Finding Logical Errors

Kinds of Errors

- When coding there are different *kinds* of errors you can make while programming
  - Syntax errors: caught at compile time. Things that make it so the compiler cannot understand what you are trying to do (missing semicolons, typo in a type name, etc)
  - Semantic errors: happen at runtime. Your program compiles and runs, but produces incorrect results. Include things like divide by zero, using incorrect types, logical errors, etc.
- The compiler/IDE will indicate syntax errors, so we will focus on finding the source of semantic errors

Tips to Avoid Semantic Errors

- Have an understanding of the goal of your code.
- Write code you can explain (use good variable names)
- Keep methods small, and have them accomplish one thing

Finding Semantic Errors: Tracing Code

- Sometimes it helps to keep track of what your code is doing by writing out your *program state* every time a variable changes.
- This can help us determine where our program is going wrong

Exercise

Trace the following code:

```
int Rn = 0;
int n1, n2, n;
final int a = 5, b = 6, m = 13;
for (int i=0; i<5; i++)
{
  n1 = Rn * a;
  n2 = n1 + b;
  n = n2 % m;
  Rn = n;
}
```

Exercise Solution

i	n1	n2	n	Rn
0	0	6	6	6
1	30	36	10	10
2	50	56	4	4
3	20	26	0	0
4	0	6	6	6

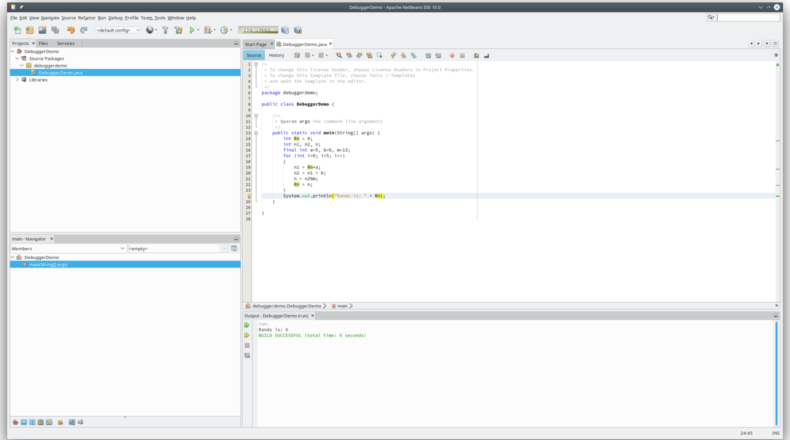
Tracing Cons

- Work intensive (large programs can be impractical to trace by hand)
- We want to be right so desparately
    - Possible to make a mistake as we are tracing by hand
    - The code might not be doing what we think it's doing, but we trace it as doing what we *want* it to be doing

Automated Tracing: The Debugger

- Many (if not all) programming languages have the ability to attach a *debugger* to a program
- A debugger allows you to pause/stop program execution, view program state, and evaluate expressions
- Most IDEs will have a debugger integrated into the UI; the following slides will demonstrate the basics of using the debugger in Netbeans 10 (which will look similar to the debugger in previous versions of Netbeans)

Debugger Example

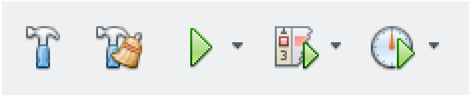


```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package debuggerdemo;

public class DebuggerDemo {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int Rn = 0;
        int n1, n2, n;
        final int a=5, b=6, m=13;
        for (int i=0; i<5; i++)
        {
            n1 = Rn*a;
            n2 = n1 + b;
            n = n2%m;
            Rn = n;
        }
        System.out.println("Rando is: " + Rn);
    }

}
```

```
run:
Rando is: 6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Using the Debugger

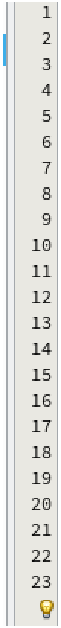
- Notice in the upper right corner (the run toolbar) there is a button that looks a square with the run triangle on it (second from the right)



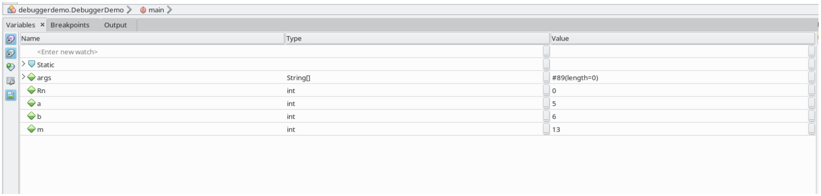
- This will start the debuggger
- Just clicking that button doesn't seem to do anything; the console reports that the program ran and the Process finished with exit code 0. What gives?

Breakpoints

- To really use the debugger, we need to set a *breakpoint*
  - A place in code where the debugger will pause the running program and wait for us to interact with it in some way.
- The area to the right of the source code (with the line numbers) is sometimes called the gutter.
- Go to line 19 and click on the line number. Notice that it changes from a 19 to a red square.
- Start the debugger again with the button in the run toolbar



The Debugger

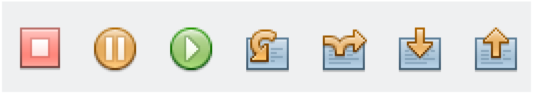

debuggerdemo.DebuggerDemo > main >

Variables × | Breakpoints | Output

Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#89(length=0)
Rn	int	0
a	int	5
b	int	6
m	int	13

Line 19 is highlighted in the source code.

New buttons appeared next to the run toolbar:



Using the Debugger

- The program is now paused and waiting for us to do things.
- The program has *NOT* executed line 19 yet.
- For now, we will focus on tracing through our program.
- Consider the buttons in the upper toolbar:

  Step Over: execute the line of code and pause execution at the next line of the current method.

  Step Into: execute the current line of code. If there is a method call in this line of code, pause execution on the first line of that method.

  Step Out Of: continue executing code in the current method; pause at the next line of the method that called the current method.

- Click the Step Over button Notice that in the Variables window, there is now a variable called `i` with a value of zero. Line 20 should now be highlighted in the source window

Walking Through Code

- If you keep clicking the Step Over button, you should see that the values of the variables change in the Variables window. If you click a total of 5 times, you will notice that line 19 is now highlighted again, and the variables i, n1, n2, n, and Rn all match the first row of the table from our manual trace
- Notice that when a variable changes its value the new value is bolded (shows the last variable to change).

Speeding Things Up

- Sometimes manually walking through code takes some time. We can speed up the process by using the Continue button on the left hand side of the debugging window.
- This will run the code until it hits a breakpoint (or the end of the program). If you click the continue button, it should again pause on line 19, but the values of the variables should have changed (marked in blue).
- You can always add more breakpoints and remove them (click the red dot next to the line; it's a toggle!)
- You can always stop the debugger with the Stop button (red square).

Debuggers

- Debuggers have many more powerful features that were not covered in this basic tutorial
  - e.g. Expression Evaluation
- Debuggers will let you see the contents of objects, arrays, and more!
- If your code is exhibiting a logical error, ask yourself:
  - “What sould the code be doing?”
  - “What is it actually doing?”
- Use the debugger to determine the answer to the second question