

Debugging with Karel

Based on a handout by Mehran Sahami, Eric Roberts and Nick Parlante.

Much of your time as a computer programmer will likely be spent debugging. This phenomenon is best described by a quotation from one of the first computer pioneers, Maurice Wilkes:

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

— Maurice Wilkes, 1949

In order to be better prepared to undertake the more complex future debugging that you will be doing, we aim to give you here both a sense of the philosophy of debugging as well as to teach you how to use some of the practical tips that make testing and debugging easier.

The Philosophy of Debugging

Debugging is one of the most creative and intellectually challenging aspects of programming. It can also be one of the most frustrating. To a large extent, the problems that people face debugging programs are not so much technical as they are psychological. To become successful debuggers, you must learn to think in a different way. There is no cookbook approach to debugging, although Nick Parlante's 11 Truths of Debugging (given below) will probably help. What you need is insight, creativity, logic, and determination.

As computer scientists, it is important to remember that the programming process leads you through a series of tasks and roles:

<u>Task</u>	—	<u>Role</u>
Design	—	Architect
Coding	—	Engineer
Testing	—	Vandal
Debugging	—	Detective

These roles require you to adopt distinct strategies and goals, and it is often difficult to shift your perspective from one to another. Although debugging can often be very difficult, it can be done. It will at times take all of the skill and creativity at your disposal, but you can succeed if you are methodical and don't give up on the task.

Debugging is an important skill that you will use every day if you continue in Computer Science or any related field. Even though it is the final task of those listed above, it is certainly not the least important. You should always plan ahead and allow sufficient time for testing and debugging, as it is required if you expect to produce quality software. In addition, you should make a concentrated effort to develop these skills now, as they will be even more important as programs become more complicated later in the quarter.

The 11 Truths of Debugging

1. Intuition and hunches are great—you just have to test them out. When a hunch and a fact collide, the fact wins. That's life in the city.
2. Don't look for complex explanations. Even the simplest omission or typo can lead to very weird behavior. Everyone is capable of producing extremely simple and obvious errors from time to time. Look at code critically—don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.
3. The clue to what is wrong in your code is in the flow of control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts.
4. Be systematic and persistent. Don't panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.
5. If you code was working a minute ago, but now it doesn't—what was the last thing you changed? This incredibly reliable rule of thumb is the reason you should test your code as you go rather than all at once.
6. Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable in an experiment at a time. It makes the observed behavior much more difficult to interpret, and you tend to introduce new bugs.
7. If you find some wrong code that does not seem to be related to the bug you were tracking, fix the wrong code anyway. Many times the wrong code was related to or obscured the bug in a way you had not imagined.
8. You should be able to explain in Sherlock Holmes style the series of facts, tests, and deductions that led you to find a bug. Alternately, if you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your methods cannot contain the bug. One of these arguments will contain a flaw since one of your methods does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.
9. Be critical of your beliefs about your code. It's almost impossible to see a bug in a method when your instinct is that the method is innocent. Only when the facts have proven without question that the method is not the source of the problem should you assume it to be correct.
10. Although you need to be systematic, there is still an enormous amount of room for beliefs, hunches, guesses, etc. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the methods you suspect the most first. Good instincts will come with experience.
11. Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. *Debugging* code is more mentally demanding than *writing* code. The longer you try to track down a bug without success, the less perspective you tend to have. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep. You cannot debug when you are not seeing things clearly. Many times a programmer can spend hours late at night hunting for a bug only to finally give up at 4:00 A.M. The next day, they find the bug in 10 minutes. What allowed them to find the bug the next day so quickly? Maybe they just needed some sleep and time for perspective. Or maybe their subconscious figured it out while they were asleep. In any case, the “go do something else for a while, come back, and find the bug immediately” scenario happens too often to be an accident.

Based on writing by Nick Parlante, Stanford University

Using the Eclipse Debugger

Because debugging is a difficult but nonetheless critical task, it is important to learn the tricks of the trade. The most important of these tricks is to get the computer to show you what it's doing, which is the key to debugging. The computer, after all, is there in front of you. You can watch it work. You can't ask the computer why it isn't working, but you can have it show you its work as it goes. Modern programming environments like Eclipse come equipped with a **debugger**, which is a special facility for monitoring a program as it runs. By using the Eclipse debugger, for example, you can step through the operation of your program and watch it work. Using the debugger helps you build up a good sense of what your program is doing, and often points the way to the mistake.

Instructions on how the Eclipse debugger works are on the course website's Eclipse page, cs106a.stanford.edu/eclipse. Read through those instructions, then come back here. The remainder of this handout will walk through how to debug a sample (buggy!) version of the **Roomba** Karel program we discussed in lecture. The code for this buggy program is in the code from **Lecture3**, linked to from the Schedule page on the course website at cs106a.stanford.edu/schedule. Download it and follow along!

Figure 2. Buggy version of Roomba

```

/*
 * File: BuggyRoomba.java
 * -----
 * An alternative implementation of Roomba,
 * where Karel alternates going left and
 * right across different rows (labeled as
 * algorithm 2 in the Lecture 3 slides),
 * but with a bug!
 */

import stanford.karel.*;

public class BuggyRoomba extends SuperKarel {

    // Karel cleans up a field of beepers, one row at a time.
    public void run() {
        sweep();
        while (leftIsClear()) {
            moveUpRight();
            sweep();
            if (rightIsClear()) {
                moveUpLeft();
                sweep();
            }
        }
    }

    /*
     * Karel cleans up all beepers in a single row.
     * Pre-condition: Karel is at the beginning of a row.
     * Post-condition: Karel is at the end of the
     *                 same row, but has cleaned up all
     *                 beepers in that row.
     */
    private void sweep() {
        safePickBeeper();
        while (frontIsClear()) {
            move();
            safePickBeeper();
        }
    }

    /*
     * Karel safely picks up a beeper only if there is one
     * on this corner.
     * Pre-condition: None
     * Post-condition: Karel has picked up a beeper on the
     *                 current square if one was present.
     */
    private void safePickBeeper() {
        if (beepersPresent()) {
            pickBeeper();
        }
    }

    // CONTINUED...

```



Figure 2. Buggy version of Roomba (continued)

```

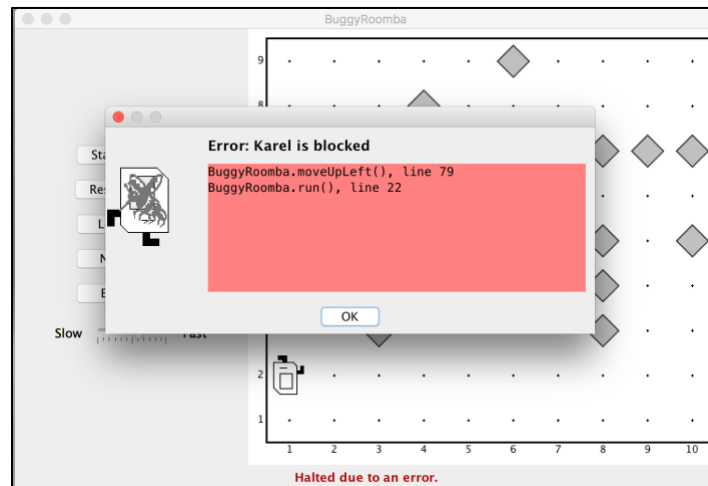
/*
 * Karel moves up on the right to the start of
 * the next row.
 * Pre-condition: Karel is facing east at the
 *                 end of a row.
 * Post-condition: Karel is facing west at the
 *                 beginning of the next row.
 */
private void moveUpRight() {
    turnLeft();
    move();
    turnLeft();
}

/*
 * Karel moves up on the left to the start of
 * the next row.
 * Pre-condition: Karel is facing west at the
 *                 end of a row.
 * Post-condition: Karel is facing east at the
 *                 beginning of the next row.
 */
private void moveUpLeft() {
    move();
    turnRight();
}
}

```

Assessing the Symptoms

Before you start using the debugger, it is always valuable to run the program to get a sense of what the problems might be. If you run the **BuggyRoomba** program and load the **Roomba.w** world, you might see the following sample run:



It looks like Karel is crashing into a wall while cleaning up the second row, which is not good. Luckily, the Eclipse debugger can help us better understand what's going on!

The Eclipse Debugger

When you run a project under Eclipse, you can use the debugger to set breakpoints in your code, which will enable you to stop it at interesting places, proceed through the program one step at a time, and do other useful things.

Debugging, however, is a creative enterprise, and there is no magic technique that tells you what to do. You need to use your intuition, guided by the data that you have. In the **BuggyRoomba** program, your intuition is likely to suggest that the problem is in the logic Karel uses for moving to the next row; in fact, the error message that pops up confirms this! It says that the crash is being caused by line **79** in the **moveUpLeft()** method. Therefore, let's add a **breakpoint** in that method so that next time Karel executes those commands, the program will pause and we can investigate what is going on. Double-click in the sidebar on line 79 to do this.

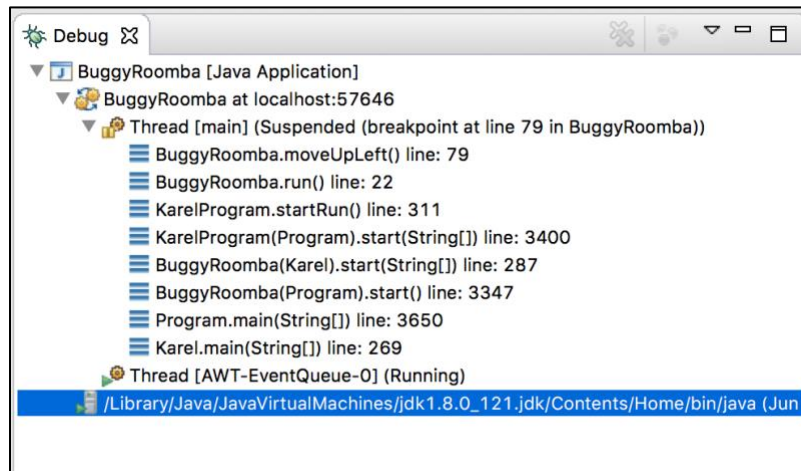
```
78 private void moveUpLeft() {
79     move();
80     turnRight();
81 }
82 }
83
```

Once you've set the breakpoint, run the program as normal. Once Karel is about to move up on the left to the next row, the program calls **moveUpLeft()**, where it switches to debug mode (if you're not in it already) and stops at the breakpoint as shown below.

```
78 private void moveUpLeft() {
79     move();
80     turnRight();
81 }
82 }
83
```

The arrow and the green highlight mark the line of code you are about to execute.

You can also see the **stack trace** above this, which shows where in the program we currently are. You can ignore all the lines except for the ones starting with our program name, **BuggyRoomba**; reading from *bottom to top*, Karel executed the **run()** command, which told Karel to execute the **moveUpLeft()** command, and that's where we are.



The screenshot shows the Eclipse IDE's Debug console. The top part shows the stack trace for the **BuggyRoomba** application. The stack trace is as follows:

- Thread [main] (Suspended (breakpoint at line 79 in BuggyRoomba))
 - BuggyRoomba.moveUpLeft() line: 79
 - BuggyRoomba.run() line: 22
 - KarelProgram.startRun() line: 311
 - KarelProgram(Program).start(String[]) line: 3400
 - BuggyRoomba(Karel).start(String[]) line: 287
 - BuggyRoomba(Program).start() line: 3347
 - Program.main(String[]) line: 3650
 - Karel.main(String[]) line: 269
- Thread [AWT-EventQueue-0] (Running)
 - /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (Jun

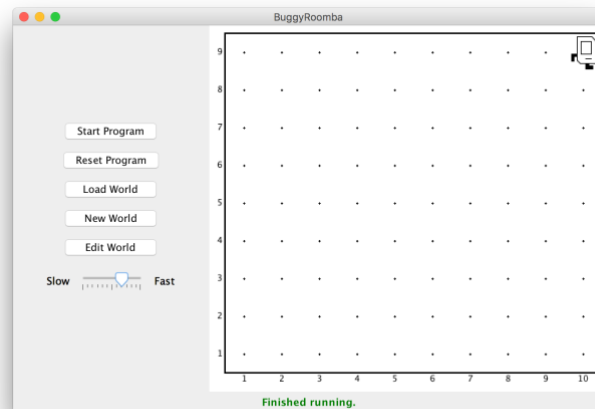
The bottom part of the screenshot shows the **Debug** tab, which displays the current state of the program. The **moveUpLeft()** method is highlighted, and the **move()** command is the next line to be executed.

Looking back at the command we are paused at, something looks suspicious; Karel is facing a wall, but the highlighted instruction (the one coming next) is `move()`! This isn't right; it looks like we forgot to have Karel *turn* before moving to the next row. Let's modify the code for `moveUpLeft()` to look as follows:

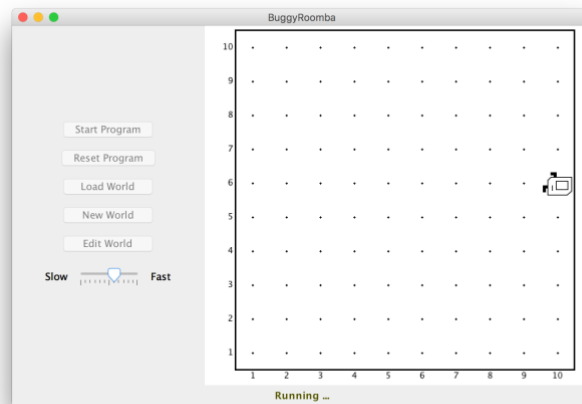
```
private void moveUpLeft() {
    turnRight();      // NEW
    move();
    turnRight();
}
```

This is an example of *incorrectly matching pre/post conditions*. If `run()` had Karel facing north, towards the next row, before calling `moveUpLeft()`, then this code would not be buggy, since that is the apparent pre-condition of `moveUpLeft()`. However, `run()` instead has Karel execute `moveUpLeft()` when facing a wall, so Karel crashes.

Now, we can quit the program and double-click on our breakpoint to remove it. If we run it again in the **Roomba.w** world, it looks like our program is working! Karel cleans up all the beepers, and then finishes executing in the top-right corner of the world.



However, remember that this program should work in worlds of *any* size; let's try running it in the other included world, **Roomba2.w**. On first glance, it seems like it works here as well! Except....



Karel is still going! And going, and going....this is clearly an infinite loop. This is a reminder that you should always test your Karel programs on *as many worlds as possible*. In this case, a bug arises in our program when running in this 10x10 world that did not come up when running in a 9x10 world. Unfortunately, since this is not a crash, we won't get a helpful error message like last time. We will need to do some investigative work.

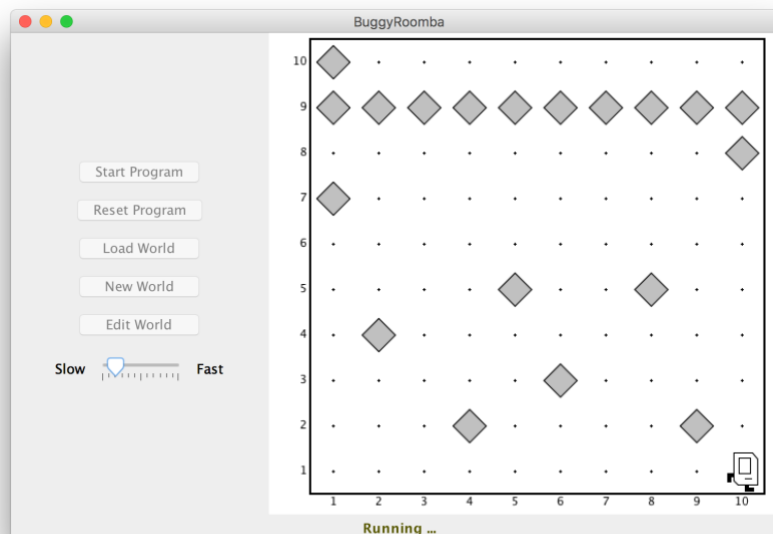
When you see an infinite loop in your program, the first thing you can do is start looking at the loops in your program and try to narrow down which loop is not terminating when you want it to. In this case, we have **two** loops; one while loop within the `sweep()` method to clean a single row of any length, and one while loop within `run()` to repeat cleaning up multiple rows. Let's put a breakpoint on line **19**, which is the first line inside the body of the while loop in `run()`. This way, the program will pause each time the loop executes.

```

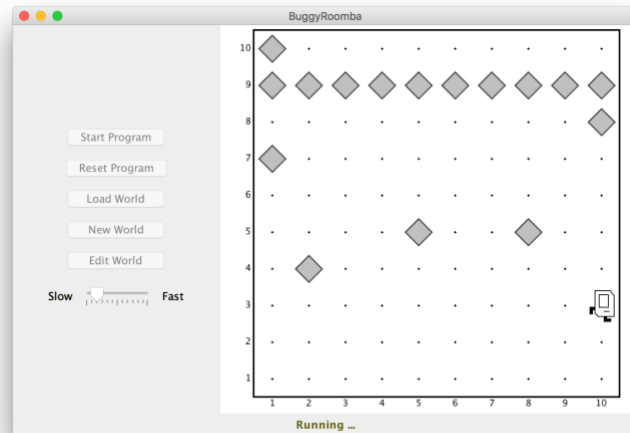
16 public void run() {
17     sweep();
18     while (leftIsClear()) {
19         moveUpRight();
20         sweep();
21         if (rightIsClear()) {
22             moveUpLeft();
23             sweep();
24         }
25     }
26 }

```

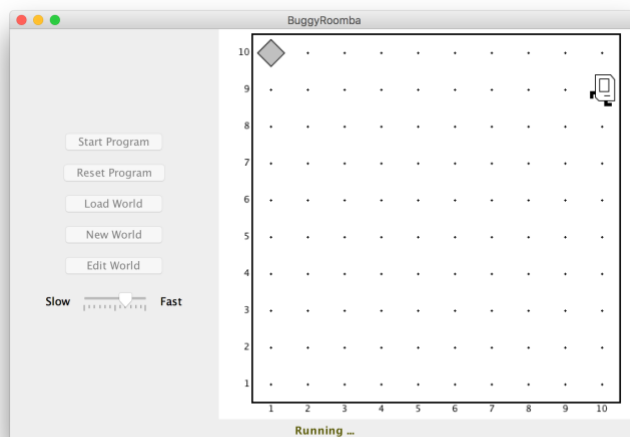
Run the program again; you'll notice that the program pauses as Karel finishes cleaning the first row; this is because `sweep()` executes, and its postcondition (as its comment mentions!) is that Karel is at the end of a row that has now been cleaned.




Karel's left is clear, so it then enters this loop. Click the "Resume" button to have Karel continue; you'll notice that Karel continues until it hits the breakpoint *again*, which is the second time Karel executes the loop.



You can see that each time around the loop, Karel sweeps *two* rows back and forth, which is the intended behavior of this solution. Since Karel’s behavior seems correct until the very end of the program, let’s keep clicking “Resume” until Karel is on the second-to-last row, facing the wall.

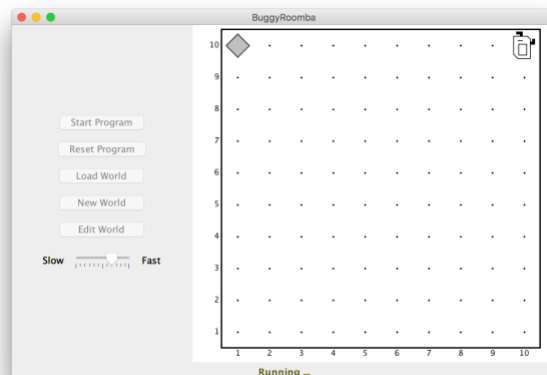


From here, we’re getting closer to isolating the buggy behavior; let’s use the “Step Over” button  to execute `moveUpRight()` and go to the next instruction. Now, the debugger is paused on the line with `sweep()`, and Karel is facing west on the topmost row.

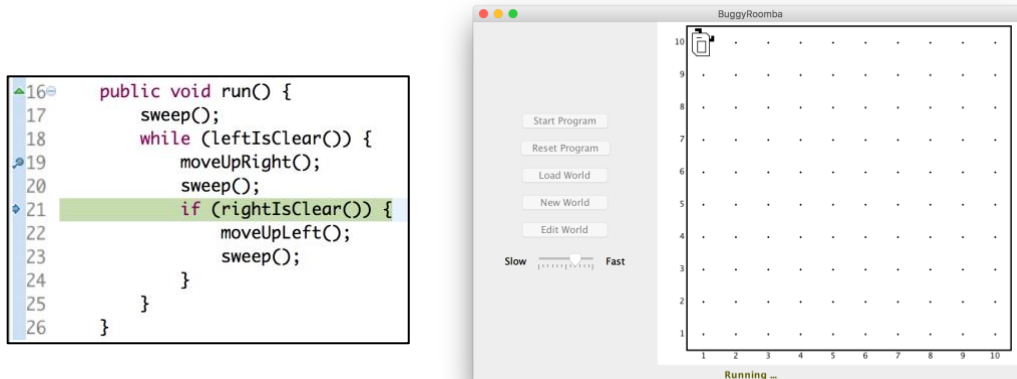
```

16 public void run() {
17     sweep();
18     while (leftIsClear()) {
19         moveUpRight();
20         sweep();
21         if (rightIsClear()) {
22             moveUpLeft();
23             sweep();
24         }
25     }
26 }

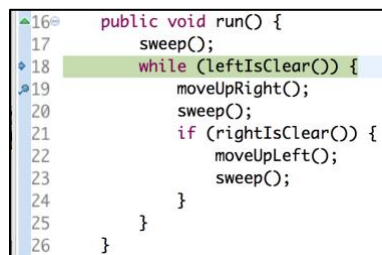
```



Click Step Over again to execute `sweep()`; Karel will sweep the topmost row and end up facing west at the end of the row.



Click Step Over once more; Karel's right is not clear, so the debugger will skip over the if statement and go back to the top of the while loop to check the condition for whether or not to execute the loop again.



Now we start to see the problem; we want this loop to stop now that Karel has swept every row, but Karel's left is still clear! If you click Step Over once more, you'll thus notice that we enter the loop again, and pause on the line `moveUpRight()`. We have found our bug! This while loop is not terminating when we want it to terminate.

The first question you may ask is, "how come we didn't have this bug in the first world?" This is another great exercise in using the debugger! We won't spoil the answer here, but it's a great exercise to try and find out why this bug *doesn't* come up in **Roomba.w**.

The second question you may ask is, "how do we fix the bug?" The answer comes down to figuring out a way to break out of the loop once Karel reaches the end of the final row. The if statement on line 21 attempts to do something like this; if our right is *blocked* after sweeping a row (line 20), we don't want to move up and sweep another row, because there are no more rows. However, because Karel does nothing, we remain facing the wall, and our left is still clear. We need to **make Karel's left blocked** somehow so the while loop will stop. An easy way to do this is to say, "well, if Karel's right is *blocked* at the end of the row, this means there are no more rows; so let's have Karel *face upwards* so that its left is blocked; this way, when we go back up to the top of the while loop, Karel's left will not be clear and the loop will stop. In other words, our run method needs to look like:

```
public void run() {
    sweep();
    while (leftIsClear()) {
        moveUpRight();
        sweep();
        if (rightIsClear()) {
            moveUpLeft();
            sweep();
        } else {
            turnRight();      // NEW
        }
    }
}
```

If you remove the breakpoint and run the program again in the **Roomba2.w** world, we no longer have an infinite loop! Huzzah! Additionally, you'll notice that the program still works in the **Roomba.w** world as well; trying it in as many different worlds as you can is always a good idea.

Now that we're done with this exercise, you can return to "Editor" mode instead of "Debugger" mode, use the Editor button in the Stanford Menu.

Hopefully this gives you a taste of the debugging process, and how the Eclipse debugger can come in handy when working on your programs. It is an extremely useful tool to better understand what your program is doing!