Debugging in the NetBeans IDE

The Concepts:

Debugging

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

A "bug" is a software defect, the nature of which may or may not have been identified. A "bug" is present when the software behaves in a way that is not intended by the developer. Bugs can be present in all phases of the software development life cycle. A defect can be introduced from the initial determination of software requirements ("what is to be built"), during software design ("how it should be built"), right through to its final implementation ("how it is built"). In software development, we often know that our software has bugs, but we don't know what they are.

The term "debugging" is used to cover a wide range of tasks but generally applies to the process of (1) detecting, (2) locating, and (3) repairing "bugs." These are three separate tasks; to detect the presence of bugs is not the same thing as locating which section of design or code is responsible for them, nor the same as the process of repairing them.

Debuggers

Debuggers let us follow our program execution step by step, examine data (including the call stack), and set “break points”. Many debuggers allow us to dynamically change data as well. NetBeans (for example) includes a basic debugger with these features.

We can follow program execution by stepping though every line that is executed if so desired, or by choosing not to follow function calls. If we do not follow a function call we just treat that function as a black box, and the debugger regains control of the program when the function returns. Since most (difficult) bugs occur after a non-trivial number of lines of code have executed we often choose not to follow every function call because it would take us too long.

We often use “break points” to help us get to the point where we suspect something of interest occurs. A “break point” is a place in your code where you would like to stop the process of normal execution so that you can manually step through (execute) your code line by line and examine the values of variables and data attributes. We use the debugger to set a break point at a particular line of code, and then at that point run the program line by line in the debugger. We can also examine the call stack. This tells us which function called function, and which function called it, and so on.

Debuggers (cont.):

Debuggers are powerful tools that can make the job of finding the cause of a problem much easier, provided we use them carefully. We always need to think about how to trigger the problem, and how best to use the tool to help resolve it.

Trace debugging (walking step-by-step through the execution of your code) allows you to see in a visual way, how your code is executing, and to watch the state of variables, the stack, and threads. Tools and IDE’s that allow trace debugging are the preferred way to locate bugs in Java software. Not only do they make it simpler for the developer, they give you a better understanding of how your code works. This method of debugging is also a more efficient way of locating errors. In these days of tight deadlines, and cutting corners, anything that can make debugging time more productive is good for you.

©2012 Christopher Hield

Debugging

Fundamental Principle of Debugging: Confirmation

Finding a bug is the process of confirming the things that you believe to be true about the way your program is running – until you find one that is not true.

Examples of things to confirm:

- Your belief that a variable “x” equals “12” at a certain time.
- Your belief that in a certain “if/then/else” statement, the “else” gets executed.
- Your belief that a call to some function returns the value you were expecting

“Buggy” Code Example:

Insertion Sort

The following code is a simple example that attempts to calculate the average value of an array in integers, as well as determine the maximum value in the array.

Sample Code:

```
public class DebugExample {

    // Input numbers
    private static int inputs[] = {12, 5, 9, 3, 2, 5};

    public static void main(String args[])
    {
        int average = calculateAverage();
        System.out.println("Average: " + average);

        int maxNum = calculateMax();
        System.out.println("Maximum: " + maxNum);
    }


    private static int calculateAverage()
    {
        int avg = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            avg += inputs[0];
        }
        return avg / inputs.length;
    }

    private static int calculateMax()
    {
        int max = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            if (inputs[i] == max)
            {
                max = inputs[i];
            }
        }
        return max;
    }
}
```

“Buggy” Code Example:

Why Buggy?

If performed manually using the input numbers found in the code, one can see that the average of the input numbers is:

12 + 5 + 9 + 3 + 2 + 5 = 36

Average = 36/6 nums = **6**

One can also see that the **maximum** value in the array is: 12

However, when executed, the sample program generates the following results:

```
Average: 12
Maximum: 0
```

Neither the average value nor the maximum value is correct. Obviously, something is not working properly in the program.

Without knowing exactly what was happening at each step of my program’s execution, it is difficult to determine what went wrong. I’d like to walk through my code execution so that I can review the path of execution and the values of the variables & attributes as that execution continues.

An interactive debugger is needed.

Fortunately, NetBeans (like most IDE’s) comes with a full-featured built-in interactive debugger!

Debugging Example:

The NetBeans IDE Debugger

Before executing the program in the debugger, we have to decide where we want to stop normal execution and begin line by line execution in the debugger. If we did not do this, the program would execute just as before with no debugger involvement.

To specify where you want to stop normal execution, you need to specify a *break point*. To do this, simply left-click on the line number where you would like to stop normal execution. *(Clicking that line again un-selects the line).*

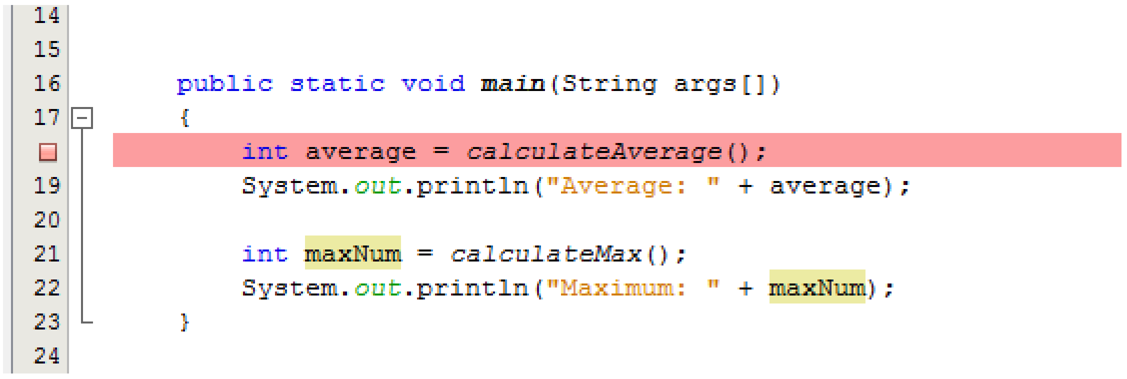
You may enter any number of breakpoints in your code. *(To show line numbers in NetBeans - if they are not already displayed - select menu option "View" -> "Show Line Numbers").*

In our code example, we'll stop execution at the first line of our "main" – the call to "calculateAverage()":

A red indicator will be displayed over the line number you selected indicating where the break point is.

The associated line of code will be highlighted in light red.

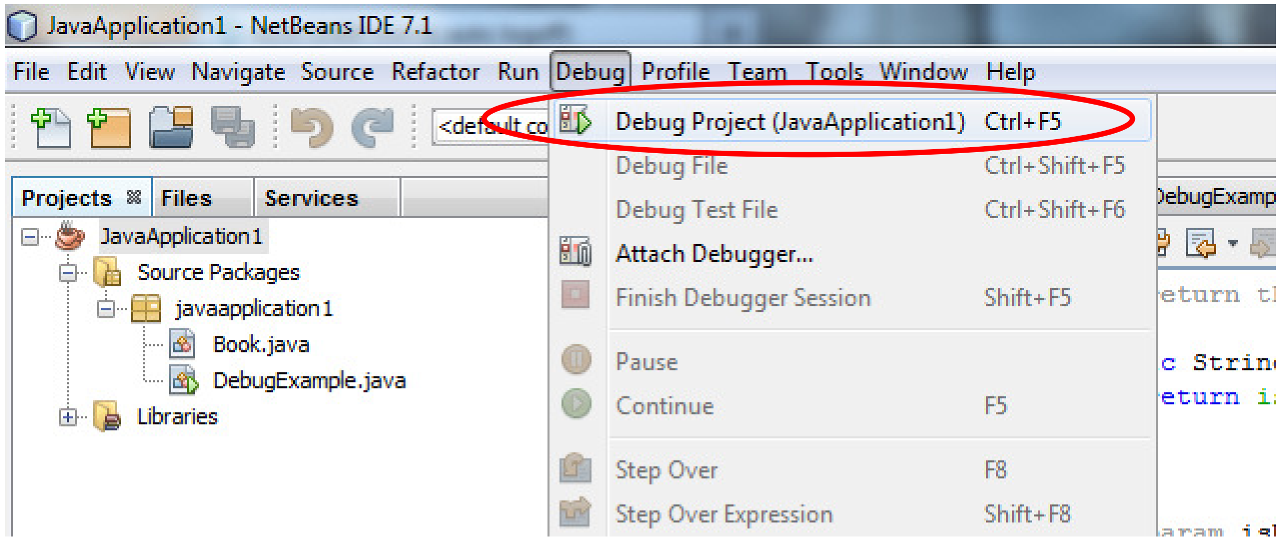
Upon debugger execution, normal execution will stop at this point and debugger-execution will commence allowing you to examine your program.

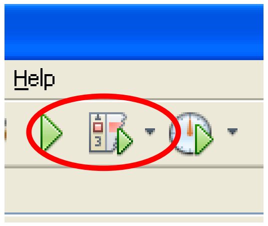


```
14
15
16      public static void main(String args[])
17      {
            int average = calculateAverage();
19          System.out.println("Average:  " + average);
20
21          int maxNum = calculateMax();
22          System.out.println("Maximum:  " + maxNum);
23      }
24
```

Debugging Code Example:

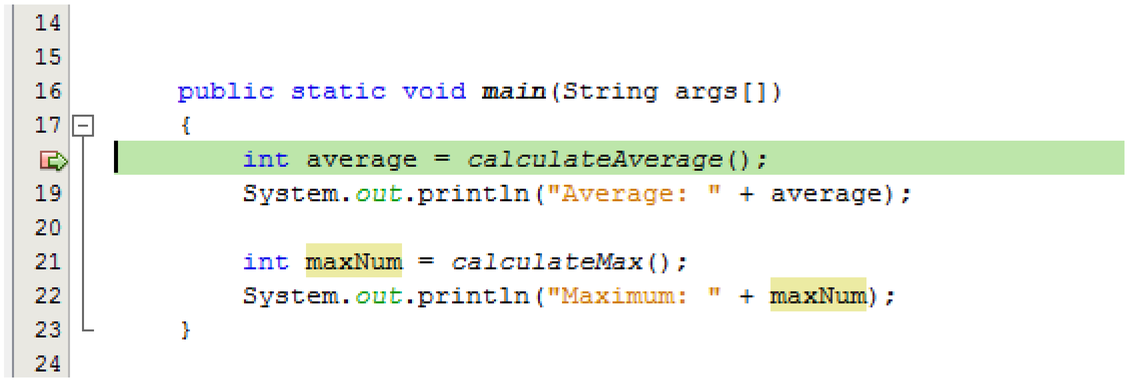
To execute your program in the NetBeans debugger, it must compile and be ready to run. Rather than choosing to run the program as usual, you should select “Debug Project” from the “Debug” menu (or click the “Debug Project” toolbar icon)


JavaApplication1 - NetBeans IDE 7.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Debug Project (JavaApplication1) Ctrl+F5
Debug File Ctrl+Shift+F5
Debug Test File Ctrl+Shift+F6
Attach Debugger...
Finish Debugger Session Shift+F5
Pause
Continue F5
Step Over F8
Step Over Expression Shift+F8
Projects
Files
Services
JavaApplication1
Source Packages
javaapplication1
Book.java
DebugExample.java
Libraries



Debugging Code Example:

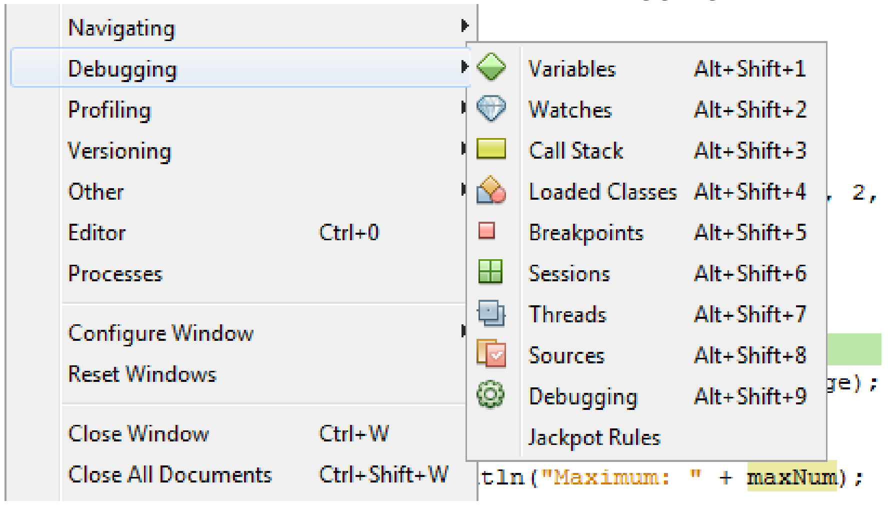
Once you select (or click) “Debug Project”, the program will execute as it normally does up to your break point. Once there, it will stop and wait for your instructions. The debugger will show you the line it has stopped on (A small light-green arrow over the line number with the break point indicates this. Additionally, the line of code is highlighted in a light-green color).



```

    public static void main(String args[])
    {
        int average = calculateAverage();
        System.out.println("Average: " + average);

        int maxNum = calculateMax();
        System.out.println("Maximum: " + maxNum);
    }

```

Debugging Code Example:

NetBeans offers a set of “investigation”- related debugging tools that each open in their own window/frame. These tools are accessed via the “Window” menu-bar, at “Window” -> “Debugging”, then selecting one of the following:



```
tln("Maximum: " + maxNum);
```

Name	**Description**
Variables	Lists the variables that are available within the scope of the current method.
Watches	Lists all variables and expressions that you elected to watch while debugging your application.
Call Stack	Lists the sequence of calls made during execution of the current thread.
Loaded Classes	Displays the hierarchy of all classes that have been loaded by the process being debugged.
Breakpoints	Lists the breakpoints in the current project.
Sessions	Lists the debugging sessions currently running in the IDE.
Threads	Lists the thread groups in the current session.
Sources	Lists the source directories on your project classpath. You can set whether to step into or step over classes by deselecting their source folders here. The IDE automatically steps over JDK classes; if you want to step into them, select the JDK sources in this window.

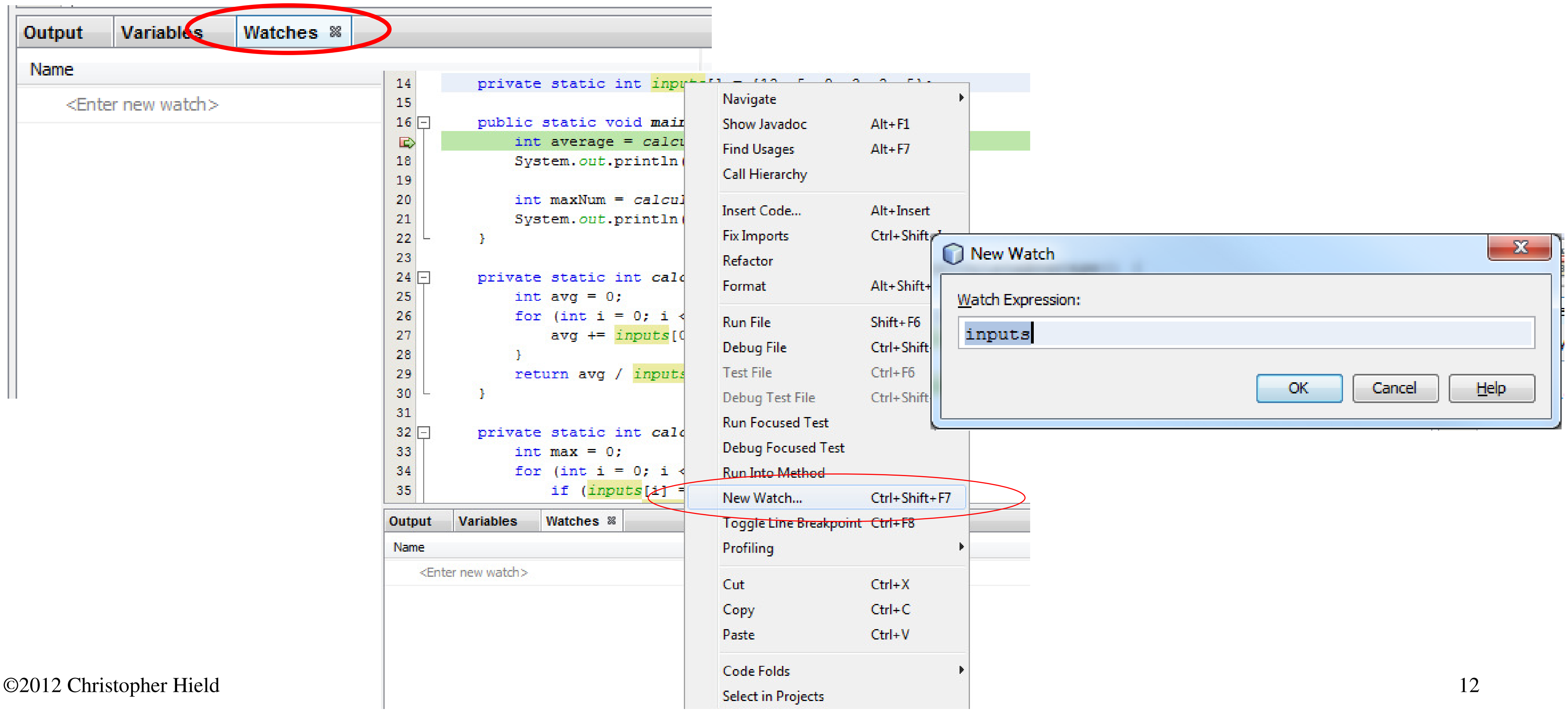
Object-Oriented Development

Debugging in NetBeans

Debugging Code Example:

Now that we've stopped normal execution at the call to "calculateAverage ()", lets examine the program's "inputs" array. Select menu option "Window -> Debugging -> Watches" to display the "Watches" in the lower-right frame.

To create a "watch", right-click the mouse on the "inputs" variable anywhere in the code and select "New Watch…" from the displayed menu, then click "Ok" in the "New Watch" pop-up window.



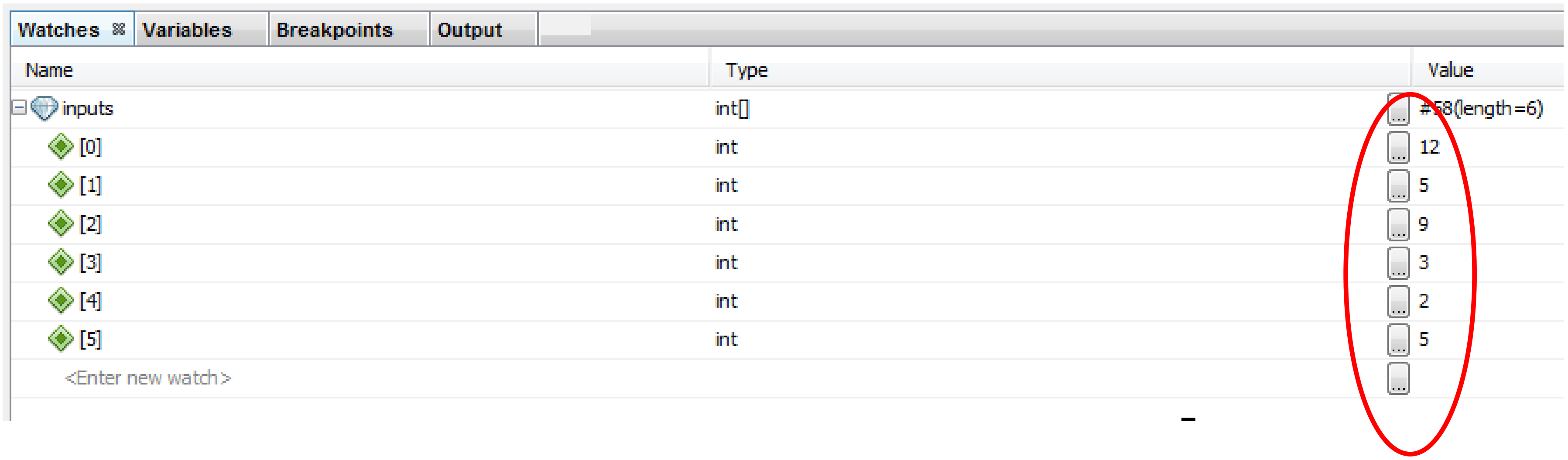
```
    private static int inpu

    public static void mair
        int average = calcu
        System.out.println(

        int maxNum = calcul
        System.out.println(
    }

    private static int calc
        int avg = 0;
        for (int i = 0; i <
            avg += inputs[0
        }
        return avg / inputs
    }

    private static int calc
        int max = 0;
        for (int i = 0; i <
            if (inputs[i] =
```

Watch Expression: inputs

Debugging Code Example:

This will create a new "watch" in the "Watches" tab (in the lower portion of the NetBeans IDE window), showing the value(s) of the selected variable or attribute. If the "Watches" tab is already open, the selected variable or attribute will be added to the list of current "watches". *If the "Watches" tab does not open automatically, select option "Windows" -> "Debugging" -> "Watches".*

Below, the values of each element of the "inputs" array are displayed. Initially it will be displayed as a "closed" item, but if you click on the small "**+**" icon next to the name, the watch opens showing you the variable's content



Name	Type	Value
inputs	int[]	#58(length=6)
[0]	int	12
[1]	int	5
[2]	int	9
[3]	int	3
[4]	int	2
[5]	int	5
<Enter new watch>		

These "input" data values look as they should (values 12, 5, 9, 3, 2 & 5), confirming that the start-state of our program is correct.

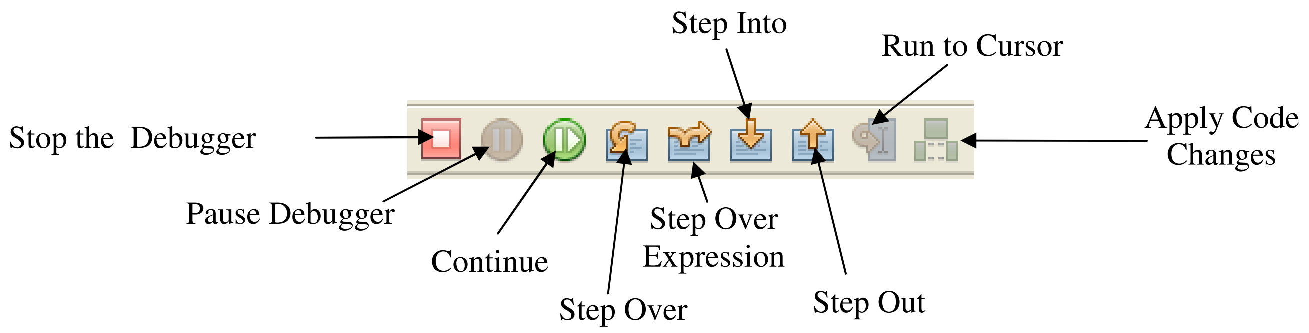
©2012 Christopher Hield

Debugging Code Example:

Once we have confirmed that the start-state of our program is correct (the “inputs” array contains what we expect), we can start line-by-line debugger execution which will allow us to follow the program’s execution flow and to examine the values of the variables, attributes and the call stack at each step.

In addition to the NetBeans debugger’s investigation tools (which we saw earlier, allowing us to examine our running program in various ways – “watches”, etc.), NetBeans’ offers a set of execution-related debugging tools. These tools can be found near the top of the NetBeans IDE window in the “Debug” toolbar when the debugger is active. (The “Debug” toolbar can be displayed at any time by selecting menu-bar “View” -> “Toolbars” -> “Debug”).

The icons present in the “Debug” toolbar are as follows, and are described below:



Debugging Code Example:

The Debug Toolbar Commands

Command	**Debugging Function**
Stop the Debugger	Ends the current debugging session.
Pause Debugger	Pauses but does not end the current debugging session
Continue	Continue debugger execution after a Pause or a Break Point
Step Over	Executes one source line. If the source line contains a call, executes the entire routine without stepping through the individual instructions.
Step Over Expression	You can use the Step Over Expression command to achieve a more fine-grained stepping than other debugging steps. Step Over Expression enables you to proceed through each method call in an expression and view the input parameters and resulting output values of each method call. You can invoke the Step Over Expression command just as you would any other step commands. If there are no further method calls, Step Over Expression behaves like the Step Over command.
Step Into	Executes one source line. If the source line contains a call, the IDE stops just before executing the first statement of the routine. You can also start a debugging session with the Step Into command. Program execution stops on the first line after the main routine before any changes have been made to the state of the program.
Step Out	Executes one source line. If the source line is part of a routine, executes the remaining lines of the routine and returns control to the caller of the routine.
Run to Cursor	Runs the current project to the cursor's location in the file and pause program execution.

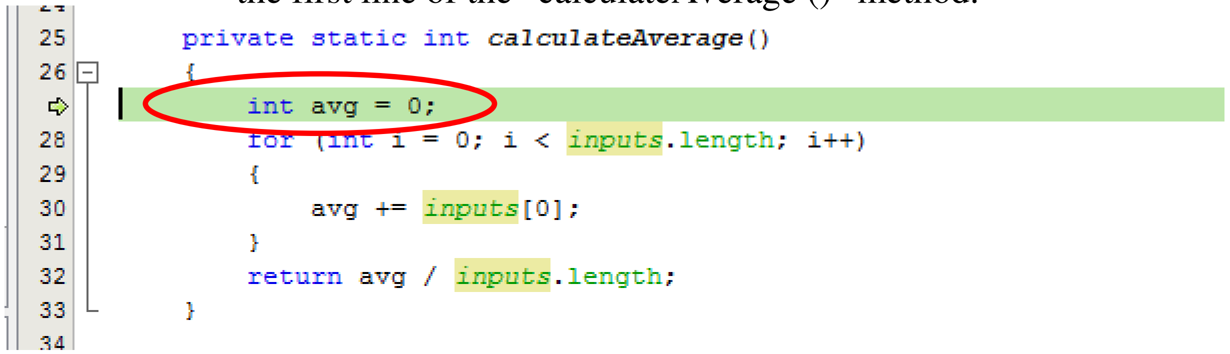
Debugging Code Example:

The Debug Toolbar Commands (cont.)

Apply Code Changes	If you find a problem while debugging, you can use the *Apply Code Changes* command to fix your source and then continue debugging with the changed code without restarting your program. It is not possible to use the Apply Code Changes command to do the following: • Change a modifier of a field, a method, or a class • Add or remove methods or fields • Change the hierarchy of classes • Change classes that have not been loaded into the virtual machine To fix your code: 1) From the main menu, choose Run > Apply Code Changes to recompile and begin repairing your source code. • If there are errors during compilation, nothing is changed in your program. Edit your source code as needed, then execute the Apply Code Changes command again. • If there are no errors, the resulting object code is swapped into the currently executing program. However, all calls on the call stack continue running the unfixed code. To use the modified code, you must pop from the call stack any calls that contain modified code. When the calls are reentered, they use the modified code. Note: If you modify the currently running method, the IDE displays an alert box. If you click the Pop Call button, the most recent call is removed from the current call stack. If you click the Leave Call button, the program executes with the original version of the code. If there is only one call in the stack, you cannot pop the call and continue. 2) Continue the program to verify that the fixed version of your code works correctly. *The Apply Code Changes command does not automatically rebuild JAR files, executable files, or similar files. You must rebuild these files if you want to debug them in a new session.*

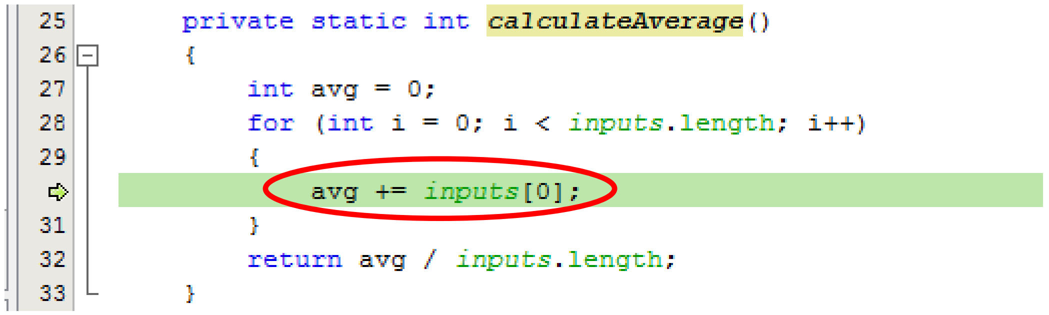
Debugging Code Example:

Since we would like to execute the "calculateAverage ()" method in the debugger, click the **"Step Into"** button to step into that method. (If we clicked "Step Over", then the method would be executed normally, not in the debugger). The debugger will change its scope to the "calculateAverage ()" method and will set the next line of code to be executed to be the first line of the "calculateAverage ()" method:



```
    private static int calculateAverage()
    {
        int avg = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            avg += inputs[0];
        }
        return avg / inputs.length;
    }
```

To step through the code in this method, we next click either the "Step Into" or "Step Over" button. (Since there are no method calls on the current line, you can use "Step Into" and "Step Over" interchangeably).
After a couple of "steps", the "for" loop will be entered. The next line to execute will add a value from the int array "inputs" to the "avg" local variable.



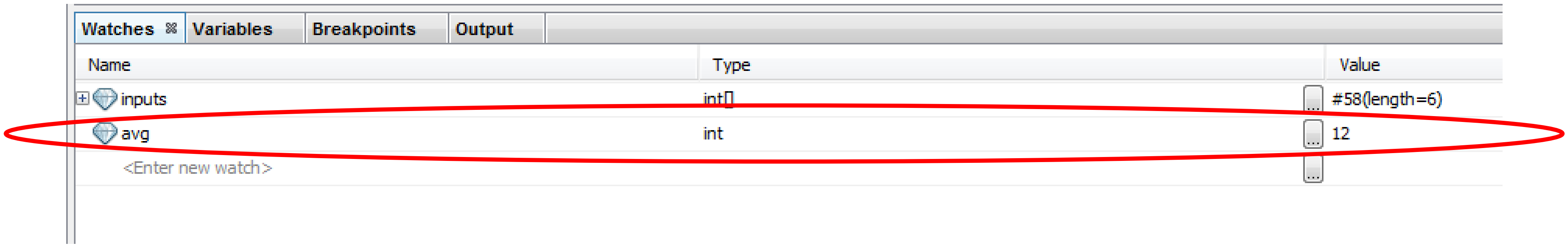
```
    private static int calculateAverage()
    {
        int avg = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            avg += inputs[0];
        }
        return avg / inputs.length;
    }
```

Debugging Code Example:

Clicking on the "Step Into" or "Step Over" button will cause the next line of code to execute - where we add the next value from the int array "inputs" to the "avg" local variable. Since this is the first loop iteration, the value added should be 12.

Doing this will execute that line of code, and bring execution back up to the "for" loop, where it will check the loop "exit" condition (i < inputs.length). At this point we can check what was added to the "avg" variable *by adding another watch*, this time to watch the "avg" variable. This is done just like we did to watch the "inputs" variable.

As you can see, the "avg" variable's value is 12, the first element in the "inputs" array. So far things are looking good.

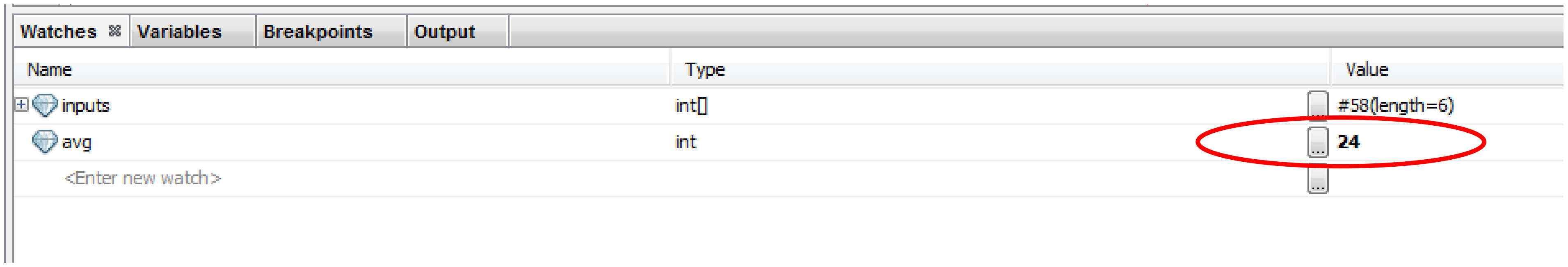


Name	Type	Value
inputs	int[]	#58(length=6)
avg	int	12
<Enter new watch>		

Debugging Code Example:

Clicking on the "Step Into" or "Step Over" button will again execute the line of code that will add the next value from the int array "inputs" (5) to the "avg" local variable.

Doing so will execute that line of code, and bring execution back up to the "for" loop, where it will check the look "exit" condition (i < inputs.length). At this point we can again check the value of the "avg" variable to insure its value is now 17 (12 + 5).



Name	Type	Value
inputs	int[]	#58(length=6)
avg	int	24
<Enter new watch>		

However, as you can see above, "avg" is not 17, its value is 24! Something has gone wrong.

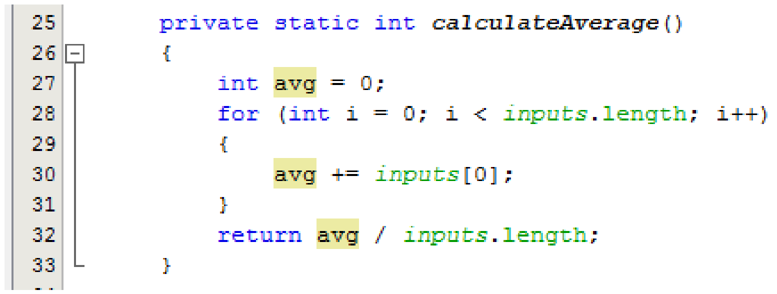
Rather than add the *next* element from the array (element [**1**] = 5) to "avg", it looks like we've re-added the first element from the array (element [**0**] = 12) to "avg" again.

If we proceed to execute the next "for" loop iteration, we see that rather than add the next element from the array (element [**2**] = 9) to "avg", it looks like we've *again* re-added the first element from the array (element [**0**] = 12) to "avg". This looks like it might be a bug – perhaps the bug that is causing our average calculation to produce incorrect values!

©2012 Christopher Hield

Debugging Code Example:

Examine the “for” loop in question more closely – do you see something wrong with the code that is trying to add each int array element to the “avg” variable?



```
25        private static int calculateAverage()
26        {
27            int avg = 0;
28            for (int i = 0; i < inputs.length; i++)
29            {
30                avg += inputs[0];
31            }
32            return avg / inputs.length;
33        }
```

From what we’ve seen in this debugging session, it seems that the “for” loop iterations keeps re-adding the *first* element in the “inputs” array (12) to the “avg” variable, rather than adding *each* individual element from the “inputs” array to the “avg” variable. If you look at the code above, you see on line 30:

```
for (int i = 0; i < inputs.length; i++)
{
    avg += inputs[0];
}
```

This line is *supposed* to be adding *each* element from the “inputs” array to the int “avg” variable, using the “for” loop counter *“i”* as the index into the “inputs” array. However, instead of using *“i”* as the index into the “inputs” array, you can see there is a hardcoded *“0”* being used as the index.
As a result each “for” loop iteration will add “inputs” element [**0**] (i.e., 12) to the “avg” variable.

This is exactly what we saw happening when we executed this code in the debugger. We have found our code bug!

Debugging Code Example:

Now that we have found a problem in the code – we can fix it to use *"i"* as the index into the "inputs" array, and now a hard-coded *"0"*:

```
for (int i = 0; i < inputs.length; i++)
{
    avg += inputs[i];
}
```

Once we fix this code, we can try re-running the program to see if we have completely fixed our problem. This time, the re-run generates the following output:

```
Average: 6
Maximum: 0
```

This is better than before; the expected average value of "6" is now properly calculated and displayed.

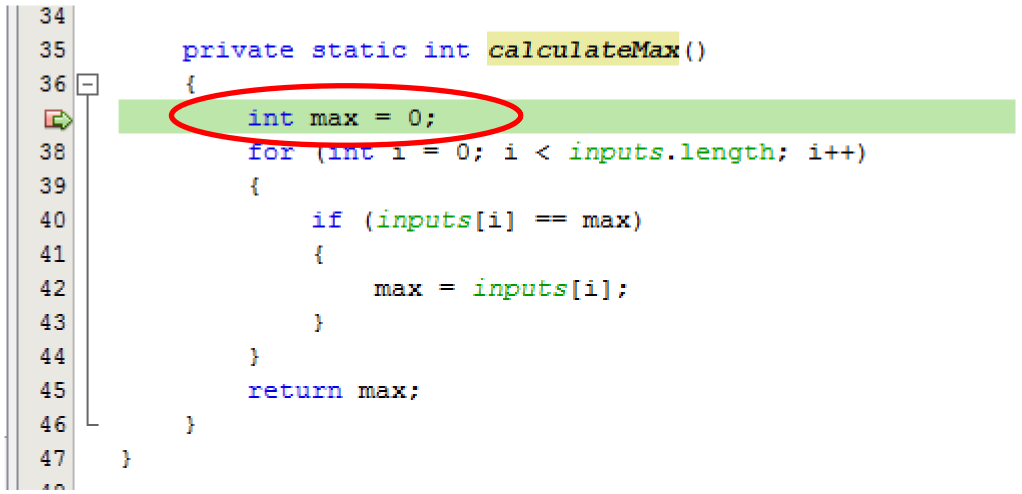
However, the "Maximum" value (which should be 12) is still showing as 0, so there must be *another* problem here somewhere. Often, fixing one bug will uncover another bug.

To find what else is wrong, we will first remove the original break point *(clicking a line number that already contains a break point will un-select/remove the breakpoint).* Then we will add a new breakpoint on the first line of the "calculateMax" method.

Remember, to add a new break point, simply left-click on the line number where you would like to stop normal execution. *(Clicking that line again un-selects the line).* Once again, select (or click) "Debug Project", and program will execute as it normally does up to your break point.

Debugging Code Example:

Once there, it will again stop and wait for your instructions. The debugger will show you the line it has stopped on (A small light-green arrow over the line number with the break point indicates this. Additionally, the line of code is highlighted in a light-green color).



```
    private static int calculateMax()
    {
        int max = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            if (inputs[i] == max)
            {
                max = inputs[i];
            }
        }
        return max;
    }
}
```

This method starts by creating a local variable called “max” that will be used to hold the “current” maximum value. Then in the “for” loop, the method will check each value of the “inputs” array to see if it is greater than the current “max” value. If not, then it’s *less than* our current “max” value, so we don’t do anything. If it is, then we’ve found a new current “max” value and we will store that value as the new current “max” value.

Once the “for” loop ends, we’ve examined all the values in “inputs”, and the “max” variable will be holding the largest value we found in the array – so we return that value.

Object-Oriented Development

Debugging in NetBeans

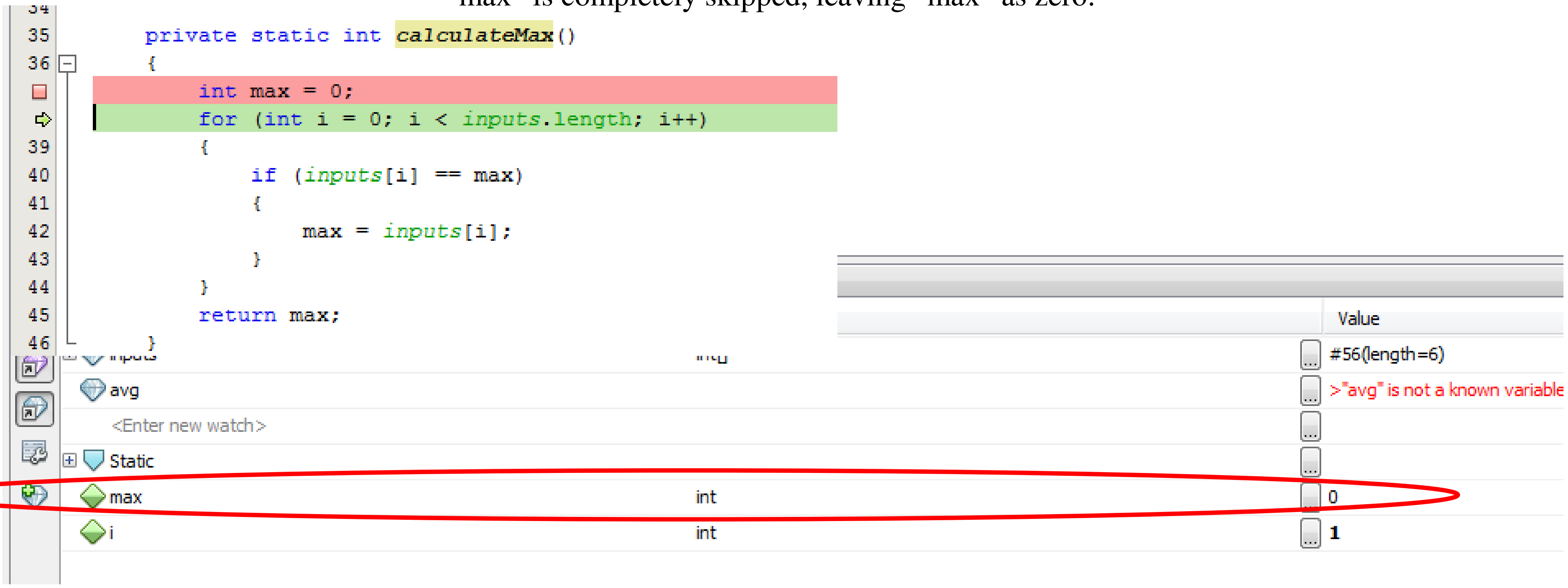
Debugging Code Example:

Since "max" starts at zero, we would expect the first element in "inputs" (element [0] = 12) to be considered the "max".

We will first add a "watch" for the "max" variable so we can track its value, and then we will "Step Into" or "Step Over" each line in the "for" loop.

Once in the "for" loop, the first thing to execute is the "if" check, that checks if the next value (12) is greater than the current "max" – which is initially zero.

However when I do this, I see the "if" statement is not considered "true", and the "if" block that does the assignment to "max" is completely skipped, leaving "max" as zero:

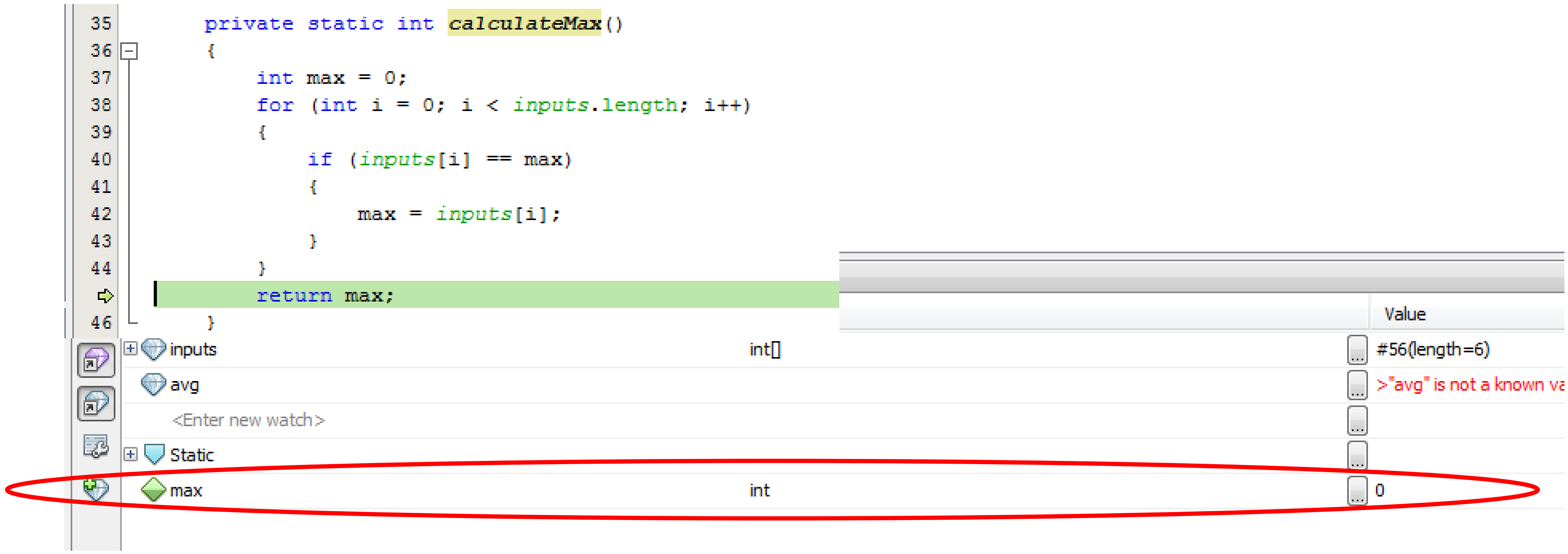


```
    private static int calculateMax()
    {
        int max = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            if (inputs[i] == max)
            {
                max = inputs[i];
            }
        }
        return max;
    }
```

		Value
inputs	int[]	#56(length=6)
avg		>"avg" is not a known variable
<Enter new watch>		
Static		
max	int	0
i	int	1

Debugging Code Example:

If we proceed to execute the next "for" loop iteration, we see that again the "if" block is completely skipped for the next element from the array (element [**1**] = 5) so again "max" remains zero. Continuing on with the remaining "for" loop iterations, we see that the "if" block is skipped in *every* "for" loop iteration, so "max" is never set. *That* is why it ends up with a value of zero when we run the program.



```
    private static int calculateMax()
    {
        int max = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            if (inputs[i] == max)
            {
                max = inputs[i];
            }
        }
        return max;
    }
```

```
                                                   Value
inputs                     int[]                   #56(length=6)
avg                                                >"avg" is not a known va
<Enter new watch>
Static
max                        int                     0
```

Debugging Code Example:

From what we've seen in your debugging session, the code never considered any input value as greater than our starting "max" value of zero. If you look at the code involved, you see on line 40:

```
if (inputs[i] == max)
{
    max = inputs[i];
}
```

This line is *supposed* to check each value of the "inputs" array to see if it is *greater than* the current "max" value. If not, then it's *less than* our current "max" value, so we don't do anything. If it is, then we've found a new current "max" value and we will store that value as the new current "max" value.

In this case, the array index *"i"* is being used properly (not hard-coded to zero like in the previous bug), but there is a problem with the *"if"* check that determines if the value in the array is *greater than* the current "max" value.

Rather than checking if the next value in the array is *greater than* the current "max" value, the "if" statement is checking if the next value in the array is *equal to* the current "max" value:

```
if (inputs[i] == max)
```

That code needs to be changed to reflect a *greater than* check:

```
if (inputs[i] > max)
```

Debugging Code Example:

Once we fix this code, we can try re-running the program to see if we have completely fixed our problem. This time, the re-run generates the following output:

```
Average: 6
Maximum: 12
```

Now we are getting the expected results given out input set. The problems in this code have been fixed!

Code Example – With Post-Debugging Fixes

```
public class DebugExample {

    // Input numbers
    private static int inputs[] = {12, 5, 9, 3, 2, 5};

    public static void main(String args[])
    {
        int average = calculateAverage();
        System.out.println("Average: " + average);

        int maxNum = calculateMax();
        System.out.println("Maximum: " + maxNum);
    }


    private static int calculateAverage()
    {
        int avg = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            avg += inputs[i];
        }
        return avg / inputs.length;
    }

    private static int calculateMax()
    {
        int max = 0;
        for (int i = 0; i < inputs.length; i++)
        {
            if (inputs[i] > max)
            {
                max = inputs[i];
            }
        }
        return max;
    }}
```

Summary:

- Debugging is a methodical process of finding and reducing the number of bugs in a computer program.
- The term "debugging" is used to cover a wide range of tasks but generally applies to the process of (1) detecting, (2) locating, and (3) repairing "bugs."
- Debuggers let us follow our program execution step by step, examine data (including the call stack), and set “break points”.
- Trace debugging (walking step-by-step through the execution of your code) allows you to see, in a visual way, how your code is executing, and to watch the state of variables, the stack, and threads. Tools and IDE’s that allow trace debugging are the preferred way to locate bugs in Java software.
- Debuggers allow you to step into or past functions and methods.
- Remember - finding a bug is the process of confirming the things that you believe to be true about the way your program is running – until you find one that is not true.
- To specify where you want to stop normal execution, you need to specify a break point. Upon debugger execution, normal execution will stop at this point and debugger-execution will commence allowing you to examine your program.
- The use of professional debugging techniques such as those detailed here will allow you to quickly discover the source of problems in your code, thereby increasing your productivity.