Debugging Loops

by Sophia

WHAT'S COVERED

This lesson focuses on debugging issues that can occur with creating and using loops. Specifically, this lesson covers:

Table of Contents

1. Infinite Loop

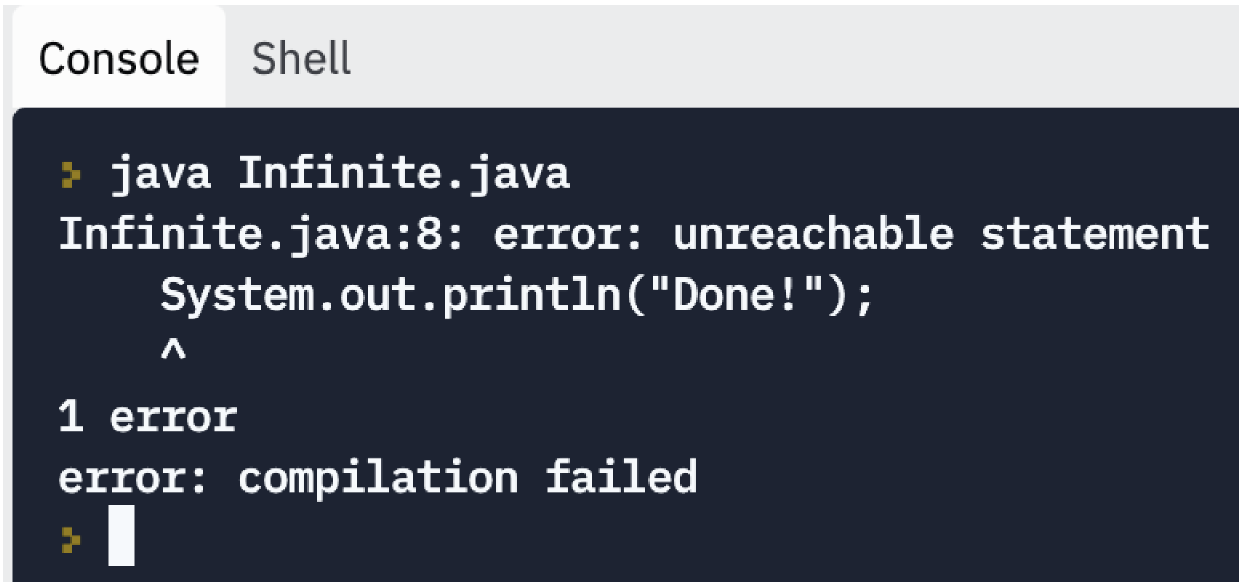
An endless source of amusement for programmers is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an **infinite loop**. In this example, there is no iteration variable telling you how many times to execute the loop. In programming, an infinite loop is a loop that includes no mechanism for ending the loop.

TRY IT

Directions: Sometimes Java will catch an infinite and treat it as an error. Run the following code:

```
class Infinite {
  public static void main(String[] args) {
    int n = 10;
    while(true) {
      // Print out n & then decement it
      System.out.print(n-- + " ");
    }
    System.out.println("Done!");
  }
}
```

Trying to run this code produces a compiler error:



```
> java Infinite.java
Infinite.java:8: error: unreachable statement
        System.out.println("Done!");
        ^
1 error
error: compilation failed
>
```

REFLECT

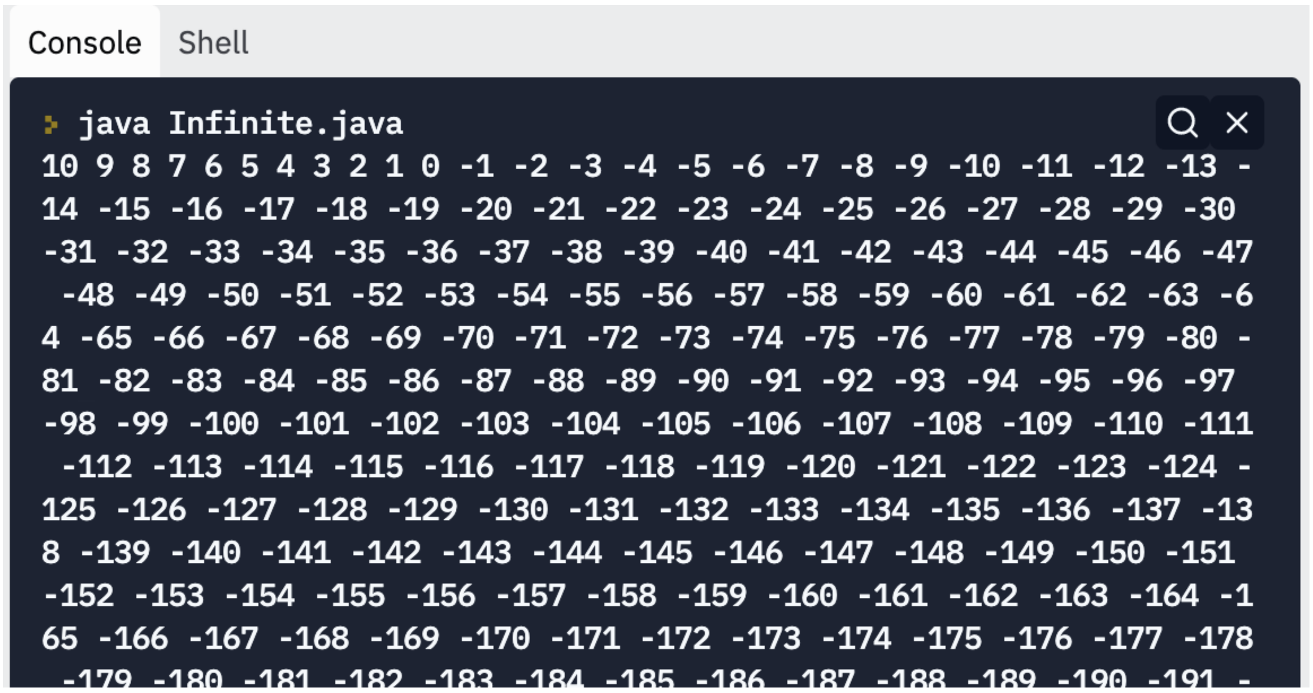
The detection of a line of code that can never be reached is a good catch by the compiler that calls attention to the problem of an infinite loop.

TRY IT

Directions: If the println() statement is removed or commented out, though, the compiler will no longer object and the loop will run forever. Run the code below:

```
class Infinite {
  public static void main(String[] args) {
    int n = 10;
    while(true) {
      // Print out n & then decement it
      System.out.print(n-- + " ");
    }
    //System.out.println("Done!");
  }
}
```

As this screen shot shows, the loop will now run indefinitely:



```
> java Infinite.java
10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -
14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 -27 -28 -29 -30
-31 -32 -33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43 -44 -45 -46 -47
 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -6
4 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 -76 -77 -78 -79 -80 -
81 -82 -83 -84 -85 -86 -87 -88 -89 -90 -91 -92 -93 -94 -95 -96 -97
-98 -99 -100 -101 -102 -103 -104 -105 -106 -107 -108 -109 -110 -111
 -112 -113 -114 -115 -116 -117 -118 -119 -120 -121 -122 -123 -124 -
125 -126 -127 -128 -129 -130 -131 -132 -133 -134 -135 -136 -137 -13
8 -139 -140 -141 -142 -143 -144 -145 -146 -147 -148 -149 -150 -151
-152 -153 -154 -155 -156 -157 -158 -159 -160 -161 -162 -163 -164 -1
65 -166 -167 -168 -169 -170 -171 -172 -173 -174 -175 -176 -177 -178
 -179 -180 -181 -182 -183 -184 -185 -186 -187 -188 -189 -190 -191 -
```

REFLECT

If you make the mistake and run this code, you will learn quickly how to stop a runaway Java process on your system or find where the power-off button is on your computer. In Replit, you can press Ctrl - C in the console to end the program.

We could have also done the same thing by creating a condition where it is always true without using the true value. Since n is starting at 10 and the value is decrementing, in the example below, n will always be < 20, so it will loop infinitely in the same way that a true boolean constant would.

TRY IT

Directions: Here is an example of an infinite loop being run the same as a true boolean constant. Run the code below:

```
class Infinite {
  public static void main(String[] args) {
    int n = 10;
    while(n < 20) {
      // Print out n & then decement it
      System.out.print(n-- + " ");
    }
    System.out.println("Done!");
  }
}
```

While this is a dysfunctional infinite loop, you can still use this pattern to build useful loops. This requires you to carefully add code to the body of the loop. This code should explicitly exit the loop using a break statement when we have reached the exit condition where the loop condition evaluates to false.

REFLECT

An incorrectly constructed loop in one program or context may be the correct loop to use in different code.

Can you think of a situation where the loop above could be useful?

TRY IT

Directions: For example, suppose you want to take input from the user until they type “done.” Write the following:

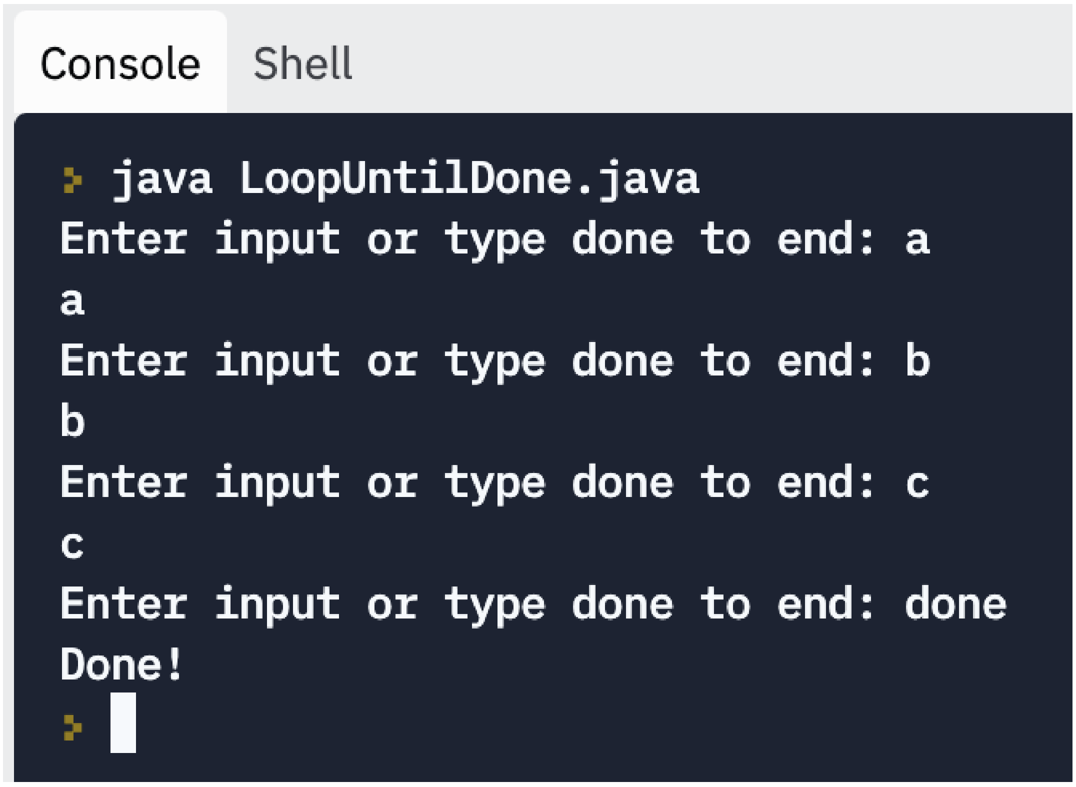
```
import java.util.Scanner;

class LoopUntilDone {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    while(true) {
      System.out.print("Enter input or type done to end: ");
      String line = input.nextLine();
      // Check if line is "done" (ignoring case)
      if(line.equalsIgnoreCase("done")) {
        break; // end loop when done has been entered
      }
      else {
        // Print out entry is not equal to "done"
        System.out.println(line);
      }
    }
    System.out.println("Done!");
  }
}
```

REFLECT

The loop condition is true, which is always true, so the loop runs repeatedly until it hits the break statement. The compiler does not generate an error message in this case because it can see that there is a way to end the loop and allow the code after the loop to run.

Each time through, the body of the loop prompts the user for input. If the user types “done,” the break statement exits the loop. If not done, the program prints out whatever the user types and goes back to the top of the loop. Here’s a sample run:



```
> java LoopUntilDone.java
Enter input or type done to end: a
a
Enter input or type done to end: b
b
Enter input or type done to end: c
c
Enter input or type done to end: done
Done!
>
```

TRY IT

Directions: Try typing the code above (in a .java file with a name that matches the name of the class in spelling and capitalization) and run the program a few times, testing out the word "done" to exit.

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top), and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

HINT

Finish iterations with the break statement.

Sometimes when in an iteration of a loop, there is a need to finish the current iteration and immediately jump to the next iteration. In that case, use the continue statement to skip to the next iteration, without finishing the body of the loop for the current iteration.

REFLECT

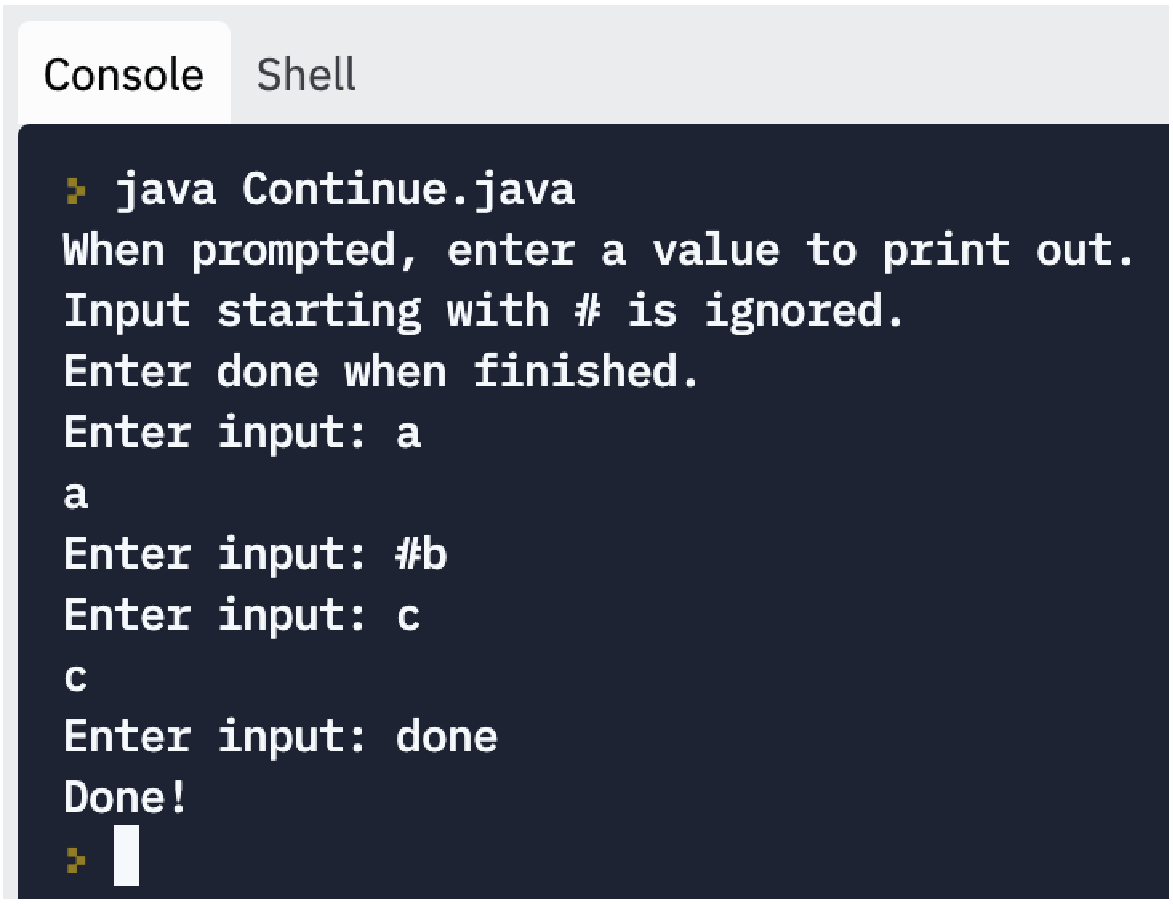
We have seen how the continue and break statements alter how a loop runs. Can you see how these statements play an especially important role in loops like the one above that are set up to run as infinite loops without such intervention?

Below you can see a loop that prints out its user input until the user types "done," but treats lines that start with the hash character as lines not to be printed, kind of like Java comments:

```
while True:
   line = input('Enter input (# tag will not print). Type done when finished > ')
   if line[0] == '#':
      continue
```

```
    if line == 'done':
        break
    print(line)
print('Done!')
```

Here is a sample run of this new program with continue statement added:



```
> java Continue.java
When prompted, enter a value to print out.
Input starting with # is ignored.
Enter done when finished.
Enter input: a
a
Enter input: #b
Enter input: c
c
Enter input: done
Done!
>
```

All the lines are printed except the one that starts with the hash sign (#). This happens when the continue statement is executed. It ends the current iteration and jumps back to the beginning of the while loop. This starts the next iteration, skipping the System.out.println() method call.

TRY IT

Directions: Go ahead and try the program with the continue statement. Notice that the program will only check if the user input starts with a hashtag. Using the symbol later in the sentence will not have the same effect.

REFLECT

When working with loops in Java code, the break statement is probably used more often than the continue statement. Think about why this would be the case.

BIG IDEA

As you write bigger programs, you may find yourself spending more time debugging. More code means more chances to make an error. There are more places for bugs to “hide.” One way to cut your debugging time is “debugging by bisection.”

For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

TERM TO KNOW

Infinite Loop
A loop that does not include a means for ending the loop, so the loop will run forever.

2. Other Common Issues

Debugging by bisection is the practice of breaking program code into “sections” for debugging and validation purposes. This will help speed up the debugging process. It may be able to find issues within a section rather than focusing on the full program. This shortens the time to debug, if no issues are present in earlier sections.

When starting, consider breaking the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a System.out.println() statement and run the program. If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice, it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

BIG IDEA

Debugging by bisection is especially important if you’ve made changes to an existing program. You most likely will not need to test the whole program prior to where the change is made but only on everything afterwards.

TERM TO KNOW

Debugging By Bisection
Debugging by bisection is the practice of breaking your program code into “sections” for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program.

SUMMARY

In this tutorial, you learned about **infinite loops**. You learned why they are an issue, since they will not stop running unless you have an option to stop the process, like the stop button in Replit. It is a best practice to always create a loop that has some means of being able to exit. In most cases, infinite loops are errors that should be resolved. You also learned about other **common issues** when using loops.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Debugging By Bisection

Debugging by bisection is the practice of breaking your program code into “sections” for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program.

Infinite Loop

A loop that does not include a means for ending the loop, so the loop will run forever.