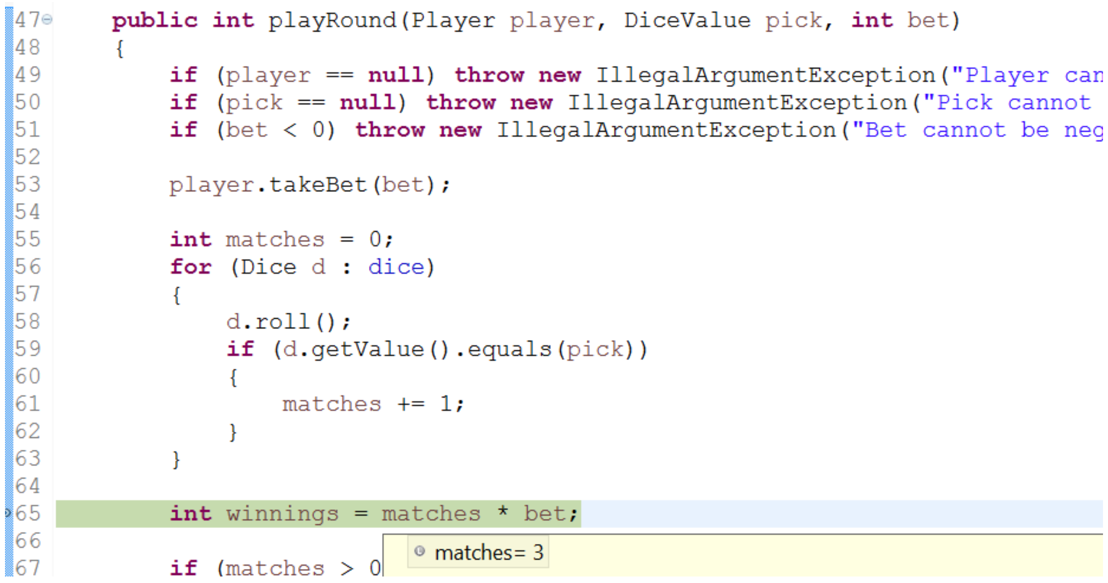
matches= 2

Testing three matches



```
    @Test
    public void TestPlayRoundThreeMatches()
    {
        int bet = START_BALANCE/4; //small bet
        gameThreeMatches.playRound(player, pickSpade, bet);
        assertEquals(START_BALANCE + bet + bet + bet, player.getBalance());

        player.setBalance(START_BALANCE); //put back to how it was to start
    }
```

Going into the right test:



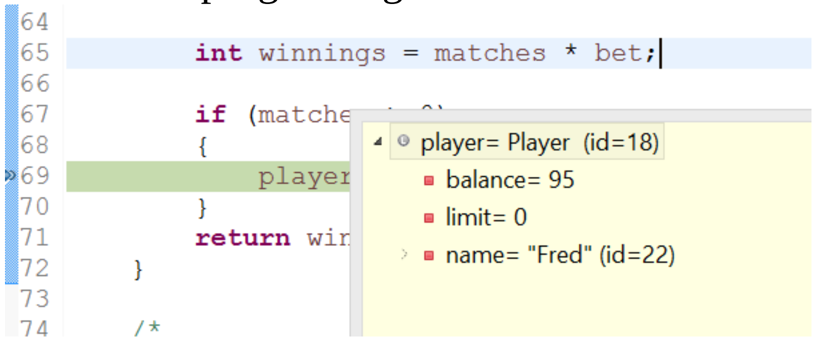
```
    public int playRound(Player player, DiceValue pick, int bet)
    {
        if (player == null) throw new IllegalArgumentException("Player can
        if (pick == null) throw new IllegalArgumentException("Pick cannot
        if (bet < 0) throw new IllegalArgumentException("Bet cannot be neg

        player.takeBet(bet);

        int matches = 0;
        for (Dice d : dice)
        {
            d.roll();
            if (d.getValue().equals(pick))
            {
                matches += 1;
            }
        }

        int winnings = matches * bet;

        if (matches > 0
```

```
matches= 3
```

Right number of matches:

So hypothesis 1 is verified. Game’s playRound can determine the correct number of matches.

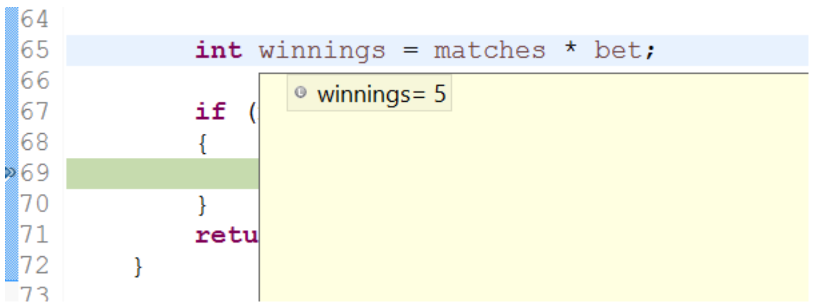
Hypothesis 2
For this one, we will put a breakpoint before receiveWinnings is called, run the program normally and see what the balance and the winnings amount is. We will then step to after it is called, and check that the balance increased by the right amount.
Here is the program right before it calls receiveWinnings:



```
        int winnings = matches * bet;

        if (matche
        {
            player
        }
        return wir
    }

    /*
```

```
player= Player (id=18)
    balance= 95
    limit= 0
    name= "Fred" (id=22)
```

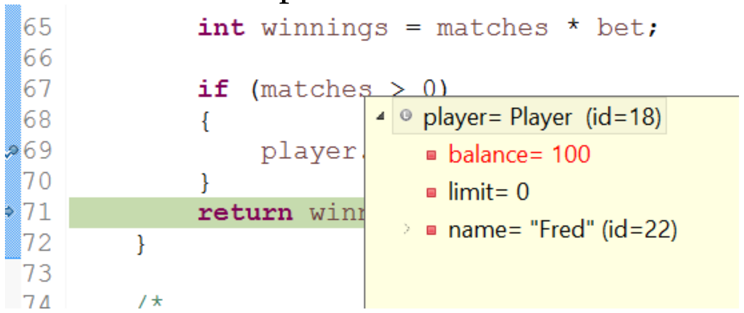
As you can see, the player’s balance is 95.
We also need to know the winnings:



```
        int winnings = matches * bet;

        if (
        {

        }
        retu
    }
```

```
winnings= 5
```

So we would expect that the balance after calling receiveWinnings would be 100:



```
        int winnings = matches * bet;

        if (matches > 0)
        {
            player.
        }
        return winn
    }

    /*
```

```
player= Player  (id=18)
    balance= 100
    limit= 0
    name= "Fred" (id=22)
```

And indeed it is. This verifies hypothesis 2.

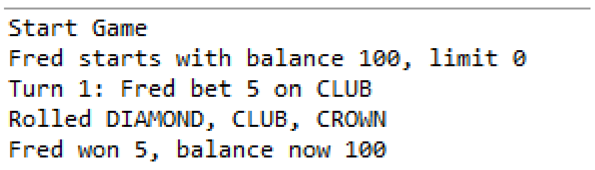
Hypothesis 3

To test these ones, we put breakpoints/step as follows:

1. At the end of **playRound**, and have a look at the player's balance.
2. Then, step to just before **takeBet** and make sure the balance is still the same (the previous breakpoint is to make sure that it hasn't been changed by anything in between the last round and this one – so it is certain that the infection happens inside the **playRound** method and not outside it).
3. Then, step to just after to confirm that the balance has changed to the incorrect value.
4. Then, step to just before **receiveWinnings**, to make sure the balance hasn't changed

The debugging shows the results for the hypotheses as below:

- Result from the console



```
Start Game
Fred starts with balance 100, limit 0
Turn 1: Fred bet 5 on CLUB
Rolled DIAMOND, CLUB, CROWN
Fred won 5, balance now 100
```