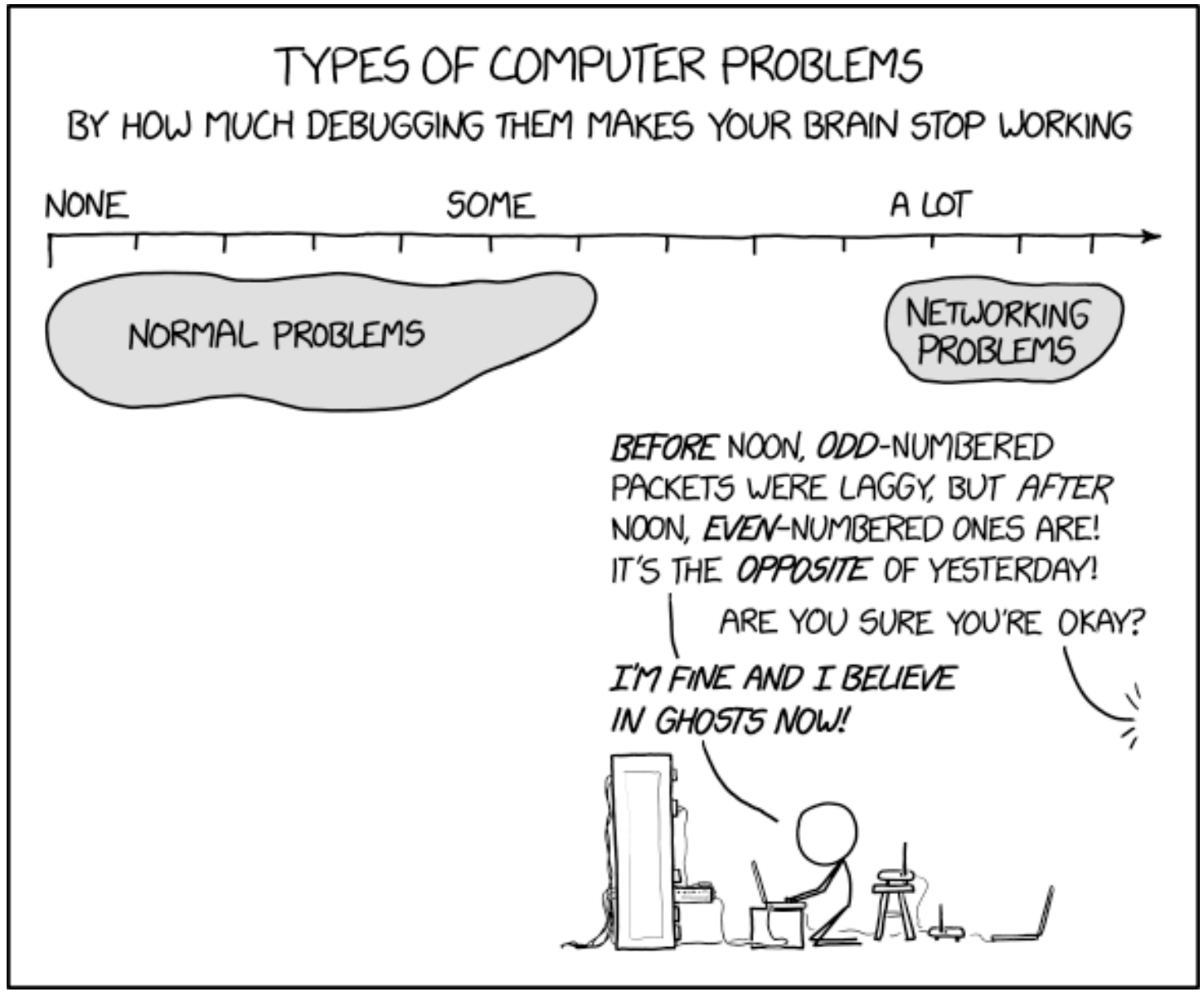


LOOK, THE LATENCY FALLS EVERY TIME YOU CLAP YOUR HANDS AND SAY YOU BELIEVE

Debugging

CSCI 2134: Software Development

Agenda

- Lecture Contents
  - Motivation
  - Debugging with the scientific method
  - Using a debugger
  - Fixing the bug
- Brightspace Quiz
- Readings:
  - This Lecture: Chapter 23
  - Next Lecture: Chapter N/A (Common Coding Mistakes)

Why Debug?

- What is debugging?

  Debugging is the process of locating and correcting the root cause of defects
- Why debug?

  Because we do not write perfect code
- How do we know we have a bug?

  Our program exhibits **symptoms** indicating there is a problem
- What do we have to do?
  1. **Find** the root cause of the bug (a.k.a defect)
  2. **Fix** the root cause
- What should we not do?

  Fix the symptom instead of the root cause

Bugs are Opportunities to Learn about …

(McConnel, CC2, 2004)

- **The program you're working on**
- The kinds of mistakes you make
- The quality of your code from the point of view of someone who has to read
- How you solve problems
- How you fix defects

How Not to Debug

(McConnel, CC2, 2004)

- Find the defect by guessing
- Don't spend time to understand the problem
- Fix the error with the obvious fix
- **Use lots of printf statements**
  - To see where the code reaches or doesn't reach
  - To derive the path taken by the program
  - To watch how variable values change
- Not use a systematic approach

Original code:

```
int sum = 0;
for (int i = 0; i <= 10; i++) {
   sum += i;
}
return sum;
```

Printf code:

```
sum = 0;
System.out.println("Enter loop");
for (int i = 0; i <= 10; i++) {
  System.out.println("Iteration " +
       i + " sum = " + sum);
  sum += i;
}
System.out.println("Sum is " + sum);
return sum;
```

Sift This!

Enter loop
Iteration 0 sum = 0
Iteration 1 sum = 0
Iteration 2 sum = 1
Iteration 3 sum = 3
Iteration 4 sum = 6
Iteration 5 sum = 10
Iteration 6 sum = 15
Iteration 7 sum = 21
Iteration 8 sum = 28
Iteration 9 sum = 36
Iteration 10 sum = 45
Iteration 11 sum = 55
Iteration 12 sum = 66
Iteration 13 sum = 78
Iteration 14 sum = 91
Iteration 15 sum = 105
Iteration 16 sum = 120
Iteration 17 sum = 136
Iteration 18 sum = 153
Iteration 19 sum = 171
Iteration 20 sum = 190
Iteration 21 sum = 210
Iteration 22 sum = 231
Iteration 23 sum = 253
Iteration 24 sum = 276
Iteration 25 sum = 300
Iteration 26 sum = 325
Iteration 27 sum = 351
Iteration 28 sum = 378
Iteration 29 sum = 406
Iteration 30 sum = 435
Iteration 31 sum = 465
Iteration 32 sum = 496
Iteration 33 sum = 528
Iteration 34 sum = 561
Iteration 35 sum = 595
Iteration 36 sum = 630
Iteration 37 sum = 666
Iteration 38 sum = 703
Iteration 39 sum = 741
Iteration 40 sum = 780
Iteration 41 sum = 820
Iteration 42 sum = 861
Iteration 43 sum = 903
Iteration 44 sum = 946
Iteration 45 sum = 990
Iteration 46 sum = 1035
Iteration 47 sum = 1081
Iteration 48 sum = 1128
Iteration 49 sum = 1176
Iteration 50 sum = 1225
Iteration 51 sum = 1275
Iteration 52 sum = 1326
Iteration 53 sum = 1378
Iteration 54 sum = 1431
Iteration 55 sum = 1485
Iteration 56 sum = 1540
Iteration 57 sum = 1596
Iteration 58 sum = 1653
Iteration 59 sum = 1711
Iteration 60 sum = 1770
Iteration 61 sum = 1830
Iteration 62 sum = 1891
Iteration 63 sum = 1953
Iteration 64 sum = 2016
Iteration 65 sum = 2080
Iteration 66 sum = 2145
Iteration 67 sum = 2211
Iteration 68 sum = 2278
Iteration 69 sum = 2346
Iteration 70 sum = 2415
Iteration 71 sum = 2485
Iteration 72 sum = 2556
Iteration 73 sum = 2628
Iteration 74 sum = 2701
Iteration 75 sum = 2775
Iteration 76 sum = 2850
Iteration 77 sum = 2926
Iteration 78 sum = 3003
Iteration 79 sum = 3081
Iteration 80 sum = 3160
Iteration 81 sum = 3240
Iteration 82 sum = 3321
Iteration 83 sum = 3403
Iteration 84 sum = 3486
Iteration 85 sum = 3570
Iteration 86 sum = 3655
Iteration 87 sum = 3741
Iteration 88 sum = 3828
Iteration 89 sum = 3916
Iteration 90 sum = 4005
Iteration 91 sum = 4095
Iteration 92 sum = 4186
Iteration 93 sum = 4278
Iteration 94 sum = 4371
Iteration 95 sum = 4465
Iteration 96 sum = 4560
Iteration 97 sum = 4656
Iteration 98 sum = 4753
Iteration 99 sum = 4851
Iteration 100 sum = 4950
Sum is 5050

What's wrong with the `printf` method?

- Problems:
  - **Error prone:** Requires modification of code as part of debugging
  - **Information overload**: The output has to be sifted (LOTS of output)
  - **Takes a long time**: Requires recompiling and running of code many times
  - **Obscures the bug**: Can make the bug nonreproducible. ☹

The Scientific Method (for Debugging)

(McConnel, CC 2, 2004)

Scientific Method for Science

1. Gather data through repeatable experiments.
2. Form a hypothesis that accounts for the relevant data.
3. Design an experiment to prove or disprove the hypothesis.
4. Prove or disprove the hypothesis.
5. Repeat as needed.

Scientific Method for Debugging

1. Stabilize the error.
2. Locate the root cause of the bug
   a. Gather the data that produces the bug.
   b. Analyze the data that has been gathered and form a hypothesis about the bug.
   c. Determine how to prove or disprove the hypothesis, either by testing the program or by examining the code.
   d. Prove or disprove the hypothesis by using the procedure identified in 2(c).
3. Fix the defect.
4. Test the fix.
5. Look for similar errors.

Stabilize the Bug (Error)

- The best kind of bug is:
  - **Reproducible**:
    - Can be reliably reproduced
    - Bugs that appear and disappear during debugging are hard to find

      Common causes:
      - In languages like C or C++ (Java avoids these)
        - Uninitialized variables
        - Stale references
      - In any language
        - user interaction
        - race conditions
  - **Easily duplicatable**:
    - Have small, unchanging set of conditions under which the bug occurs
    - Has a small simple test-case
- The first task is to create a test-case that
  - Causes the bug to occur every time it is executed (reproducible)
  - Is as small as possible (easily duplicatable)

Example: A Sorted Employee List

Scenario

- You are working on a program to track employees
- Given a list of employees, your program outputs a sorted list of employees
- You notice that the list is not quite sorted.

What do you do?

1. Rerun the test with the same input
   Ensure reproducibility
2. Reduce size of input to the smallest possible such that the bug is reproducible

```
while (test is reproducible)
    reduce test size
    rerun test
```

- Input (Unsorted)

```
Global, Gary
Statement, Sue Switch
Modula, Mavis
Formatting, Frita
Freeform Fruit-Loop, Fred
Many-Loop, Mildred
Whileloop, Wendy
```

- Output (Sorted)

```
Freeform Fruit-Loop, Fred
Formatting, Frita
Global, Gary
Modula, Mavis
Many-Loop, Mildred
Statement, Sue Switch
Whileloop, Wendy
```

Our *Employee* class

```
public class Employee implements Comparable<Employee> {
  private String firstName;
  private String lastName;

  public Employee(String first, String last) {
    firstName = first;
    lastName = last;
  }

  public String toString() {
    return lastName + ", " + firstName;
  }

  public int compareTo(Employee e) {
    int c = getLastName().compareTo((e.getLastName());
    if (c == 0) {
      c = getFirstName().compareTo(e.getFirstName());
    }
    return c;
  }
  ...
```

Use our “Scientific Method” to Locate the Bug

Round 1

- Gather the data that produces the bug
  - Our test case
  - Conditions under which the bug occurs
- Form a hypothesis
  - Names are not being properly store
- Determine how to test hypothesis
  - Look at constructor of *Employee* class
  - Ensure names are stored correctly in unsorted list
- Test hypothesis
  - The code for `toString()` is

    ```
    return lastName + ", " + firstName;
    ```
  - Constructor is only piece of code that sets `firstName` and `lastName`
  - Names are correctly printed
  - Hypothesis is not correct

Round 2

- Gather the data that produces the bug
  - Our test case
  - Conditions under which the bug occurs
  - Names are being properly stored and printed
- Form a hypothesis
  - The sorting algorithm is not working
- Determine how to test hypothesis
  - Look at code that sorts the employees
- Test hypothesis
  - The code uses the built-in method

    ```
    Collections.sort()
    ```
  - Unlikely to be broken
  - Hypothesis is not correct

Use our “Scientific Method” to Locate the Bug

Round 3

- Gather the data that produces the bug
  - Our test case
  - Conditions under which the bug occurs
  - Names are being correctly stored and printed
  - Uses well tested sorting algorithm
- Form a hypothesis
  - Names are not being compared properly
- Determine how to test hypothesis
  - Test `compareTo()` method
- Test hypothesis
  - `compareTo()` returns opposite expected result when comparing
    - “`Freeform Fruit-Loop, Fred`”
    - “`Formatting, Frita`”

```
public int compareTo(Employee e) {
  String last = e.getLastName();
  int c = getLastName().compareTo(last);
  if (c == 0) {
    String first = e.getFirstName();
    c = getFirstName().compareTo(first);
  }
  return c;
}
```

Use our “Scientific Method” to Locate the Bug

Round 4

- Gather the data that produces the bug
  - Our test case
  - Conditions under which the bug occurs
  - Names are being correctly stored and printed
  - Uses well tested sorting algorithm
  - `compareTo()` is not working
- Form a hypothesis
  - Last names are not being compared properly
- Determine how to test hypothesis
  - Test `getLastName()` method
- Test hypothesis
  - `getLastName()` is returning first name not last name!

```
public int compareTo(Employee e) {
  String last = e.getLastName();
  int c = getLastName().compareTo(last);
  if (c == 0) {
    String first = e.getFirstName();
    c = getFirstName().compareTo(first);
  }
  return c;
}

public String getFirstName() {
  return firstName;
}

public String getLastName() {
  return firstName;
}
```

Gathering Data

- Use all available data to generate hypotheses
  - What is known about the code
  - Results of negative tests
    - Allows you to rule out some hypotheses
    - All hypotheses tests are useful as they generate additional data
  - Ways in which the error is reproduced
    - An error that occurs in several different ways must have a common point
- Generate more data to generate more hypotheses

  Example:
  - In each round of our debugging we added to our knowledge
  - Each round led to a further hypothesis
- **Note**: Keep notes on your debugging to avoid getting lost

Forming a Hypothesis

Question: Where do we start?

- Code that's changed recently

  Many bugs are introduced by changes to the code

- Classes and methods that have had defects before

  Code that was buggy may

  - Be complex
  - Not be well written
  - Have fixes that cause other bugs

- Common defects

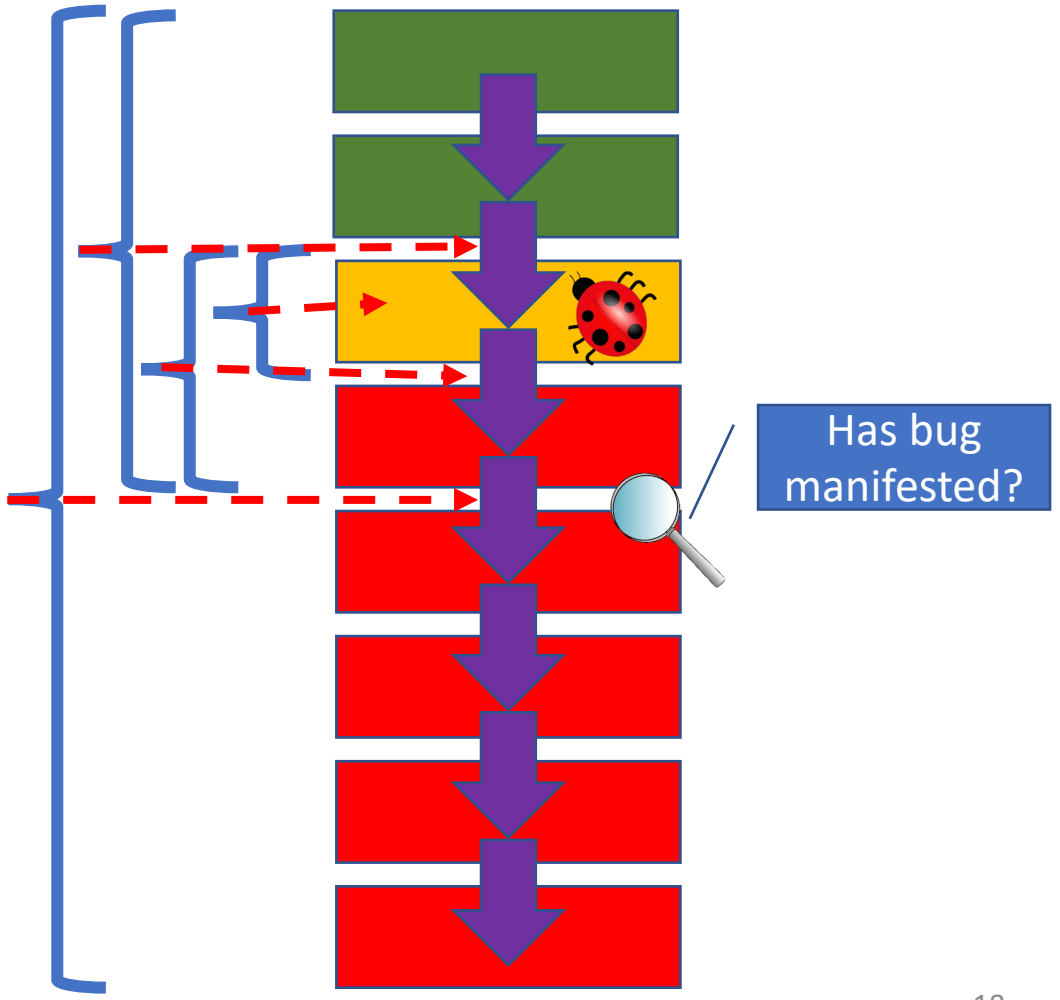
  New code will have more common bugs than old code

Narrow the Search with Binary Search

Question: Where do we go next?

- The goal is to locate the root cause of the bug
- Idea: locate the earliest point in the code where a symptom appears
- Approach: Use a binary search
  - Hypothesis: The bug is in the 1st half of the region?
  - Test Hypothesis:
    - If yes, focus on 1st half
    - If no, focus on 2nd half
  - Repeat

- `git bisect` lets you perform a binary search over commits



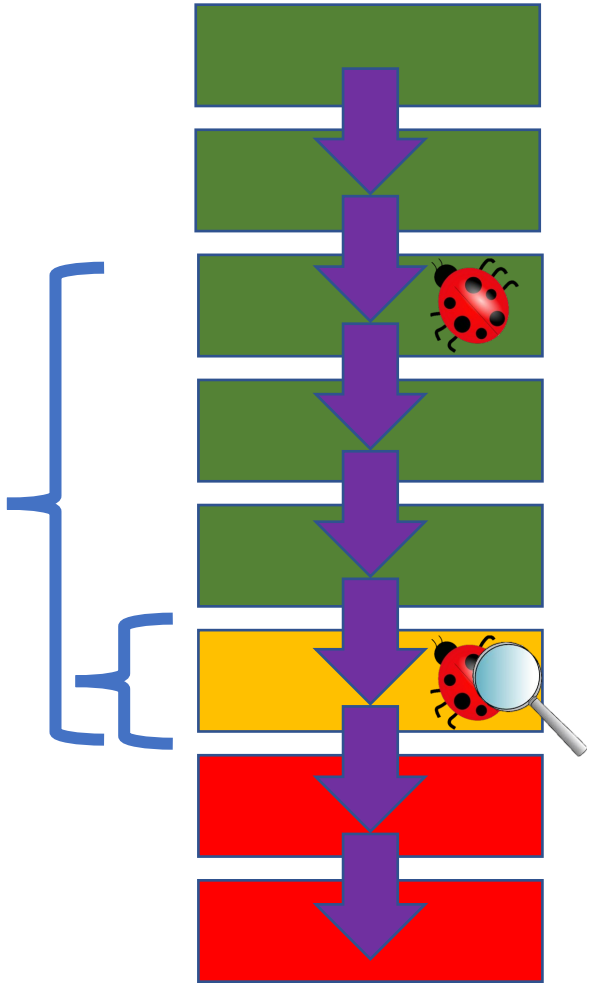
Expand the Search

Question: What if look in the wrong place?

- It is possible that we focus on a region where there is no bug, but where the symptoms appear

Question: What do we do?

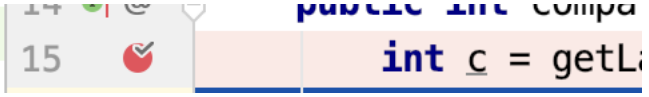
- Expand the suspicious region of the code
- Create hypotheses as to
  - What would cause the symptoms?
  - What other symptoms should appear earlier
- Test the hypotheses



Test the Hypothesis: Use the Right Tools

- The right tools for most of our debugging needs is the visual (symbolic) debugger
- The debugger allows the programmer to view the program as it runs
  - Pause at various locations in the program (breakpoints)
  - Inspect the values in variables
  - Examine objects and data structures
  - Examine code
  - Execute program until a specified variable is set (watchpoints)
  - Run additional code on the fly
- Using a debugger vs printf statement is like driving a Porche vs a unicycle.

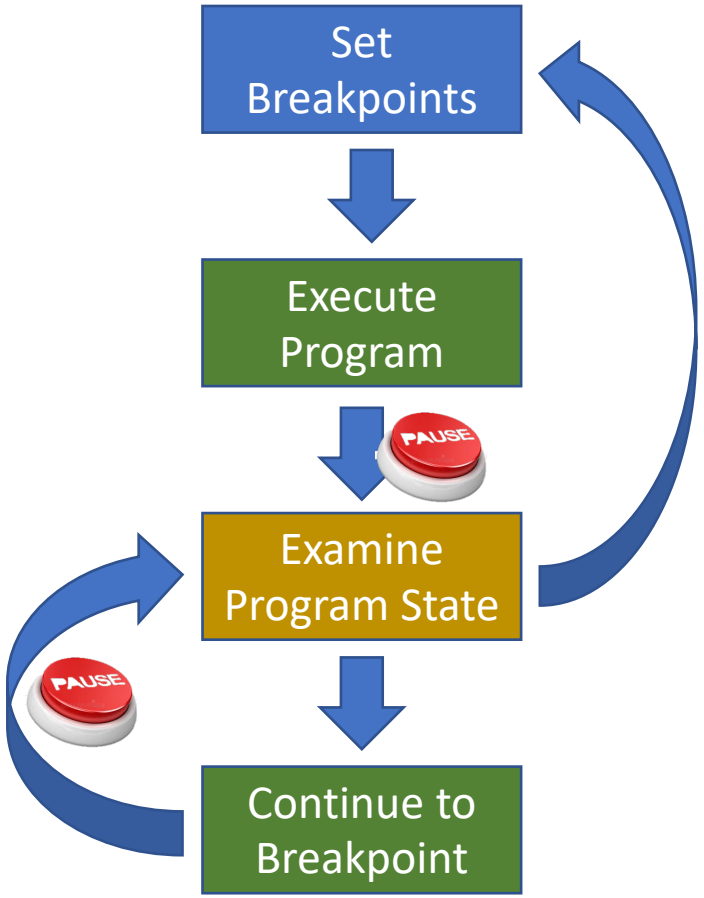
Debugger Jargon



- **Breakpoint**:
  A position in the code where the debugger should pause execution to allow the developer to examine the program state
- **Step Over / Next**:
  Execute current line of code, including calling any methods on that line 
- **Step / Step Into**:
  Execute the current line of code, step into the next method on the line. 
- **Finish / Step Out / Step return**:
  Run to end of current method and return 
- **Continue**:
  Continue running paused program 
- **Run to**:
  Run to cursor (a temporary break point) 
- **Watchpoint**:
  Like a breakpoint, but on a variable. The program executes until the value of the variable changes

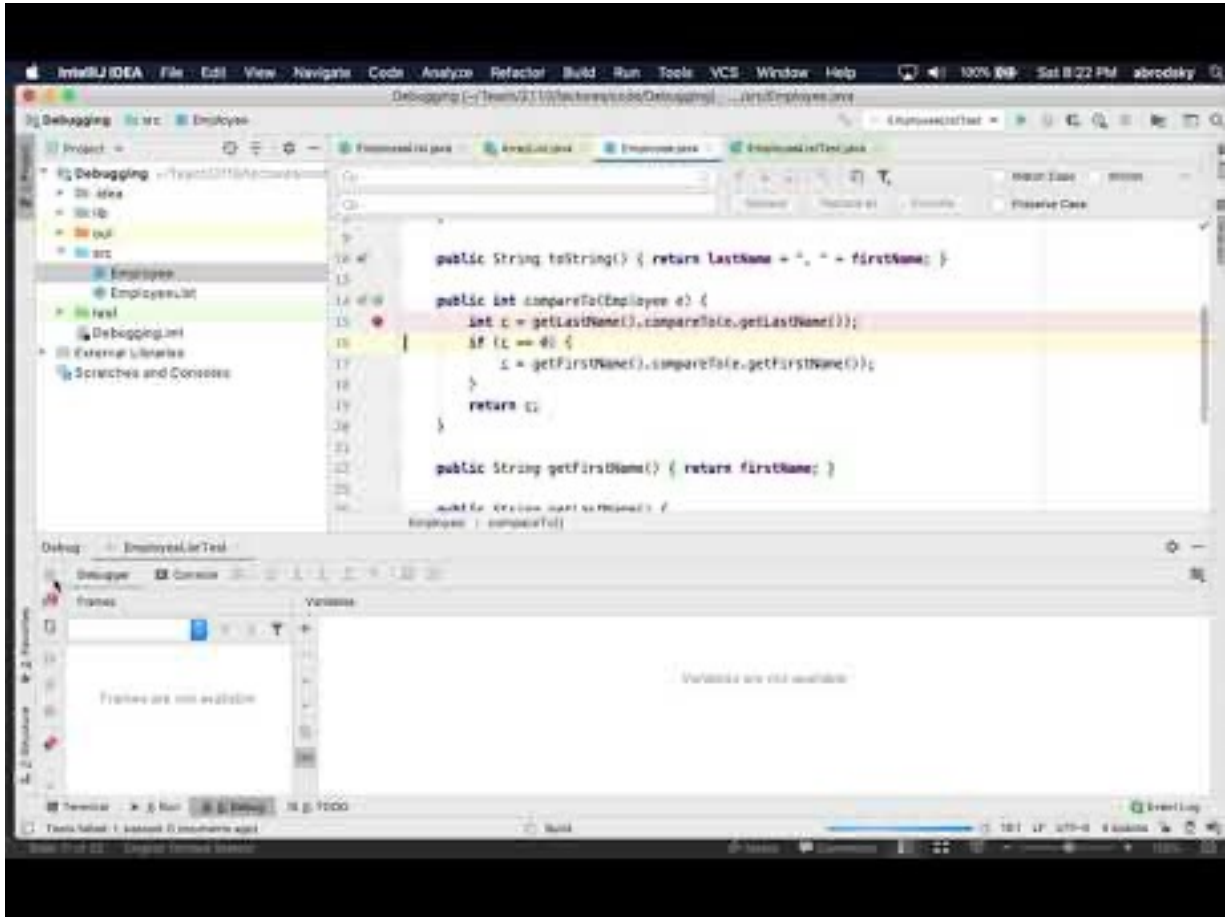
Typical Process when Using a Debugger

1. Set one or more breakpoints (locations depend on your hypothesis)
   - The breakpoints are typically located in the “suspicious” region of code
2. Execute program, execution will proceed until a breakpoint is hit
3. At a breakpoint:
   - Examine values of variables
   - Step through code, keeping track of program state
   - You should be able to **predict** what happens at each step
   - If your prediction is incorrect, then something may be wrong
4. If you do not discover anything wrong at the current breakpoint, continue running program to proceed to the next break point



Debugging Live:

See video below if you did not see the live demo



https://youtu.be/pmi0uhQ52js

What if you get stuck?

- It happens. You've spent 20 hours looking for a bug and still can't find it! ☹
- Now what?
  - Brainstorm for possible hypotheses
  - Make a list of things to try
  - Talk to someone else about the problem
  - Take a break from the problem
- What can I do as a software developer?
  - Exercise the code in your unit test suite
  - Integrate incrementally

Brute Force Debugging

Occasionally other quick solutions may work:

- Recompile with more warnings
- Following the code execution line-by-line (stepping through the code)
- Run the program in a different environment
  - May be needed if bug is not reproducible
- Backtrack on commits to determine which modification introduced the bug
- Re-run and/or expand previous unit tests to better isolate the error

Know When to Stop

Limit Brute-Force debugging to a short period of time

- Try the easy bits to isolate problems
- Don’t spend too much time on unproductive efforts
  - This applies to adding “print” statements

Sometimes debugging will fail

- You may need to do something different:
  - Use a different approach like a code review
  - Rewrite:

    Expensive, but sometimes better than never understanding the code that’s there

Rubber Duck Debugging

- Idea 1: explaining the problem to someone else can lead to breakthroughs in your own understanding
- Idea 2: *thinking* to yourself is not enough
- Idea 3: needing another person's help to debug is not always the best use of human resources.

Rubber Duck Debugging:

- Explain the problem, and what is expected, to an inanimate object (e.g. rubber duck). Articulating the problem vocally will help you find its root cause.

Fixing Bugs: Before the fix

- **Step 0**: Understand what needs to be fixed
  - Fix the problem (root cause), not the symptom
  - Before making a fix
    - Understand the problem
    - Understand the program
- **Step 1**: Before performing the fix
  - Confirm the bug
  - Relax
  - Ensure the current version is committed

Fixing Bugs: Implementation

- **Step 2**: Implement the fix
  - Fix the problem (root cause), not the symptom
  - Change the code only when necessary
  - Make one change at a time
  - Check your fix (rerun all regression tests)
- **Step 3**: After the fix
  - Add a unit test that exposes the defect
  - Look for similar defects

Key  Points

- Debugging consists of two (or three) tasks: finding the bug, correcting it, (writing new tests).
- Effective debugging requires a structured approach and appropriate tools
- An iterative approach to debugging involves:
  - i) gathering data about the bug
  - ii) formulating a hypothesis about the cause of the bug
  - iii) determining how to test the hypothesis
  - iv) testing the hypothesis
  - v) updating our understanding of the bug
- When fixing a bug, it is important to have a full understanding of what the problem is and how the fix will affect the program

Image References

Retrieved January 29, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- https://i.pinimg.com/originals/72/2c/4b/722c4bef3d62e450fd07d43977beff3f.jpg