Debugging Tutorial

1004 - Intro to Computer Science and Programming in Java

This is a tutorial on debugging, from both a theoretical perspective and a practical perspective with the Eclipse debugging tool. Written by James Wen.

Table of Contents

Terms and Definitions:

Debugger

- A type of program or plugin into IDEs serves as a tool that helps programmers find errors in their code

Eclipse

- One of the most widely used IDEs for Java and also the IDE that we will be supporting for 1004
- Also allows for installation of plugins that add extra features and support for a large variety of other programming languages

Eclipse Debugger

- The built-in debugger that the Eclipse IDE utilizes. A powerful and relatively easy to use tool for debugging.

Print Statement

- General term for some program statement or instruction that will result in some output indicating either execution flow (that program execution has reached where the print statement is) or information about variables or data.

Control Flow

- The order in which the instructions specified by your code, statements, function calls, etc. will be executed by the machine.

Program Trace/Stack Trace

- The sequence and results of currently executing programs and methods. Java Stack Traces are very helpful when debugging thrown exceptions (errors).

Breakpoint

- Intentional stopping or pausing of a program to allow for user inspection of variables and program state or environment.

Watchpoint

- Breakpoint executed when a specific variable is accessed and/or changed.

Theory:

It's important to note that debugging and fixing bugs when your program does not work as expected is **not** an exercise in randomly or semi-randomly changing parts of your code and rerunning your program to see if it works. Even if one of the changes you make might make the program work as expected, you may have not actually fixed the bug/mistake and only applied a temporary solution. After implementing more code, the original bug may resurface and you will have not really fixed it as you didn't determine the core logic behind the issue.

Effective debugging that actually fixes bugs is more methodical and scientific.

Ex. Consider the Debugging2.java program that is included with this tutorial. It's an implementation of fizzBuzz, a commonly used interview question that is usually asked to discern if you have bare minimal programming knowledge. For fizzBuzz, you are asked to write a program that prints the numbers from 1 to some number. But for multiples of 3 print "Fizz" instead of the number and for the multiples of 5 print "Buzz". For numbers which are multiples of both 3 and 5, print "FizzBuzz".

Determine what the bug actually is.

A software bug is usually some instance in your program execution where what is expected diverges from the actual results, state, etc.

Ex. I expect executing my FizzBuzz program and fizzBuzz method with an input parameter of 15 should cause it to print "FizzBuzz" on the last line. However, when I actually execute the program and fizzBuzz method with an input parameter of 15, no "FizzBuzz" is printed as the last line.

Collect enough information for a hypothesis/guess at what's wrong or causing the error in actual vs. expected.

Find out what information (which classes, variables, methods, etc.) is relevant to the bug or unexpected behavior. Check this information/statuses of these things with print statements or in the debugger.

Ex. My fizzBuzz method is not working as expected. Calling the fizzBuzz method with 15 as the input parameter does not print fizzBuzz for 15. My fizzBuzz method utilizes a for loop that increments based on the index variable i and a series of if else conditionals inside the loop. I'm going to use a breakpoint in this for loop of this method to collect more information about what's happening in each iteration with i and the conditionals.

Make a hypothesis. Guess what is wrong or causing the bug.

Use the information you got from print statements, debugging, etc. to make an educated guess about what might be causing the bug.

Ex. The information I got from breakpoints tells me that i iterates up to 14 inside the for loop. The problem is hence caused by how the condition to continue executing the for loop should be when the i is <= end and not when i < end.

Try to apply a fix.

Change your code to fix what you're estimating is the issue.

Ex. I'll change the i < end conditional to i <= end.

Test the fix.

See if your results/output are correct again and the program now works as expected. Gauge if your expectations of values match their actual values.

Ex. Okay, the program now prints fizzBuzz for 15 so it works as expected now.

Repeat until fixed!

If your guess was wrong or you don't have enough information to really understand the bug, repeat! Get more information, guess what could be wrong, test it, etc. With each fix attempt/tested guess, the domain of elements of your program that could be causing the error get less and less and your understanding of how the program or specific methods work increases.

Print Statement Debugging:

Print statement debugging involves printing to the console values or general statements to get an idea of respectively the status of variables and program execution flow.

Example Use Cases:

- Printing out the control variable (i or index or etc.) within each iteration of a loop.
- Printing out the value of a variable to be returned at the end of a method.
- Printing out a test statement ("test") within a series of if-else statements to determine which if is executed.

toString method:

It's important to note that when you pass any object into the System.out.println or System.out.print method calls, it will invoke the toString() method of that object. If there is no toString() method, then it will print a memory location (which isn't usually useful). This is a bit of motivation to add meaningful toString methods to the classes you write to assist with debugging, logging, etc.

Some Print Statement commands/representations:

```
System.out.println();
System.out.print();
(Error Printing)
(Logging)
```

Eclipse Debugger Setup/Start:

- To Start the Eclipse Debugger, click the Debug button in the top right corner of the window.

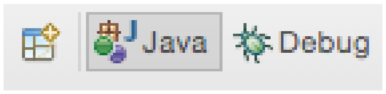
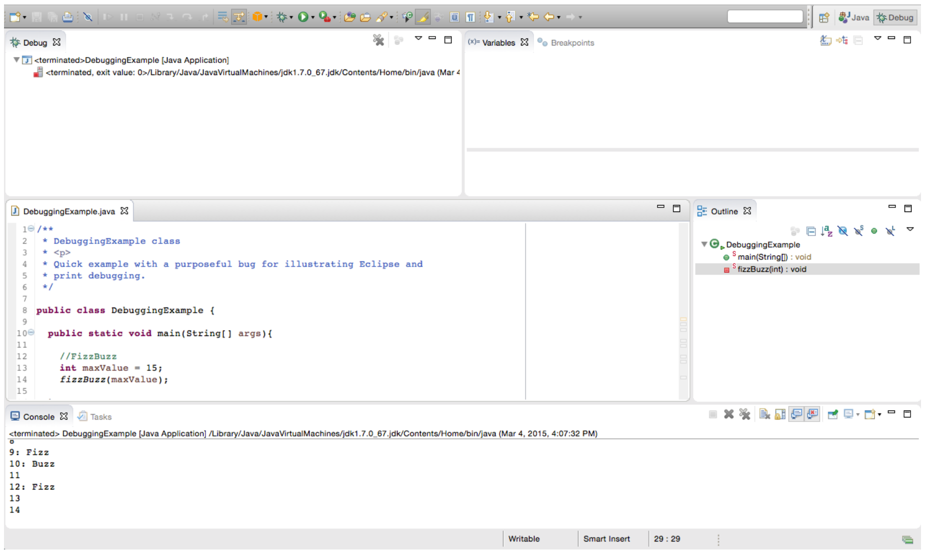


Figure 1: Debug Perspective Button

- This will switch you to the Debug Perspective where you should see a debug view panel, variables and breakpoints view panels, your program code, and a console view panel.



```java
/**
 * DebuggingExample class
 * <p>
 * Quick example with a purposeful bug for illustrating Eclipse and
 * print debugging.
 */

public class DebuggingExample {

  public static void main(String[] args){

    //FizzBuzz
    int maxValue = 15;
    fizzBuzz(maxValue);

```

```
<terminated> DebuggingExample [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:07:32 PM)
9: Fizz
10: Buzz
11
12: Fizz
13
14
```

Figure 2: Debug Perspective

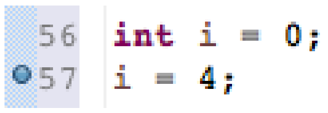
- To switch back to normal Java execution, simply click the Java button in the top right (next to the Debug button).

Using the Eclipse Debugger:

General Breakpoint Info:

When you specify a breakpoint, you specify to the debugger a point in your program at which execution will pause when that point is reached by your machine. It's important to note that when you set a breakpoint on a line, the program will pause execution before that line is actually executed (not after).

So if we had two lines of Java code in a program we were debugging:



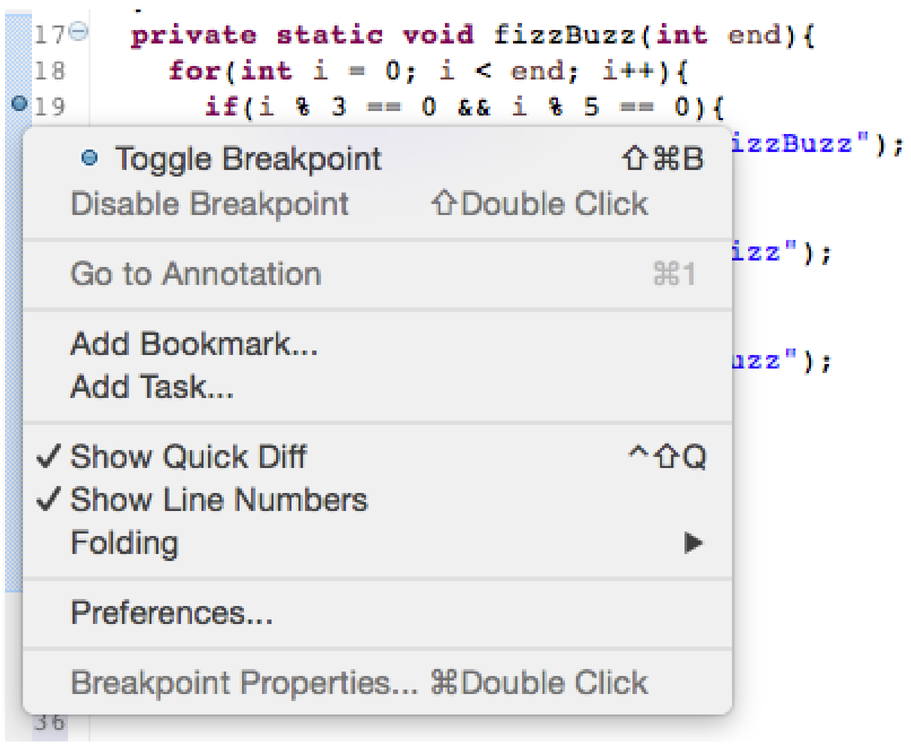
```java
int i = 0;
i = 4;
```

Figure 3: Debugger Breakpoint Line

And we set the breakpoint on the 'i = 4;' line (indicated by the blue dot), then the program will pause right after 'int i = 0;' has been executed and the value of i will be 0, not 4.

Setting Breakpoints:

To set breakpoints, you can right click on the left rail next to the line of code at which you want to set a breakpoint. Note that you can do this in either the Java or Debug perspectives.



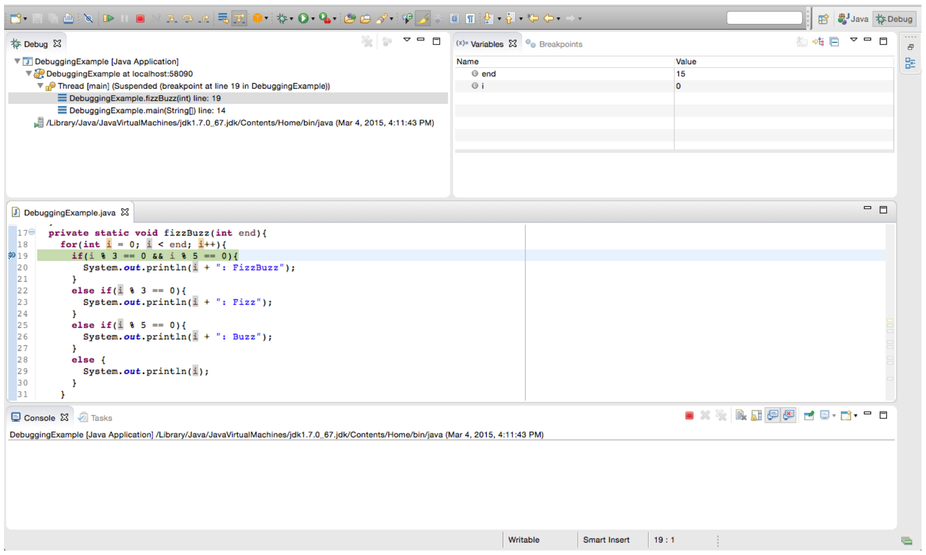
```
private static void fizzBuzz(int end){
  for(int i = 0; i < end; i++){
    if(i % 3 == 0 && i % 5 == 0){
                                   izzBuzz");

                                   izz");

                                   uzz");
```

Figure 4: Set Breakpoint

Running/Pausing Execution in Debug Pespective:

Then, to run your program with breakpoints, you have to be in the Debug perspective and click on the bug icon. Your program will execute and run up until the first instance of a breakpoint and then pause.

Bear in mind that clicking on the bug multiple times will run your program multiple times, potentially without the other instances of program execution having finished. Each program execution will show up as a separate execution of a Java Application in the debug window. You can check the program variables, state, etc. of each instance of program execution by clicking on the appropriate program runtime instance in the debug view panel.



```
Debug
DebuggingExample [Java Application]
  DebuggingExample at localhost:58090
    Thread [main] (Suspended (breakpoint at line 19 in DebuggingExample))
      DebuggingExample.fizzBuzz(int) line: 19
      DebuggingExample.main(String[]) line: 14
  /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:11:43 PM)
```

Variables

Name	Value
end	15
i	0

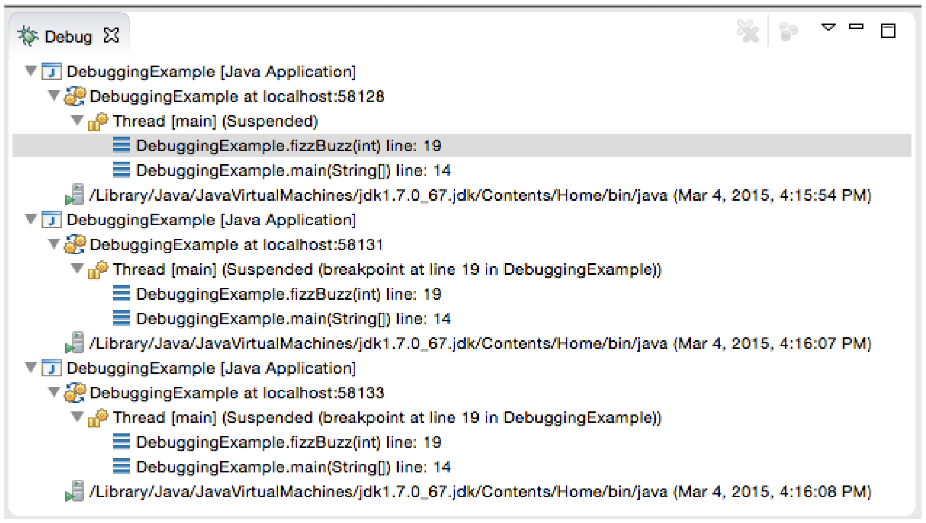
DebuggingExample.java

```java
  private static void fizzBuzz(int end){
    for(int i = 0; i < end; i++){
      if(i % 3 == 0 && i % 5 == 0){
        System.out.println(i + ": FizzBuzz");
      }
      else if(i % 3 == 0){
        System.out.println(i + ": Fizz");
      }
      else if(i % 5 == 0){
        System.out.println(i + ": Buzz");
      }
      else {
        System.out.println(i);
      }
    }
```

```
Console
DebuggingExample [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:11:43 PM)
```

Writable | Smart Insert | 19 : 1

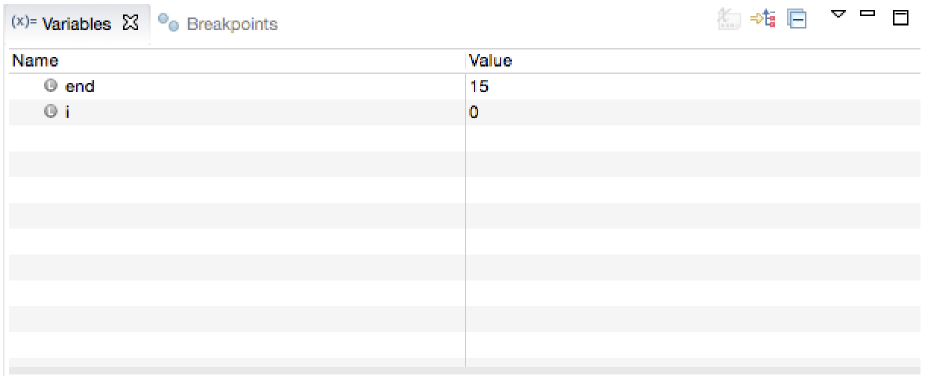
Figure 5: Debug Run



```
Debug
DebuggingExample [Java Application]
  DebuggingExample at localhost:58128
    Thread [main] (Suspended)
      DebuggingExample.fizzBuzz(int) line: 19
      DebuggingExample.main(String[]) line: 14
  /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:15:54 PM)
DebuggingExample [Java Application]
  DebuggingExample at localhost:58131
    Thread [main] (Suspended (breakpoint at line 19 in DebuggingExample))
      DebuggingExample.fizzBuzz(int) line: 19
      DebuggingExample.main(String[]) line: 14
  /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:16:07 PM)
DebuggingExample [Java Application]
  DebuggingExample at localhost:58133
    Thread [main] (Suspended (breakpoint at line 19 in DebuggingExample))
      DebuggingExample.fizzBuzz(int) line: 19
      DebuggingExample.main(String[]) line: 14
  /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:16:08 PM)
```

Figure 6: Multiple Runtimes

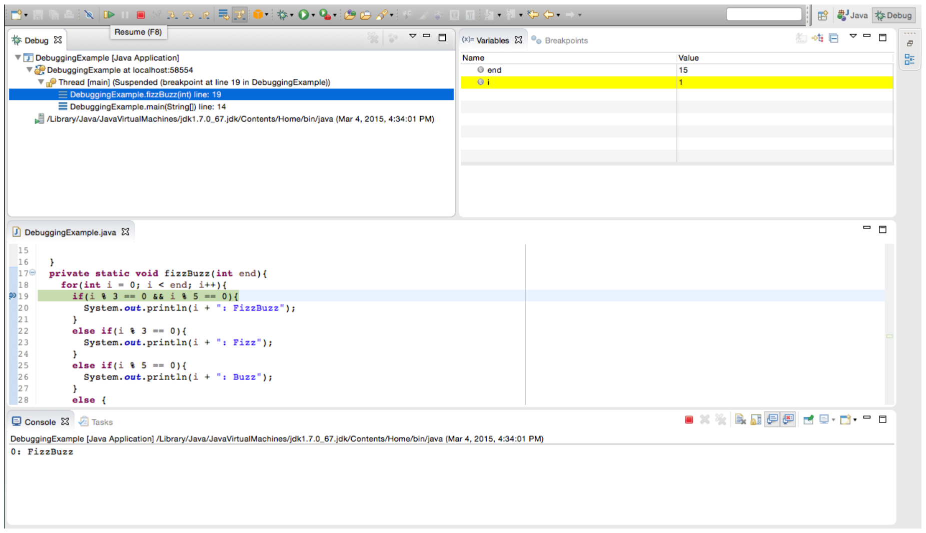
While a program is paused at a breakpoint, you can view the values of the variables that have been instantiated and are available in the current scope (hence why maxValue is not accessible) in the Variables window.



Name	Value
end	15
i	0

Figure 7: Breakpoint Variables

To continue executing a program after it has been paused at a breakpoint, press the resume button. Your program will continue executing until either the next instance of a breakpoint or it has finished executing.



```
DebuggingExample [Java Application]
  DebuggingExample at localhost:58554
    Thread [main] (Suspended (breakpoint at line 19 in DebuggingExample))
      DebuggingExample.fizzBuzz(int) line: 19
      DebuggingExample.main(String[]) line: 14
  /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:34:01 PM)
```

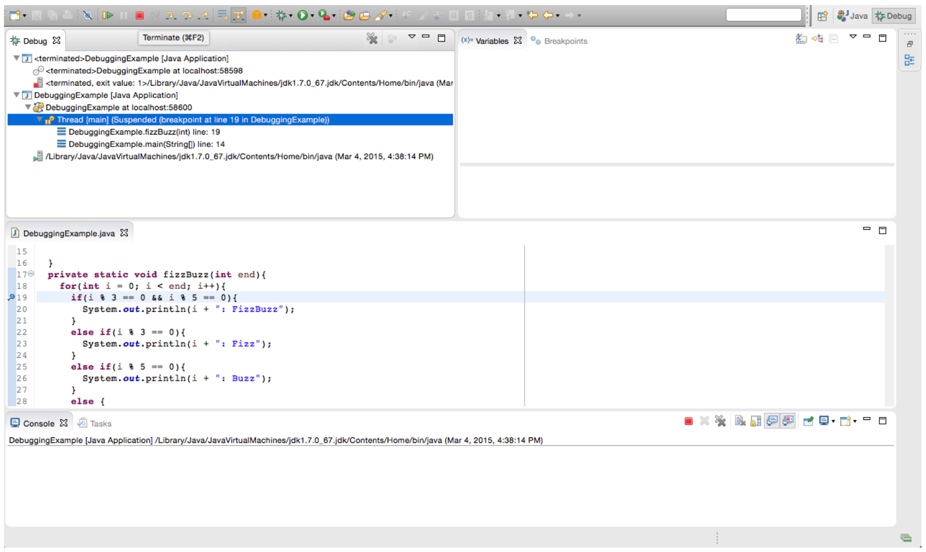
Name	Value
end	15
i	1

```java

  }
  private static void fizzBuzz(int end){
    for(int i = 0; i < end; i++){
      if(i % 3 == 0 && i % 5 == 0){
        System.out.println(i + ": FizzBuzz");
      }
      else if(i % 3 == 0){
        System.out.println(i + ": Fizz");
      }
      else if(i % 5 == 0){
        System.out.println(i + ": Buzz");
      }
      else {
```

```
DebuggingExample [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:34:01 PM)
0: FizzBuzz
```

Figure 8: Debug Resume

To stop your program prematurely before it has stopped executing, select the instance of your program execution you want to stop in the Debug window and click on the red square (Terminate).



```
<terminated>DebuggingExample [Java Application]
    <terminated>DebuggingExample at localhost:58598
    <terminated, exit value: 1>/Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar
DebuggingExample [Java Application]
    DebuggingExample at localhost:58600
        Thread [main] (Suspended (breakpoint at line 19 in DebuggingExample))
            DebuggingExample.fizzBuzz(int) line: 19
            DebuggingExample.main(String[]) line: 14
    /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:38:14 PM)
```

```java

}
private static void fizzBuzz(int end){
  for(int i = 0; i < end; i++){
    if(i % 3 == 0 && i % 5 == 0){
      System.out.println(i + ": FizzBuzz");
    }
    else if(i % 3 == 0){
      System.out.println(i + ": Fizz");
    }
    else if(i % 5 == 0){
      System.out.println(i + ": Buzz");
    }
    else {
```

```
DebuggingExample [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Mar 4, 2015, 4:38:14 PM)
```

Figure 9: Debug Stop

Breakpoint Properties:

To access the properties of a breakpoint, right click on a breakpoint and select Breakpoint Properties.

Can't Change:

- Type = the class in which the breakpoint resides
- Line Number = the line number of the line of code marked as a breakpoint
- Member = the method that the breakpoint is set in

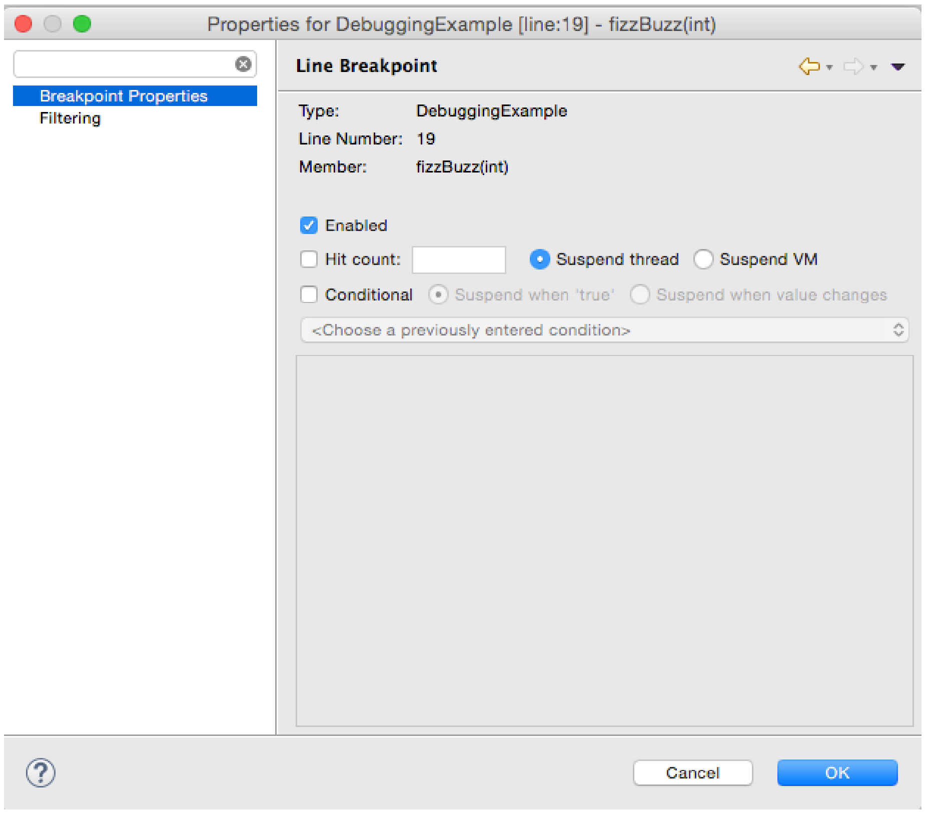
Can Change:

- Enabled = whether the breakpoint is currently enabled and will pause the program execution when reached
- Hit Count = how many times the breakpoint needs to be reached for program execution to pause
- Conditional = specifying under what conditions (boolean condition) the breakpoint will be active

Advanced Debugging/Breakpoints:

For more advanced Eclipse debugging techniques, feel free to look at various online resources. With more advanced techniques and different types/usages of breakpoints, you can:

- set breakpoints that only become active only when certain conditions become true



Type:	DebuggingExample
Line Number:	19
Member:	fizzBuzz(int)

[✓] Enabled
[ ] Hit count: [✓ Suspend thread] Suspend VM
[ ] Conditional [✓ Suspend when 'true'] Suspend when value changes
<Choose a previously entered condition>

Figure 10: Breakpoint Properties

- set breakpoints that watch a specific variable for read or write (assigning a value) attempts
- set breakpoints for program exceptions or prior to crashes
- set breakpoints for when a specific method is entered or exited
- set breakpoints for when a specific class is loaded by any other class

---

Other Notes:

- Debugging is a skill in its own right and something that you get better at over time.
- As a skill/practice, debugging has great returns on time. Expending extra time/effort to learn more advanced debugging techniques or tools saves a lot of time later when encountering and fixing bugs.
- However, for debugging to fulfill this, you have to make sure that you understand the root cause of your bugs and also how the fix specifically works and fixes the bug. This is especially important because if/when you encounter a similar or even the same bug in another or the same program, you understand why it's happening and know how to quickly fix it.
- Thurs, Debugging still requires an understanding of your code and how your programs work. Debugging is primarily expected vs. actual output/results. However, to really get a sense of expectations for certain values or at certain points of your program, you need an understanding of the program and the code.

More Resources/Documentation:

- Vogella Eclipse Debugging Tutorial
- Great Debugging Tips
- SourceForge - Eclipse Tutorial (2008)
- Eclipse Debugging Tutorial
- Wikipedia Page: Debugging