



# Chapter 7: Software Construction for Robustness

## 7.4 Debugging

Ming Liu

April 17, 2019

# Outline

- **What is bug and debugging?**
- **Process for debugging**
  - Reproduce the bug
  - Diagnosing the bug
  - Fix the bug
  - Reflection
- **Debugging techniques and tools**
  - Print debugging / stack tracing / memory dump
  - Logging
  - Compiler Warning Messages
  - Debugger: watch points, break points, etc
- **Summary**

# Bug? De-bug?

- The terms "bug" and "debugging" are popularly attributed to Admiral Grace Hopper in the 1940s.
- While she was working on a Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system.

9/9

0800 Antan started  
 1000 " stopped - antan ✓


1300 (033) MP-MC ~~1.98264000~~ 2.130476415  
 (033) PRO 2 2.130476415  
 correct 2.130676415

Relays 6-2 in 033 failed special speed test  
 in relay " 10.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)  
 1525 Started Multy Adder Test.

1545



Relay #70 Panel F  
 (moth) in relay.

First actual case of bug being found.

1630 Antan started.  
 1700 closed down.

Relay 3372



# 1 What is Bug and What is debugging





# (1) What is a Bug?



# What is a Bug?

## ■ What is a Bug?

- A fault in a program, which causes the program to perform in an unintended or unanticipated manner.
- A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be buggy.
- Reports detailing bugs in a program are commonly known as bug reports, fault reports, problem reports, trouble reports, defect reports etc.

## ■ Why Bug Occurs?

- Code errors
- Unfinished requirements or not detailed enough
- Misunderstanding of user needs
- Logic errors in the design documents
- Lack of documentation
- Not enough sufficient testing

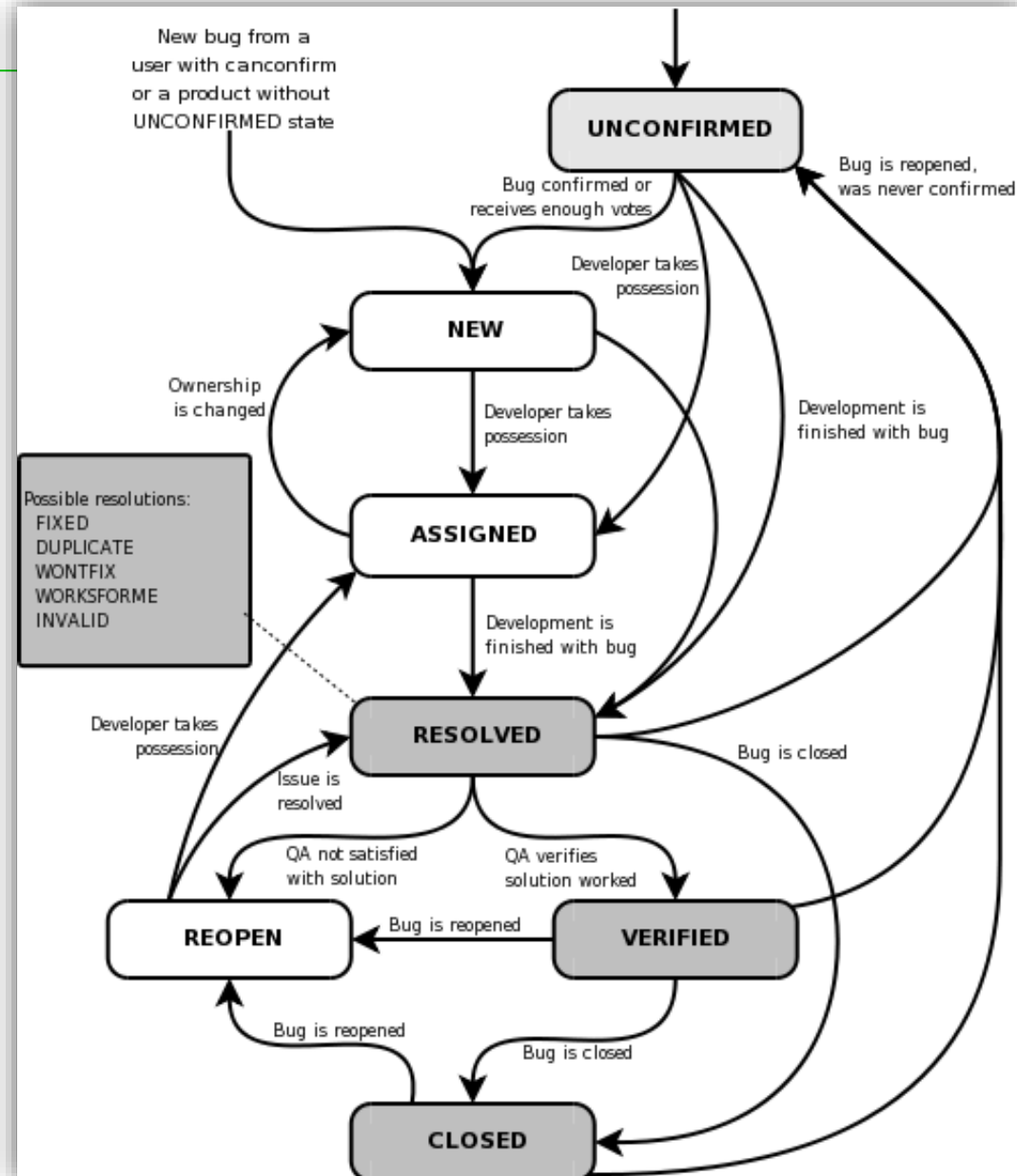


## (2) Bug Lifecycle



# Bug Lifecycle

- A bug should go through the life cycle to be closed. The bug attains different states in the life cycle.
- States of a bug:
  - New
  - Open
  - Assign
  - Tested/Verified
  - Deferred
  - Reopened
  - Rejected
  - Close







## (3) Common Types of Bugs



# Common Types of Bugs

- **Maths bugs, such as Division by zero, Arithmetic overflow**
- **Logic bugs, such as Infinite loops and infinite recursion**
- **Syntax bugs, such as Use of the wrong operator, such as performing assignment instead of equality**
- **Resource bugs**
  - Using an un-initialized variable
  - Resource leaks, where a finite system resource such as memory or file handles are exhausted by repeated allocation without release.
  - Buffer overflow, in which a program tries to store data past the end of allocated storage.
- **Team working bugs**
  - Comments out of date or incorrect
  - Differences between documentation and the actual product



## (4) Bug report system and issue tracking system



# Bugzilla



# Bugzilla

- **Bugzilla is a web-based general-purpose bug tracker and testing tool originally developed and used by the Mozilla project**
  - Report a new bug
  - Search for an existing bug
  - See summary bug reports and charts
  - Make contributions to bug fix
- **First released in 1998, it has been adopted by a variety of organizations for use as a bug tracking system for both free and open-source software and proprietary projects and products.**

# Adding a bug

**Bug 169837 - scroll bookmarks but not other menu items in bookmarks menu - Mozilla**

Firefox Edition Affichage Aller à Marque-pages Outils Fenêtre Aide

Précédent Suivant Actualiser Arrêter [https://bugzilla.mozilla.org/show\\_bug.cgi?id=169837](https://bugzilla.mozilla.org/show_bug.cgi?id=169837)

Accueil Marque-pages Le site Mozilla Mozilla en français

**mozilla.org** Bugzilla Version 2.19.1+

**Bugzilla Bug 169837** scroll bookmarks but not other menu items in bookmarks menu

[Search page](#) [Enter new bug](#)

Bug#: 169837 alias:  Hardware: PC  
 Product: Firefox OS: All  
 Component: Bookmarks Version: unspecified  
 Status: NEW Priority: P4  
 Resolution: Severity: normal  
 Assigned To: Vladimir Vukicevic (Bookmarks Bugs Only) <vladimir+bm@vlad1.com> Target Milestone: Future  
 QA Contact: mconnor@steelgyphon.com  
 URL:   
 Summary: scroll bookmarks but not other menu items in bookmarks menu  
 Status Whiteboard:   
 Keywords:

Attachment	Type	Created	Size	Flags	Actions
<a href="#">patch</a>	patch	2004-07-25 08:55 PDT	5.74 KB	none	<a href="#">Edit</a>   <a href="#">Diff</a>

[Create a New Attachment](#) (proposed patch, testcase, etc.) [View All](#)

Bug 169837 depends on:  [Show dependency tree](#)  
 Bug 169837 blocks:  [Show dependency graph](#)  
 Votes: 8 [Show votes for this bug](#) [Vote for this bug](#)

Additional Comments:

**Bug 52094 - hyatt should give ben \$50 - Mozilla Firefox 3 Beta 3 (Build 2008020513)**

File Edit View History Bookmarks Tools Help

[https://bugzilla.mozilla.org/show\\_bug.cgi?id=52094](#) Wikipedia (EN)

Home Smart Bookmarks Places frmget Gordon Hill sort table numeri number rows

**Bugzilla@Mozilla - Bug 52094**

[Home](#) | [New](#) | [Search](#) |  Find | [Reports](#) | [My Requests](#) | [My Votes](#) | [Preferences](#) | [Log out](#) gerv@mozilla.org

**Bug List:** (This bug is not in your last search results) [Show last search results](#) [Last Comment](#)

**Bug 52094 - hyatt should give ben \$50 (edit)**

**Status:** VERIFIED FIXED  
**Severity:** critical  
**Keywords:** access, helpwanted, meta, modern, pp, testca  
**Whiteboard:**  
**URL:** http://www.zachlipton.com/ben  
**Product:** Core  
**Component:** Tracking  
**Version:** Trunk  
**Hardware:** PC  
**OS:** Windows 2000  
**Assigned To:** David Hyatt  
**QA Contact:** John Morrison (edit)  
**Priority:** P1  
**Target Milestone:** Future  
**Depends on:** (edit)  
**Blocks:** [215379](#) (edit)  
[Show dependency tree - Show dependency graph](#)

**Reported:** 2000-09-10 23:08 PST by [Ben Goodger \(use ben at mozilla...\)](#)  
**Modified:** 2006-10-31 09:50:17 PST ([View Bug Activity](#))  
**Votes:** 10 ([show](#)) ([vote](#))  
**Add CC:**   
**CC:** adam@gimp.org  
 andersma@charter.net  
 berkut.bugzilla@gmail.com  
 bhart@cvip.net  
 brian@mozdev.org  
☐ Remove selected CCs

**Flags:**  
 blocking1.8.0.15 ☐  
 wanted1.8.0.x ☐  
 blocking1.8.1.13 ☐  
 wanted1.8.1.x ☐  
 blocking1.9 ☐  
 wanted1.9 ☐  
 wanted1.9.0.x ☐  
 in-litmus ☐  
 in-testsuite ☐

**Attachments**

Attachment	Size	Flags	Details
<a href="#">Testcase for breakage.</a> (991 bytes, text/html)		no flags	<a href="#">Details</a>

Done bugzilla.mozilla.org



## (4) Debugging



# Defensive Programming → Testing → Debugging

- **Defensive Programming:** to make bugs impossible at design time;
- **Testing:** to uncover problems in a program and thereby increase your confidence in the program's correctness.
- **So what?**
- **What if some tests fail?**
- **You have no choice but to debug** – particularly when the bug is found only when you plug the whole system together, or reported by a user after the system is deployed, in which case it may be hard to localize it to a particular module.

# What is Debugging?

- **Debugging is the process of identifying the root cause of an error and correcting it.**
- **It contrasts with testing, which is the process of detecting the error initially, debugging occurs as a consequence of successful testing.**
  - On some projects, debugging occupies as much as 50 percent of the total development time.
  - For many programmers, debugging is the hardest part of programming.
- **Like testing, debugging isn't a way to improve the quality of your software, but it's a way to diagnose defects.**
  - Software quality must be built in from the start. The best way to build a quality product is to develop requirements carefully, design well, and use high-quality coding practices.
  - **Debugging is a last resort.**



# Why is debugging so difficult?

- **The symptom and the cause may be geographically remote.**
  - The symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed.
  - Highly coupled components exacerbate this situation.
- **The symptom may disappear (temporarily) when another error is corrected.**
- **The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).**
- **The symptom may be caused by human error that is not easily traced.**

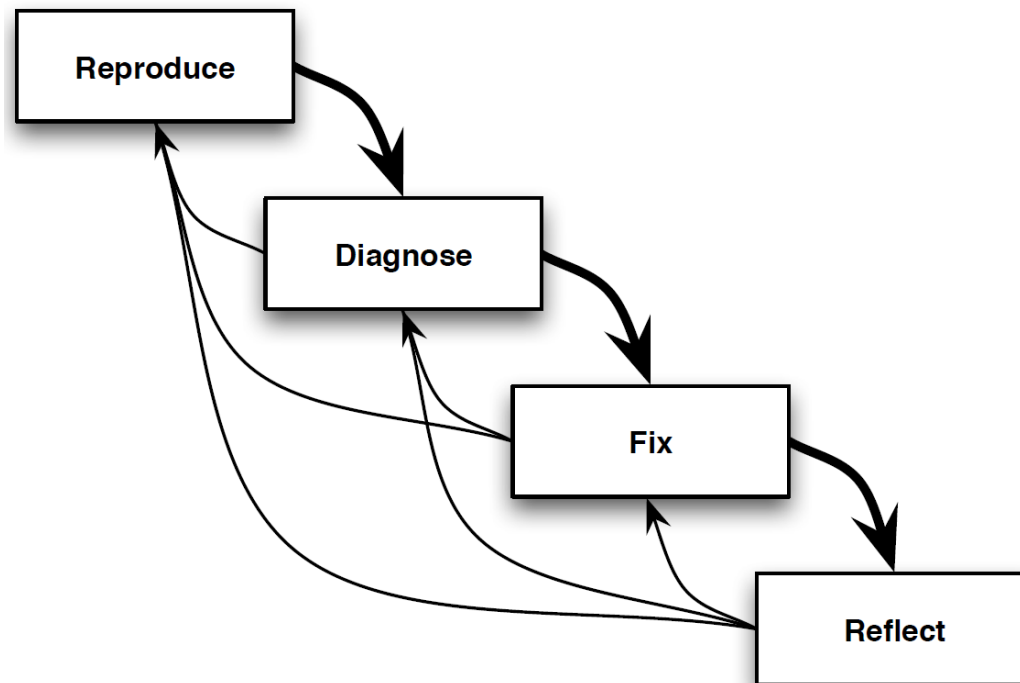


## 2 Process for debugging



# Debugging Process

- **Reproduce:** Find a way to reliably and conveniently reproduce the problem on demand.
- **Diagnose/Locating:** Construct hypotheses, and test them by performing experiments until you are confident that you have identified the underlying cause of the bug.



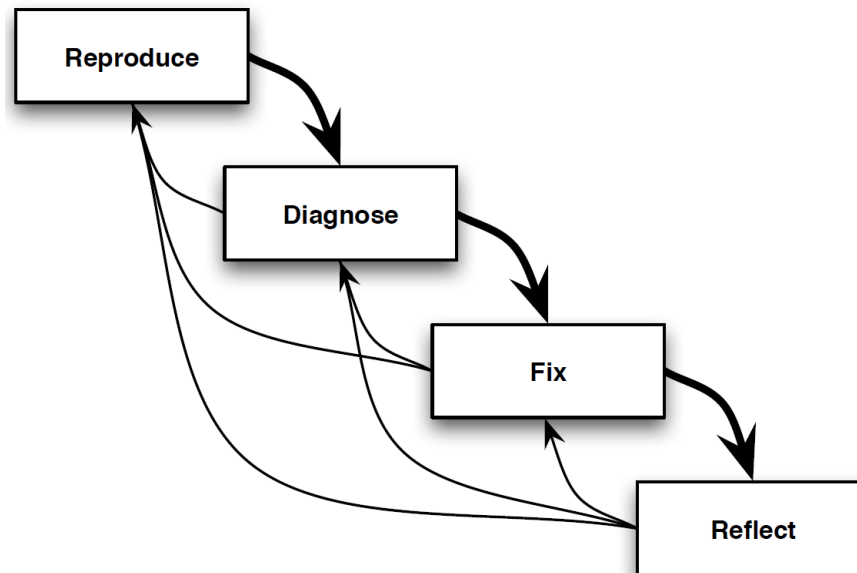
# Debugging Process

## ■ Fix:

- Design and implement changes that fix the problem, avoid introducing regressions, and maintain or improve the overall quality of the software.

## ■ Reflect:

- Learn the lessons of the bug. Where did things go wrong? Are there any other examples of the same problem that will also need fixing? What can you do to ensure that the same problem doesn't happen again?



# The Scientific Method of Debugging

- **Here are the steps you go through when you use the scientific method:**
  - 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.



# (1) Reproduce the bug



# Reproduce the Bug

- Start by finding a small, repeatable test case that produces the failure.
- If the bug was found by regression testing, then you're in luck; you already have a failing test case in your test suite.
- If the bug was reported by a user, it may take some effort to reproduce the bug.
- For graphical user interfaces and multithreaded programs, a bug may be hard to reproduce consistently if it depends on timing of events or thread execution. (to be discussed in Chapter 10)

# Reproduce

- **Why is reproducing the problem so important? Because if you can't, then it's almost impossible to make progress.**
- **Specifically:**
  - The empirical process relies upon our ability to watch the software executing in the presence of the bug. If we can't get the software to misbehave in the first place, then this, the most powerful weapon in our armory, is lost.
  - Even if you do somehow manage to come up with a theory about why the software might be misbehaving, how are you going to prove it if you can't reproduce the problem?
  - If you think that you've implemented a fix, how are you going to demonstrate that it really does fix the problem?



# What to control in order to reproduce bugs?



## ■ The software itself

- If the bug is in an area that has changed recently, then ensuring that you're running the same version of the software as it was reported against is a good first step.

## ■ The environment it's running within

- If interaction with an external system (some particular piece of hardware or a remote server perhaps) is involved, then you probably want to ensure that you're using the same.

## ■ The inputs you provide to it

- If the bug is related to an area that behaves very differently depending upon how the software is configured, then start by replicating the user's configuration.

# (1) To control the software

## ■ Controlling the Software

- Firstly, compiling from the same source. You also need to ensure that you use the same compiler, configured in the same way, and the same runtime, libraries, and any third-party code that is integrated with your software.
- Of course, using the same tools gets you nowhere if you don't use them in exactly the right sequence and with the same configuration as the software was originally built with.
- The best way to ensure that you do is to create an automated build process.

## (2) Controlling the Environment

- **What constitutes your software's environment depends on what kind of software it is.**
  - For traditional desktop software, the operating system is probably most relevant.
  - For web software, it's the browser.
  - For network software, it's the other software you're communicating with,
  - and for embedded code, it's the hardware you're interfacing with.
- **The key in all cases is first knowing what environment the bug manifests in.**

## (3) Controlling Inputs

- Your software's inputs may be files on disc, sequences of user interface operations, or responses from third-party servers or hardware.
- Whatever form they take, the key is to first identify them so that you can then replay them exactly.
- The most lucky thing is that the relevant inputs will be specified in the bug report.
- If you don't have all the information you need, you can
  - Infer what the inputs might be
  - record the inputs.

# An example for controlling inputs

```
/**
 * Find the most common word in a string.
 * @param text string containing zero or more words, where a word
 *       is a string of alphanumeric characters bounded by nonalphanumerics.
 * @return a word that occurs maximally often in text, ignoring alphabetic case.
 */
public static String mostCommonWord(String text) {
    ...
}
```

- A user passes the whole text of Shakespeare's plays into your method and discovers that instead of returning a predictably common English word like "the" or "a", the method returns something unexpected, perhaps "e".
- Shakespeare's plays have 100,000 lines containing over 800,000 words, so this input would be very painful to debug by normal methods, like print-debugging and breakpoint-debugging.

# An example for controlling inputs

- **Debugging will be easier if you first work on reducing the size of the buggy input to something manageable that still exhibits the same (or very similar) bug:**
  - Does the first half of Shakespeare show the same bug? (Binary search! Always a good technique)
  - Does a single play have the same bug?
  - Does a single speech have the same bug?
- **Once you've found a small test case, find and fix the bug using that smaller test case, and then go back to the original buggy input and confirm that you fixed the same bug.**

# An example for controlling inputs

- Suppose a user reports that `mostCommonWord("chicken chicken chicken beef")` returns "beef" instead of "chicken".
- To shorten and simplify this input before you start debugging, which of the following inputs are worth trying?
  - `mostCommonWord("chicken chicken beef")`
  - `mostCommonWord("Chicken Chicken Chicken beef")`
  - `mostCommonWord("chicken beef")`
  - `mostCommonWord("a b c")`
  - `mostCommonWord("b b c")`

## (3) Controlling Inputs by Load and Stress

### ■ Load and Stress

- Some bugs manifest only when the software is under some kind of stress.
- Software itself is having to do (handle a large number of simultaneous requests, for example, or particularly large data sets).
- Something within the environment (high levels of general network traffic, say, or restricted free memory).

### ■ A load-testing tool executes a script that simulates a more-or-less realistic usage pattern.

- Stress-testing tools are similar, except they generate load indirectly (E.g., you might use one to allocate and deallocate lots of memory while your software is running, or to consume lots of CPU time.)
- QuickTest Professional and LoadRunner: <http://www.hp.com/>
- JMeter: <http://jakarta.apache.org/jmeter/>
- The Grinder: <http://grinder.sourceforge.net/>



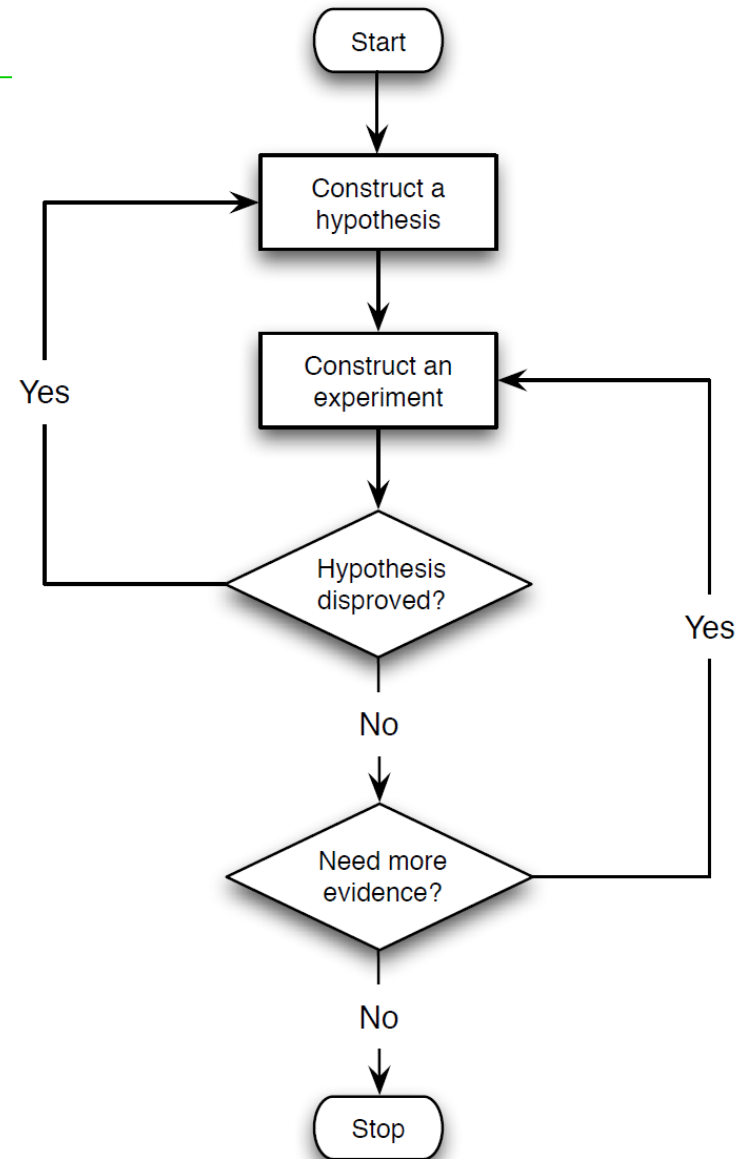


## (2) Diagnosing the bug



# Diagnose

- **Diagnose: Understand the Location and Cause of the Bug**



# Diagnose by experiments



## ■ Experiments Must Prove Something

- Experiments are a means to an end, not an end in themselves. There is no point performing an experiment unless it proves something.

## ■ One Change at a Time

- One of the basic rules of constructing experiments is that you should make only a single change at a time.
- This rule applies to any kind of change—changes to the source, the environment, input files, and so on. It applies to anything, in fact, that might have an effect on the software.

## ■ Keep a Record of What You've Tried

## ■ Ignore Nothing

# Diagnosis stratagem 1: Instrumentation



## ■ Instrumentation

- Instrumentation is code that doesn't affect how the software behaves but instead provides insight into why it behaves as it does.
- E.g., logging.
- Instrumentation isn't limited to simple output statements, however — you have the full facilities of the language at your disposal.
- You can collect and collate data, evaluate arbitrary code, and test for relevant conditions.

# Diagnosis stratagem 1: Instrumentation

```
while(node != null) {  
    node.process();  
    node = node.getNext();  
}
```



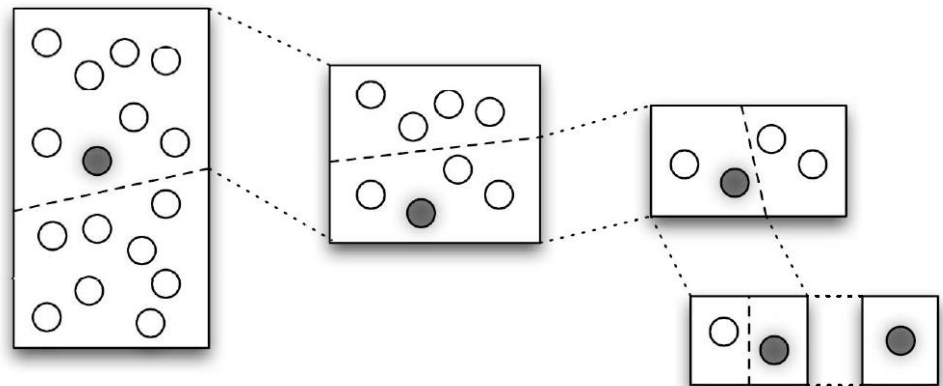
```
HashSet processed = new HashSet();
```

```
while(node != null) {  
    if(!processed.add(node)) {  
        System.out.println("The problem node is: " + node);  
    }  
    node.process();  
    node = node.getNext();  
}
```

Bug:  
getNext() is returning  
one or more nodes more  
than once. It's not  
clear which nodes are  
being processed more  
than once.

# Diagnosis stratagem 2: Divide and Conquer

- **Divide and conquer, or binary chop / search, is a search strategy.**
  - It is the Swiss Army knife of debugging—it crops up again and again in a wide variety of situations.
  - It can provide you with a quick and easy way to exclude a large number of candidates.
  - E.g., perhaps your software contains a number of modules that can be enabled and disabled independently? If so, try disabling them all and see whether the bug still occurs. If it does, then you've eliminated a lot of code that you won't need to examine (and won't confuse matters). If it doesn't, then you can quickly identify the problem module by enabling half and rerunning your test.



## Diagnosis stratagem 3: Leveraging VCS

- **Occasionally, a bug in functionality that used to work correctly but was broken by some subsequent change. To this kind of problem, there is one tool of particular value when regression hunting – source control system.**
  - The first step is to review check-in comments – it may be that the culprit is obvious.
  - If not, however, you can quickly pinpoint the change using the following procedure.
    - E.g., Imagine that you know that the bug wasn't present in version 2.3, but it is present in the current version, 3.0. In between 2.3 and 3.0 are 200 different check-ins.
    - You know the drill by now – check out and build the middle revision, and see whether the bug is present. If not, it was introduced by a more recent change; otherwise, it was one of the earlier ones. A few iterations later, and you know exactly which change it was. -- **binary chop**
- **Git source control system provides direct support for it in the form of the `git bisect` command ("**Wolf fence**")**

## Diagnosis stratagem 4: Focus on the differences

- Your software normally works. So, the feature affected by the bug you're trying to diagnose probably works correctly in almost all situations or for almost everyone else.
- So, what you're looking for is something that makes this particular situation or customer special.
- Often these differences come to light when trying to reproduce the problem.
- Does it happen in only one particular environment? In that case, the problem is most likely in environment-specific code.
  - E.g., Does it happen only with large input files? Most likely you're looking for a resource leak or a limit being exceeded.



## Diagnosis stratagem 5: Learn from others

- Sometimes the bug will relate to a widely used technology (your compiler, for example, or a library or framework you're using) in which case there's a chance that someone else has run afoul of the same problem before you.
- In such instances, a little research on the Web can play dividends. Perhaps someone has asked a question about the same kind of failure module on a forum or has written a blog post describing the pitfall they fell into, which turns out to be exactly the one you find yourself in.

# Diagnosis stratagem 6: Debugger

## ■ Debugger

- Debuggers vary dramatically in both sophistication and capabilities, from simple command line-oriented examples to those that are fully integrated into a graphical IDE.
  - What they all have in common is that they allow us to examine the code as it executes, setting breakpoints, single-stepping, and examining program state.
- ## ■ It's particularly helpful at three different points of the development life cycle:
- During initial development, it's helpful when single-stepping through code helps to convince us that what it's really doing agrees with what we thought we were implementing.
  - If we have a theory about why the code is behaving in a particular way, we can use the debugger to confirm or refute this theory.
  - Finally, a debugger helps us explore code that is behaving in a way we simply don't understand.



## (3) Fix the bug



# Fix: Start from a clean source tree

- **Start from a clean source tree**
  - Before diving in and starting to design your fix, the first order of business is to ensure that you start from a clean source tree.
- **Work out how you're going to test your fix before making changes.**
- **Use testing to ensure that you're working on a clean source tree and fixing rightly.**
  - 1. Run the existing tests, and demonstrate that they pass.
  - 2. Add one or more new tests, or fix the existing tests, to demonstrate the bug (in other words, to fail).
  - 3. Fix the bug.
  - 4. Demonstrate that your fix works (the failing tests no longer fail).
  - 5. Demonstrate that you haven't introduced any regressions (none of the tests that previously passed now fail).



## (4) Reflection



# Reflect: How to ensure It'll Never Happen Again



## ■ Requirements:

- Were the requirements complete and correct? Perhaps they were ambiguous, interpreted incorrectly, or misunderstood?

## ■ Architecture or design:

- Was there an oversight within the architecture or design—something we failed to take into account or allow for? Or perhaps they're fine, but we failed to follow the design correctly?

## ■ Testing:

- Did we have adequate tests covering this area? Or maybe the error was in the tests themselves?

## ■ Construction:

- Perhaps the author made a simple mistake when writing the code, or maybe they misunderstood some aspect of the underlying technology (libraries, compilers, and so forth).



## 3 Debugging tools



# Debugging by Brute Force

- **The most common scheme for debugging a program is the “brute force” method.**
  - It is popular because it requires little thought and is the least mentally taxing of the methods, but it is inefficient and generally unsuccessful.
- **Brute force methods can be partitioned into at least three categories:**
  - Debugging with a memory dump.
  - Debugging according to the common suggestion to “scatter print statements throughout your program.”
  - Debugging with automated debugging tools.





# (1) Post-mortem debugging: memory dump

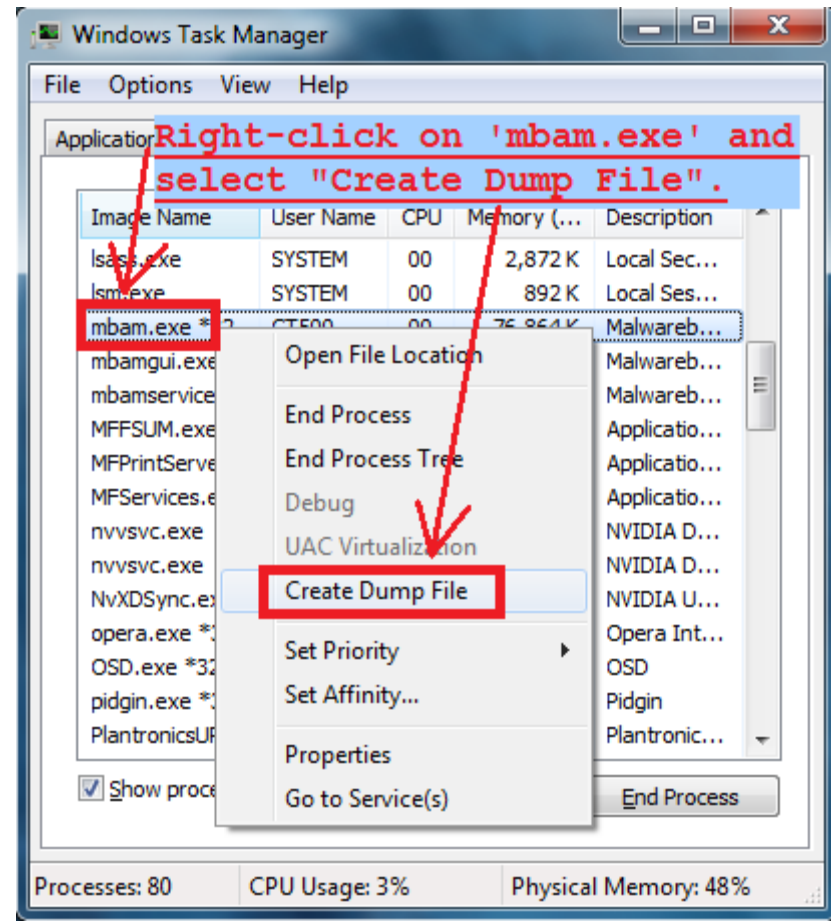


# Post-mortem debugging

- **Post-mortem debugging is debugging of the program after it has already crashed.**
- **Related techniques often include various tracing techniques and/or analysis of memory dump (or core dump) of the crashed process.**
  - The dump of the process could be obtained automatically by the system (for example, when process has terminated due to an unhandled exception), or by a programmer-inserted instruction, or manually by the interactive user.

# Memory dump

- **Memory dump:** a file on hard disk containing a copy of the contents of a process's memory of a specific time, produced when a process is aborted by certain kinds of internal error or signal.
- When a program aborts, a memory dump can be taken in order to examine the status of the program at the time of the crash.





## (2) Post-mortem debugging: stack trace



# Stack trace based debugging

- A stack trace is a listing of all pending method calls at a particular point in the execution of a program.
- You have almost certainly seen stack trace listings – they are displayed whenever a Java program terminates with an uncaught exception.

```
java.lang.NullPointerException
```

```
    at MyClass.mash(MyClass.java:9)
```

```
    at MyClass.crunch(MyClass.java:6)
```

```
    at MyClass.main(MyClass.java:3)
```

# Stack trace based debugging

- You can access the text description of a stack trace by calling the `printStackTrace` method of the `Throwable` class.

```
Throwable t = new Throwable();  
StringWriter out = new StringWriter();  
t.printStackTrace(new PrintWriter(out));  
String description = out.toString();
```

# Stack trace based debugging

- A more flexible approach is the `getStackTrace` method that yields an array of `StackTraceElement` objects, which you can analyze in your program.

```
public class WhoCalled {
    static void f() {
        // Generate an exception to fill in the stack trace
        try {
            throw new Exception();
        } catch (Exception e) {
            for(StackTraceElement ste : e.getStackTrace())
                System.out.println(ste.getMethodName());
        }
    }
    static void g() { f(); }
    static void h() { g(); }
    public static void main(String[] args) {
        f();
        System.out.println("-----");
        g();
        System.out.println("-----");
        h();
    }
}
```

# Stack trace based debugging

```
public class WhoCalled {
    static void f() {
        // Generate an exception to fill in the stack trace
        try {
            throw new Exception();
        } catch (Exception e) {
            for(StackTraceElement ste : e.getStackTrace())
                System.out.println(ste.getMethodName());
        }
    }
    static void g() { f(); }
    static void h() { g(); }
    public static void main(String[] args) {
        f();
        System.out.println("-----");
        g();
        System.out.println("-----");
        h();
    }
}
```

## Output:

```
f
main
-----
f
g
main
-----
f
g
h
main
```

The `StackTraceElement` class has methods to obtain the file name and line number, as well as the class and method name, of the executing line of code. The `toString` method yields a formatted string containing all of this information.





## (3) Printf debugging



# Print debugging/tracing

- Print debugging (or tracing) is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process. This is sometimes called `printf` debugging, due to the use of the `printf` function in C.
- It is to add trace code to the program to print out values of variables as the program executes, using `printf()` or `System.out.println()` statements, for example. It's Simple but effective.

```
System.out.println("Entering callMethod");  
result = callMethod();  
System.out.println("The result is " + result);
```

# A tip for print debugging

- Once the cause of trouble is figured out or the program will be released, the printing statements for debugging should be removed or disabled.

- **Method1- comment the print code**

```
//System.out.println("Entering callMethod");  
result = callMethod();  
//System.out.println("The result is " + result);
```

- **Method2**

```
public static void MyPrint(String in) {  
    System.out.println(in);  
}
```

```
MyPrint("Entering callMethod");  
result = callMethod();  
MyPrint("The result is " + result);
```

**Coding & debugging**

```
public static void MyPrint(String in) {  
    //System.out.println(in);  
}
```

```
MyPrint("Entering callMethod");  
result = callMethod();  
MyPrint("The result is " + result);
```

**released**



# (4) Logging



# Logging

- At its simplest level, `System.out.println()` or similar throughout the code is enough.
- If your logging requirements are at all complex, however, you should consider using one of the many logging frameworks.
- Logging frameworks provide the ability for your code to contain configurable logging that can be enabled, disabled, or increased in detail, typically at runtime and by individual feature.

# Logging frameworks

- `java.util.logging`
  - <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/>
  - As of 1.4.2, Java includes a standard logging API `java.util.logging`, commonly known as JUL.
- Log4j
  - <http://logging.apache.org/log4j/>
  - Apache log4j is probably the best-known Java logging library, and ports exist to most major languages.
- Logback: <http://logback.qos.ch/>
- SLF4J: <http://www.slf4j.org/>
- syslog-ng: <http://www.balabit.com/network-security/syslog-ng/>

# java.util.logging

## ■ java.util.logging API:

- It is easy to suppress all log records or just those below a certain level, and just as easy to turn them back on.
- Suppressed logs are very cheap, so that there is only a minimal penalty for leaving the logging code in your application.
- Log records can be directed to different handlers—for displaying in the console, writing to a file, and so on.
- Both loggers and handlers can filter records. Filters can discard boring log entries, using any criteria supplied by the filter implementor.
- Log records can be formatted in different ways—for example, in plain text or XML.
- Applications can use multiple loggers, with hierarchical names such as `com.mycompany.myapp`, similar to package names.
- By default, the logging configuration is controlled by a configuration file.
- Applications can replace this mechanism if desired.

# java.util.logging

## ■ Basic Logging

- For simple logging, use the global logger and call its info method:

```
import java.util.logging.*;
```

```
Logger.getLogger().info("File->Open menu item selected");
```

**Results:**

```
May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen
```

```
INFO: File->Open menu item selected
```

- You can call the below code at an appropriate place (such as the beginning of main), then all logging is suppressed

```
Logger.getLogger().setLevel(Level.OFF);
```



# java.util.logging

## ■ Advanced Logging -- define your own logger

- In a professional application, you wouldn't want to log all records to a single global logger. Instead, you can define your own loggers.
- Call the getLogger method to create or retrieve a logger:

```
import java.util.logging.*;
```

```
private static final Logger myLogger = Logger.getLogger("com.mycompany.myapp");  
//often using class name as logger name
```

Or

```
public class LogTest {  
    static String strClassName = LogTest.class.getName(); //get class name  
    static Logger myLogger = Logger.getLogger(strClassName);  
    // using class name as logger name  
    .....  
    myLogger.info("  XXXX  ");  
}
```

# java.util.logging

- **There are seven logging levels for logger:**
  - SEVERE
  - WARNING      By default, the top three levels are actually logged.
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST
- **Set logging level `setLevel()`:**      E.g., `logger.setLevel(Level.FINE);`
  - Now FINE and all levels above it are logged.
  - You can also use `Level.ALL` parameter to turn on logging for all levels
  - `Level.OFF` parameter to turn all logging off.

# java.util.logging

```
import java.util.logging.Logger;

public class LevelTest {
    private static String name = HelloLogWorld.class.getName();
    private static Logger log = Logger.getLogger(name);

    public void sub(){
        log.severe("severe level");
        log.warning("warning level");
        log.info("info level");
        log.config("config level");
        log.fine("fine level");
        log.finer("finer level");
        log.finest("finest level");
    }

    public static void main(String[] args){
        LevelTest test = new LevelTest();
        test.sub();
    }
}
```

# java.util.logging

## ■ Logging Handlers

- By default, loggers send records to a `ConsoleHandler`(which control what to show in console) that prints them to the `System.err` stream.
- **Like loggers, handlers have a logging level (also the 7 levels). For a record to be logged, its logging level must be above the threshold of both the logger and the handler.**
- **The log manager configuration file sets the logging level of the default console handler as**
  - `java.util.logging.ConsoleHandler.level=INFO`

# java.util.logging

- Alternatively, you can bypass the configuration file altogether and install your own handler.

```
import java.util.logging.*;
public class LevelTest {
    private static String name = test.class.getName();
    private static Logger log = Logger.getLogger(name);

    public void sub() {
        log.setLevel(Level.FINEST);
        log.setUseParentHandlers(false);
        Handler handler = new ConsoleHandler();
        handler.setLevel(Level.FINEST);
        log.addHandler(handler);

        log.severe("severe level");
        log.warning("warning level");
        log.info("info level");
        log.config("config level");
        log.fine("fine level");
        log.finer("finer level");
        log.finest("finest level");
    }
}

public static void main(String[] args) {
    LevelTest test = new LevelTest();
    test.sub();
}
```

# java.util.logging

- To send log records elsewhere, add another handler. The logging API provides two useful handlers for this purpose: a `FileHandler` and a `SocketHandler`.
- The `SocketHandler` sends records to a specified host and port.
- Of greater interest is the `FileHandler` that collects records in a file.
  - By default, the records are formatted in XML.

```
FileHandler handler = new FileHandler();  
logger.addHandler(handler);
```

# Log4j

- **Apache Log4j is a Java-based logging utility.**
  - <http://logging.apache.org/log4j/>
  - Apache log4j is probably the best-known Java logging library, and ports exist to most major languages.

Framework	Supported log levels	Standard appenders	Popularity	Cost / license
<b>Log4J</b>	FATAL ERROR WARN INFO DEBUG TRACE	AsyncAppender, JDBCAppender, JMSAppender, LF5Appender, NTEventLogAppender, NullAppender, SMTPAppender, SocketAppender, SocketHubAppender, SyslogAppender, TelnetAppender, WriterAppender	Widely used in many projects and platforms	Apache License, Version 2.0
<b>Java Logging API</b>	SEVERE WARNING INFO CONFIG FINE FINER FINEST	Sun's default Java Virtual Machine (JVM) has the following: ConsoleHandler, FileHandler, SocketHandler, MemoryHandler		Comes with the JRE



## (5) Compiler Warning Messages





# Compiler Warning Messages

- One of the simplest and most effective debugging tools is your own compiler.
- Set your compiler's warning level to the highest, pickiest level possible and fix the code so that it doesn't produce any compiler warnings.
  - Assume that the people who wrote the compiler know a great deal more about your language than you do. If they're warning you about something, it usually means you have an opportunity to learn something new about your language. Make the effort to understand what the warning really means.



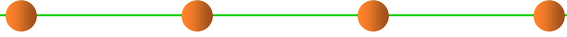
## (6) Debugger: breakpoints, etc



# Debugger

- **Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, restart it, set breakpoints, and change values in memory.**
  
- **Main Debugger Operations**
  - Stepping Through the Source Code
    - Breakpoints
    - Single-stepping
    - Resume operation
    - Temporary breakpoints
  - Inspecting Variables
  - Issuing an “All Points Bulletin” for Changes to a Variable
    - A watchpoint combines the notions of breakpoint and variable inspection.
  - Moving Up and Down the Call Stack

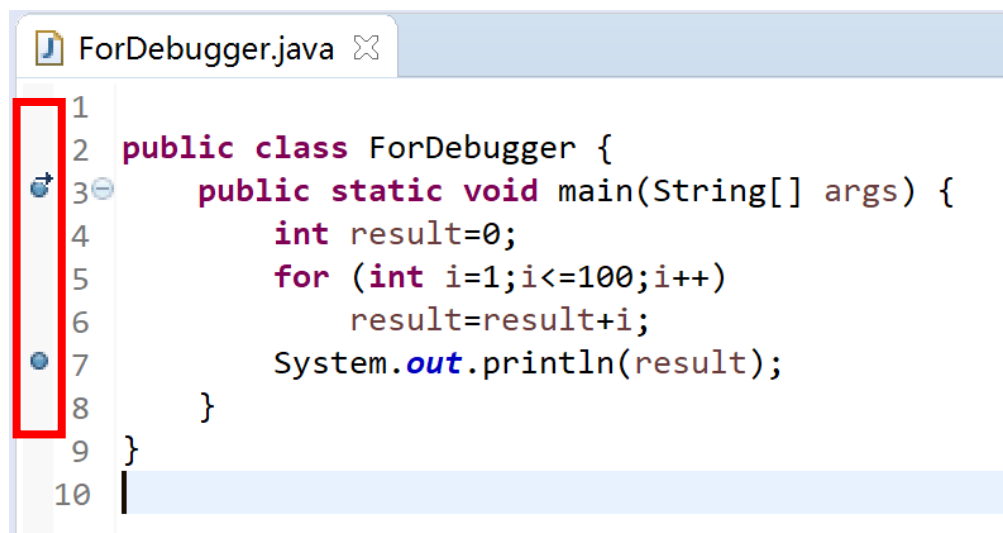
# Debugger: breakpoint

- 
- In software development, a breakpoint is an intentional stopping or pausing place in a program, put in place for debugging purposes. It is also sometimes simply referred to as a pause.
  - A breakpoint is like a tripwire within a program: You set a breakpoint at a particular “place” within your program, and when execution reaches that point, the debugger will pause the program’s execution, and then the programmer can examine the current state of the program.

# Debugger: breakpoint

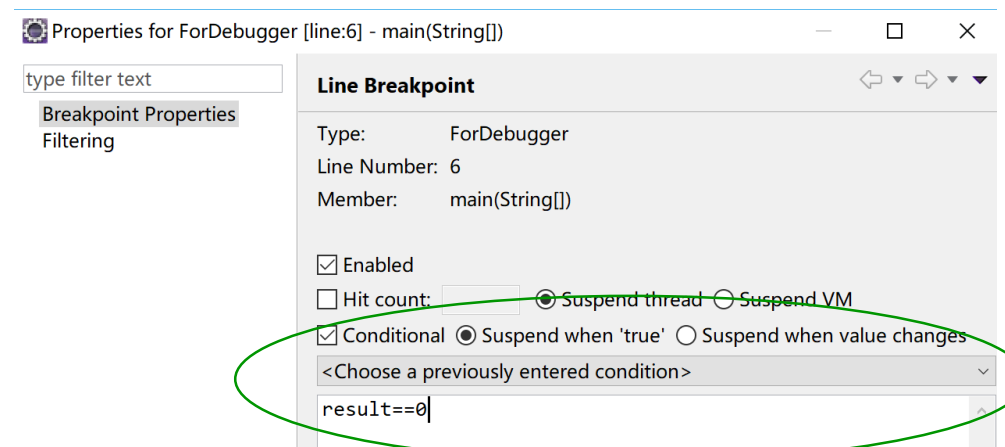
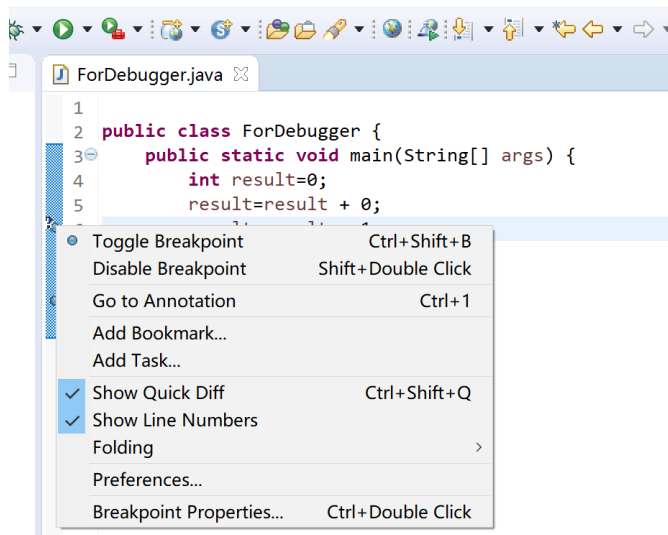
## ■ Setting Breakpoints in Eclipse

- To set a breakpoint at a given line in Eclipse, double-click on that line.
- To delete a breakpoint at a given line in Eclipse, double-click the breakpoint symbol on that line.
- To disable a breakpoint in Eclipse by right-clicking the breakpoint symbol in the line in question. A menu will pop up. Note that the Toggle option means to delete the break-point, while Disable/Enable means the obvious.



# Debugger: watchpoint

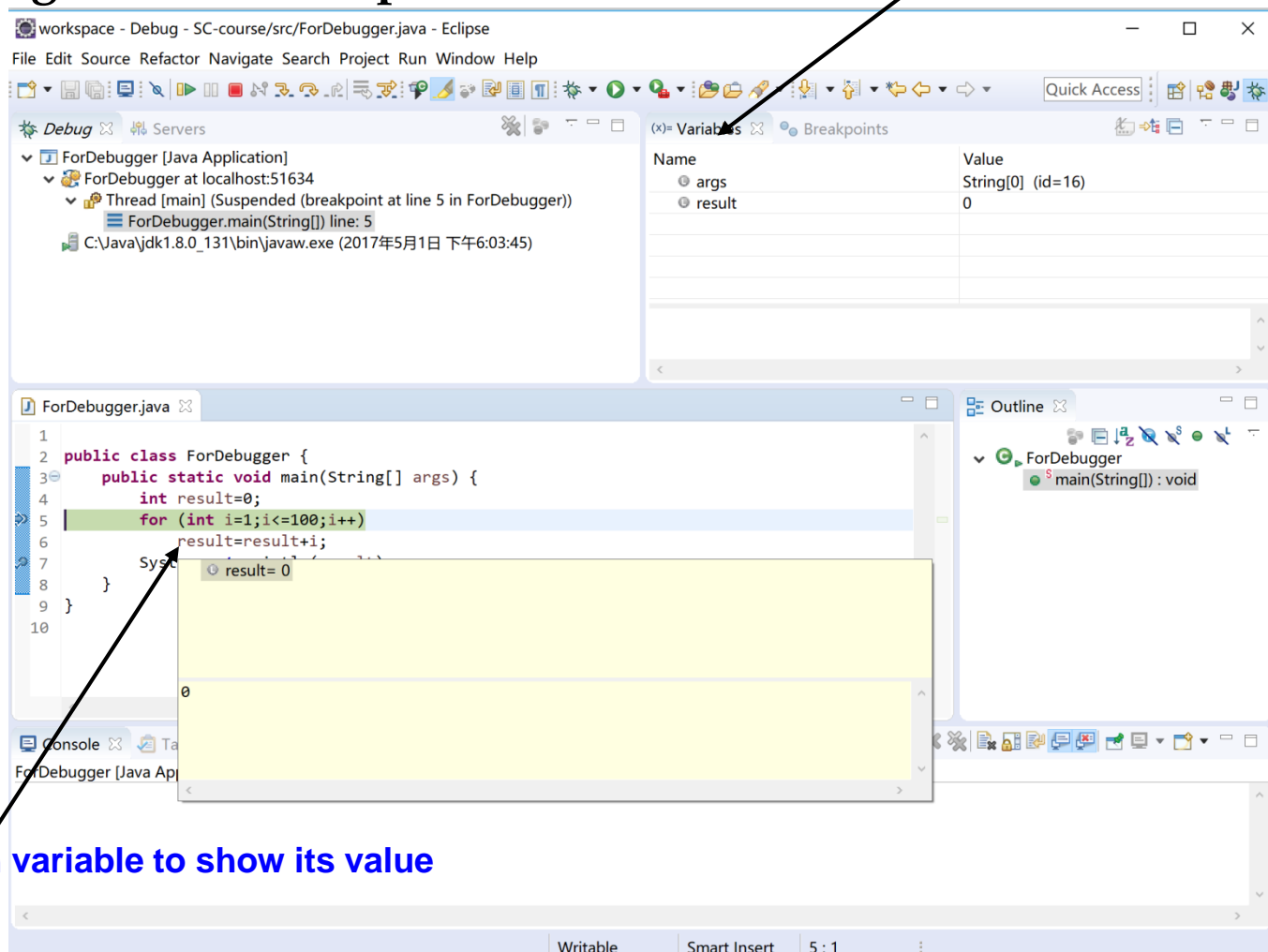
- A watchpoint combines the notions of breakpoint and variable inspection.
- The most basic form instructs the debugger to pause execution of the program whenever the value of a specified variable changes.
- Conditional breakpoints in Eclipse:



# Debugger: inspecting variables

## ■ Inspecting values in Eclipse

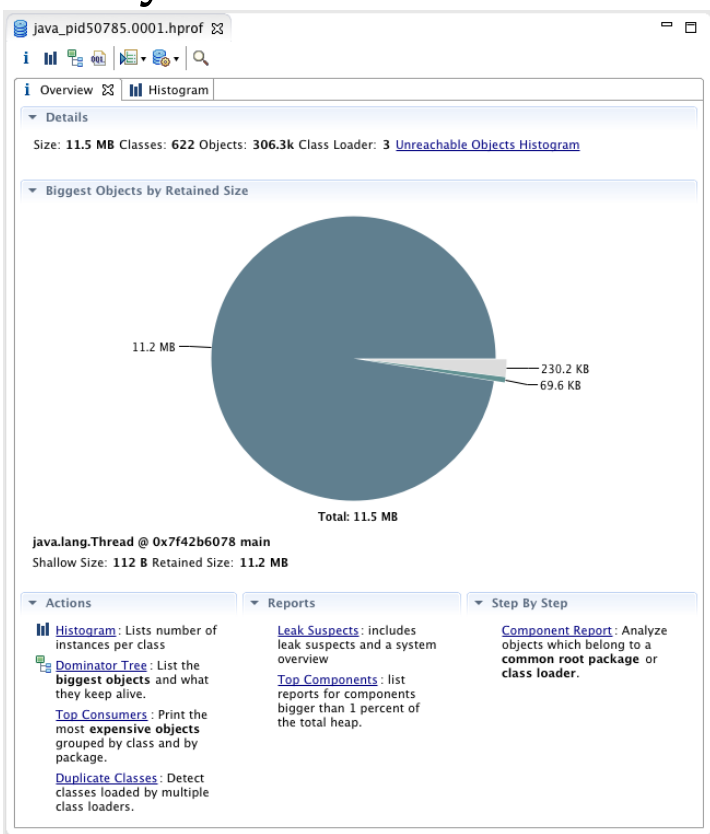
show all variable values



Put mouse on variable to show its value

# Debugger: Eclipse Memory Analyzer (MAT)

- The Eclipse Memory Analyzer (MAT) can help provide details of an application's memory use. The tool is useful for both tracking memory leaks and for periodically reviewing the state of your system.



The screenshot shows the Eclipse Memory Analyzer (MAT) Overview tab for a Java process (java\_pid50785.0001.hprof). The interface includes a toolbar, tabs for Overview and Histogram, and a table of memory usage.

Class Name	Objects	Shallow Heap	Retained Heap
<b>com.example.mat.*</b>	<Numeric>	<Numeric>	<Numeric>
com.example.mat.Listener	100,000	1,600,000	
com.example.mat.Controller	1	24	
com.example.mat.Allocator	1	24	
com.example.mat.List2	1	16	
com.example.mat.List1	1	16	
com.example.mat.Main	0	0	
com.example.mat.Main\$1	0	0	
<b>Total: 7 entries (615 filtered)</b>	<b>100,004</b>	<b>1,600,080</b>	

第8章有介绍

[eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/)





The end

April 17, 2019