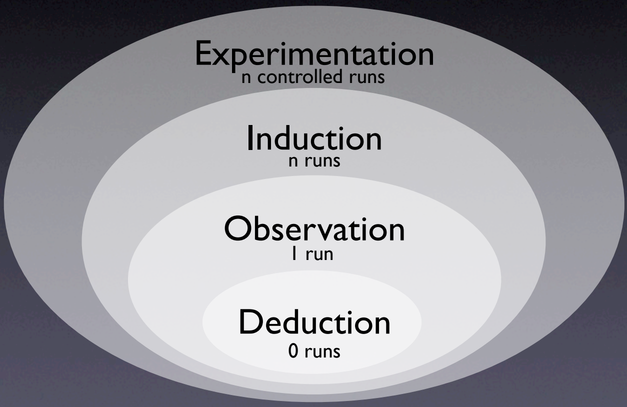
Observing Facts

Andreas Zeller

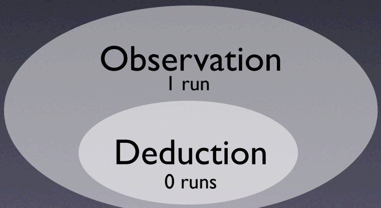
1

Reasoning about Runs

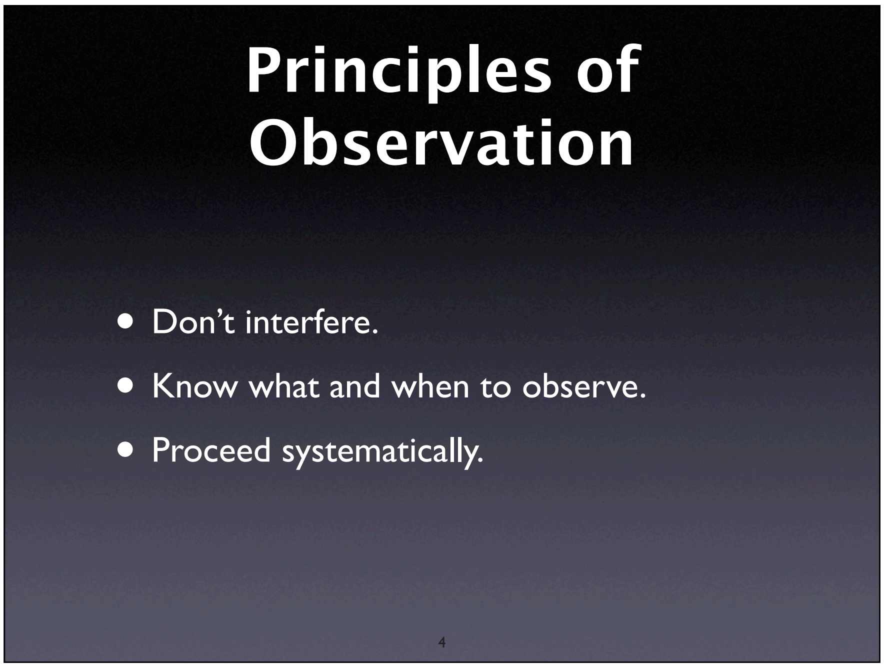


2

Reasoning about Runs



3

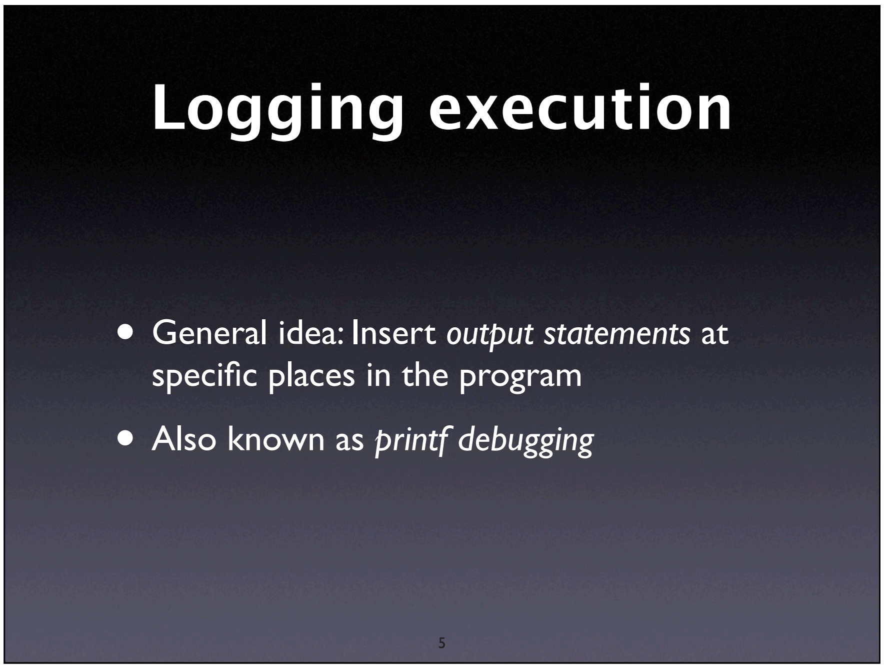


Principles of Observation

- Don’t interfere.
- Know what and when to observe.
- Proceed systematically.

4

4



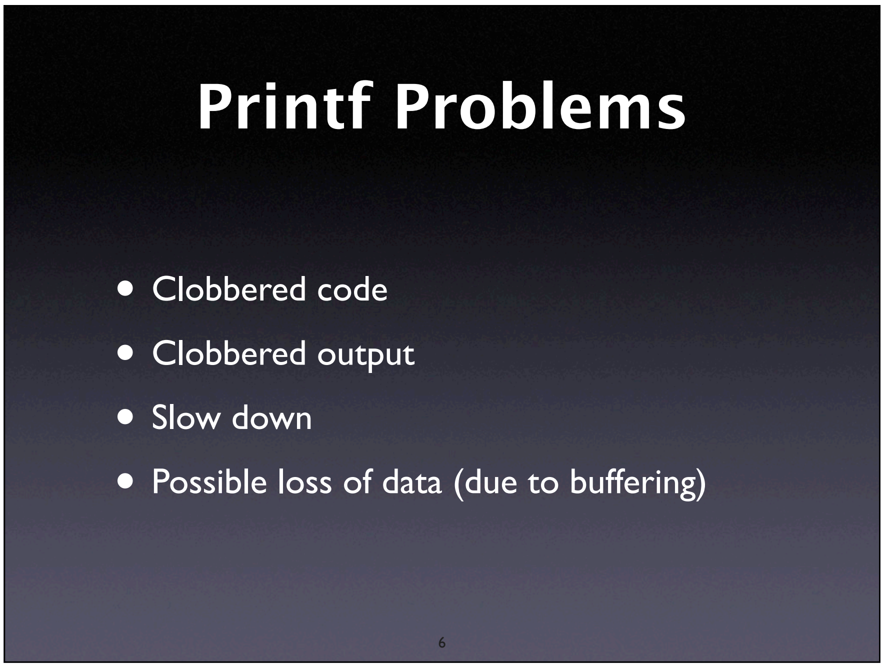
Logging execution

- General idea: Insert *output statements* at specific places in the program
- Also known as *printf debugging*

5

5

Demonstrate technique, using sample program

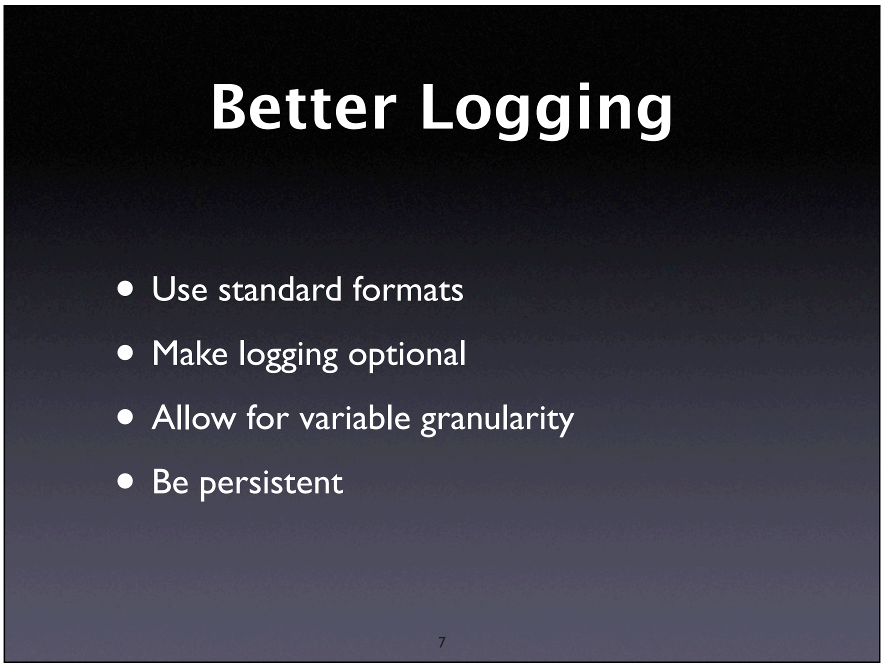


Printf Problems

- Clobbered code
- Clobbered output
- Slow down
- Possible loss of data (due to buffering)

6

6

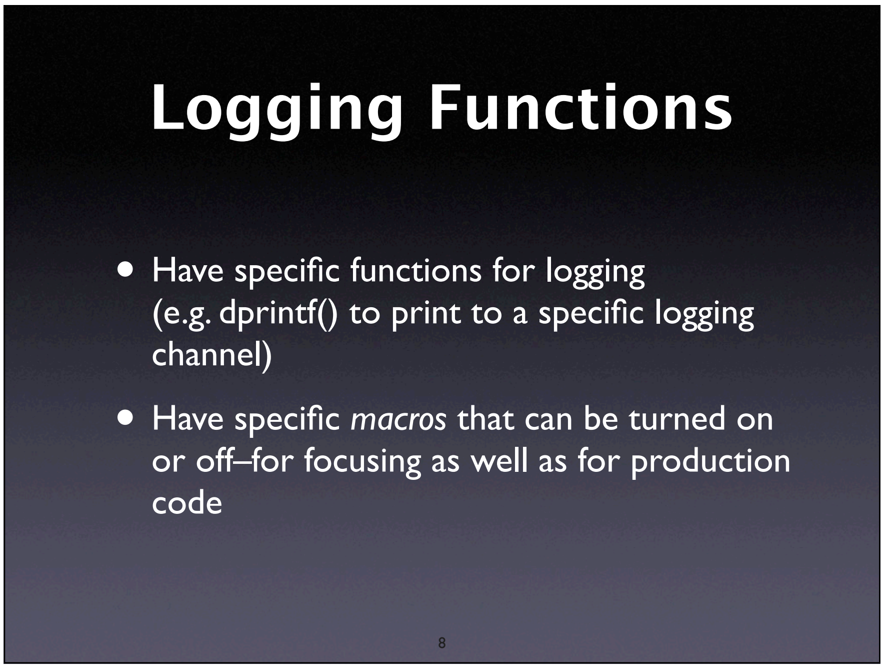


Better Logging

- Use standard formats
- Make logging optional
- Allow for variable granularity
- Be persistent

7

7



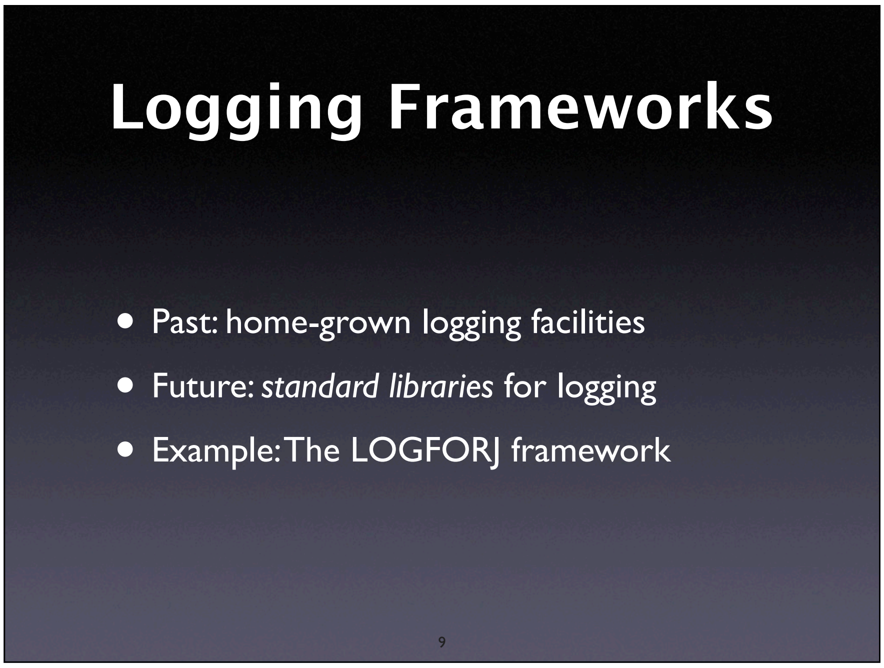
Logging Functions

- Have specific functions for logging (e.g. dprintf() to print to a specific logging channel)
- Have specific *macros* that can be turned on or off–for focusing as well as for production code

8

Again, demonstrate the use of LOG() interactively

8



Logging Frameworks

- Past: home-grown logging facilities
- Future: *standard libraries* for logging
- Example: The LOGFORJ framework

9

9

LOGFORJ

```
// Initialize a logger.
final ULogger logger =
    LoggerFactory.getLogger(TestLogging.class);

// Try a few logging methods
public static void main(String args[]) {
    logger.debug("Start of main()");
    logger.info ("A log message with level set to INFO");
    logger.warn ("A log message with level set to WARN");
    logger.error("A log message with level set to ERROR");
    logger.fatal("A log message with level set to FATAL");

    new TestLogging().init();
}
```

10

The core idea of LOGFORJ is to assign each class in an application an individual or common logger. A logger is a component which
takes a request for logging and logs it. Each logger has a
level, from DEBUG over INFO, WARN, and ERROR to FATAL (very important messages).

10

Customizing Logs

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=DEBUG, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=\
%d [%t] %-5p %c %x - %m%n
```

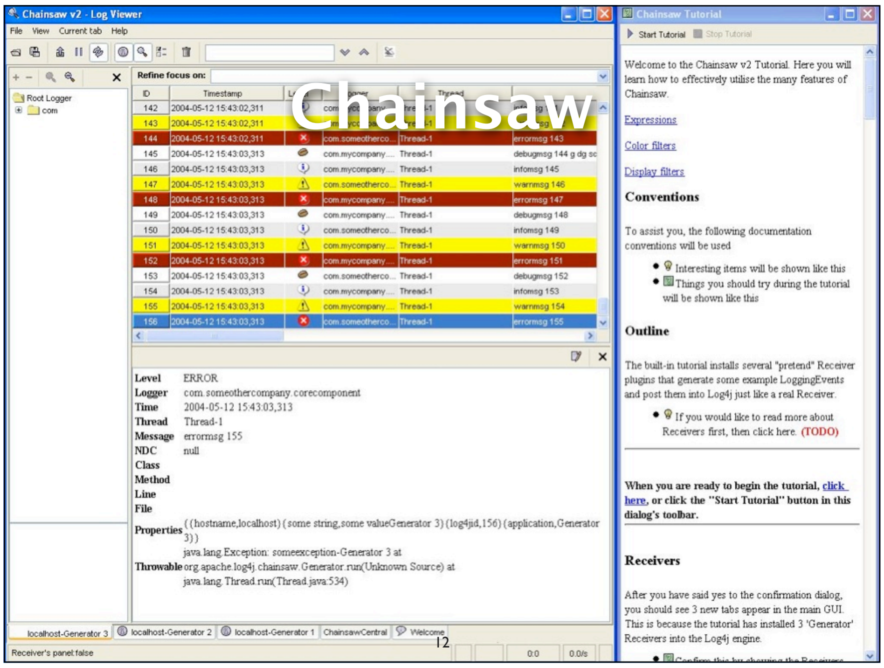
```
2005-02-06 20:47:31,508 [main] DEBUG  TestLogging - Start of
main()
2005-02-06 20:47:31,529 [main] INFO   TestLogging - A log
message with level set to INFO
```

11

The core idea of LOGFORJ is to assign each class in an application an individual or common logger. A logger is a component which
takes a request for logging and logs it. Each logger has a
level, from DEBUG over INFO, WARN, and ERROR to FATAL (very important messages).

11

Chainsaw



ID	Timestamp	L[illegible]	[illegible]	Thread	
142	2004-05-12 15:43:02,311	ⓘ	com[illegible]	[illegible]	[illegible]
143	2004-05-12 15:43:02,311	[illegible]	[illegible]	[illegible]	[illegible]
144	2004-05-12 15:43:02,311	✖	com.someotherco...	Thread-1	errormsg 143
145	2004-05-12 15:43:03,313	●	com.mycompany....	Thread-1	debugmsg 144 g dg sc
146	2004-05-12 15:43:03,313	ⓘ	com.mycompany....	Thread-1	infomsg 145
147	2004-05-12 15:43:03,313	⚠	com.someotherco...	Thread-1	warnmsg 146
148	2004-05-12 15:43:03,313	✖	com.mycompany....	Thread-1	errormsg 147
149	2004-05-12 15:43:03,313	●	com.mycompany....	Thread-1	debugmsg 148
150	2004-05-12 15:43:03,313	ⓘ	com.someotherco...	Thread-1	infomsg 149
151	2004-05-12 15:43:03,313	⚠	com.mycompany....	Thread-1	warnmsg 150
152	2004-05-12 15:43:03,313	✖	com.mycompany....	Thread-1	errormsg 151
153	2004-05-12 15:43:03,313	●	com.someotherco...	Thread-1	debugmsg 152
154	2004-05-12 15:43:03,313	ⓘ	com.mycompany....	Thread-1	infomsg 153
155	2004-05-12 15:43:03,313	⚠	com.mycompany....	Thread-1	warnmsg 154
156	2004-05-12 15:43:03,313	✖	com.someotherco...	Thread-1	errormsg 155

Level	ERROR
Logger	com.someothercompany.corecomponent
Time	2004-05-12 15:43:03,313
Thread	Thread-1
Message	errormsg 155
NDC	null
Class	
Method	
Line	
File	
Properties	{{hostname,localhost}{some string,some valueGenerator 3}{log4jid,156}{application,Generator 3}}
Throwable	java.lang.Exception: someexception-Generator 3 at org.apache.log4j.chainsaw.Generator.run(Unknown Source) at java.lang.Thread.run(Thread.java:534)

Welcome to the Chainsaw v2 Tutorial. Here you will learn how to effectively utilise the many features of Chainsaw.

Expressions

Color filters

Display filters

Conventions

To assist you, the following documentation conventions will be used

- Interesting items will be shown like this
- Things you should try during the tutorial will be shown like this

Outline

The built-in tutorial installs several "pretend" Receiver plugins that generate some example LoggingEvents and post them into Log4j just like a real Receiver.

- If you would like to read more about Receivers first, then click here. (TODO)

When you are ready to begin the tutorial, click here, or click the "Start Tutorial" button in this dialog's toolbar.

Receivers

After you have said yes to the confirmation dialog, you should see 3 new tabs appear in the main GUI. This is because the tutorial has installed 3 'Generator' Receivers into the Log4j engine.

12

12

Logging with Aspects

- Basic idea: Separate concerns into individual syntactic entities *(aspects)*
- Aspect code *(advice)* is *woven* into the program code at specific places *(join points)*
- The same aspect code can be woven into multiple places *(pointcuts)*

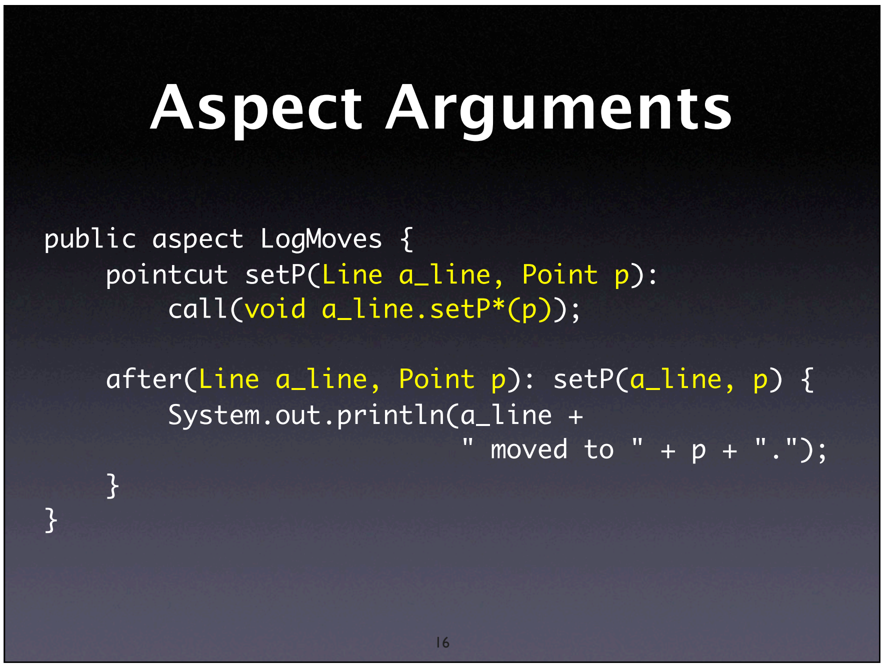
A Logging Aspect

```
public aspect LogBuy {
    pointcut buyMethod():
        call(public void Article.buy());
    before(): buyMethod() {
      System.out.println("Entering Article.buy()")
    }
    after(): buyMethod() {
      System.out.println("Leaving Article.buy()")
    }
}
```

```
$ ajc logBuy.aj Article.java
$ java Article
```

Using Pointcuts

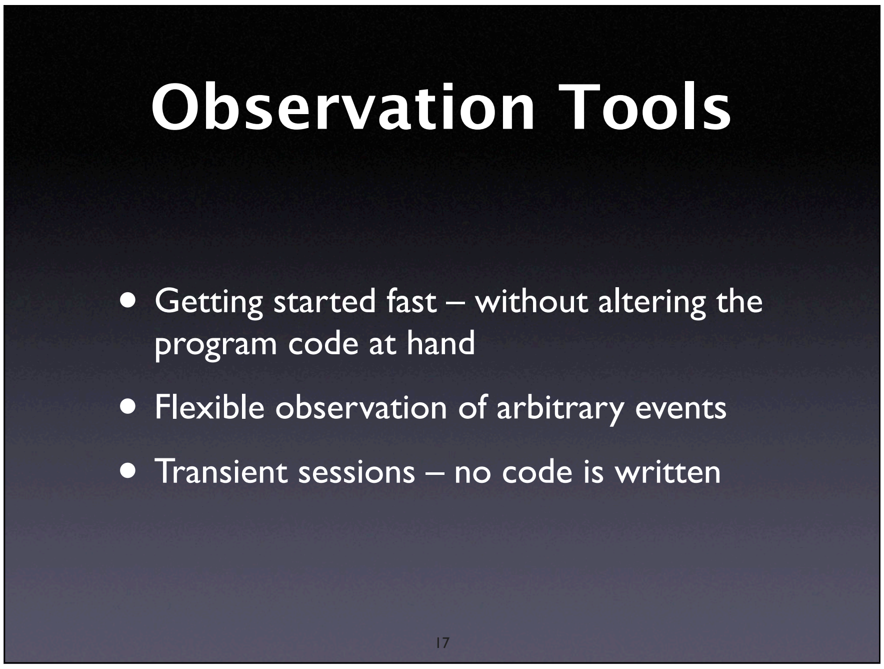
```
public aspect LogArticle {
  pointcut allMethods():
    call(public * Article.*(..));
  before(): allMethods() {
    System.out.println("Entering " + thisJoinPoint)
  }
  after(): allMethods() {
    System.out.println("Leaving " + thisJoinPoint)
  }
}
```



Aspect Arguments

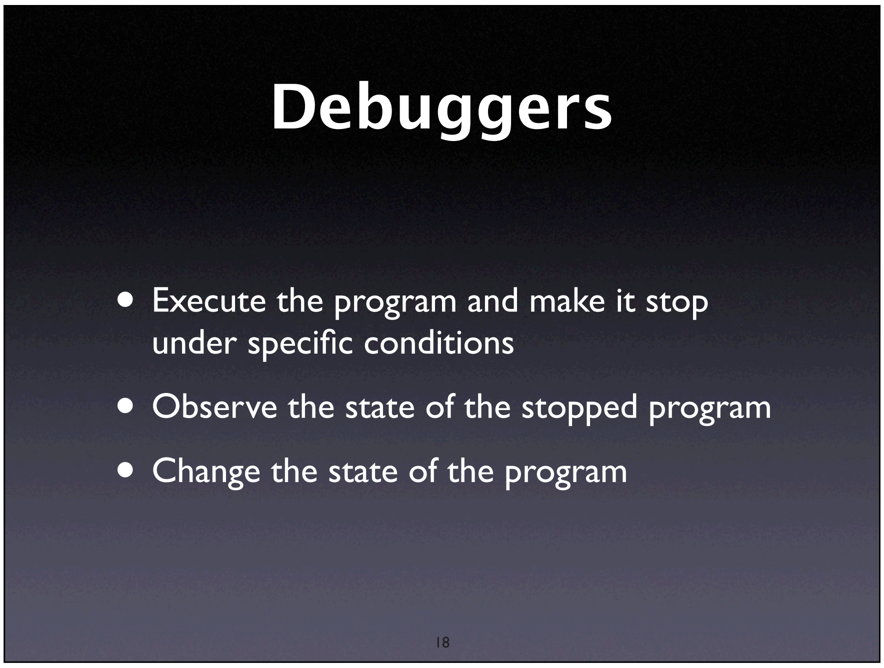
```
public aspect LogMoves {
    pointcut setP(Line a_line, Point p):
        call(void a_line.setP*(p));

    after(Line a_line, Point p): setP(a_line, p) {
        System.out.println(a_line +
                           " moved to " + p + ".");
    }
}
```



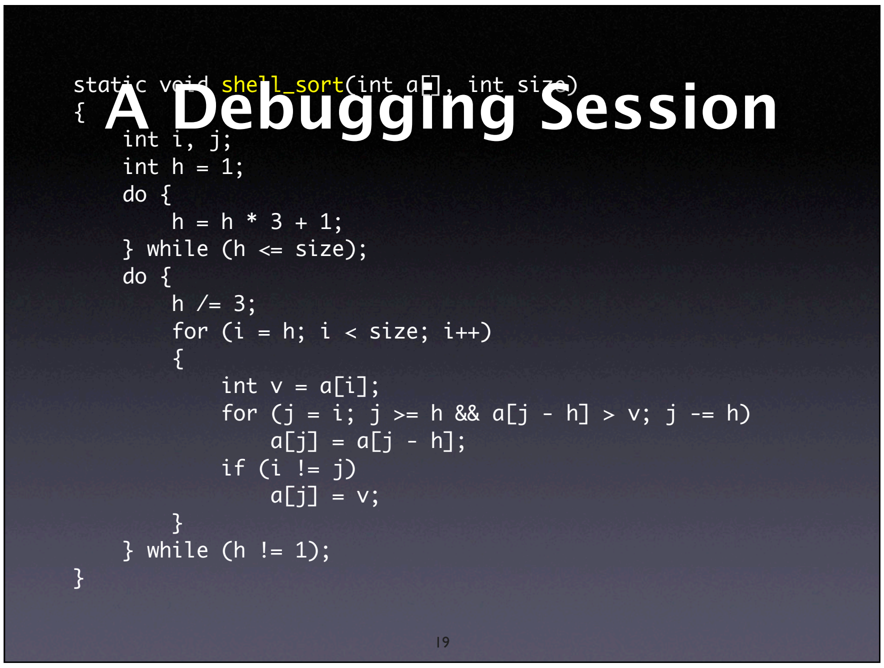
Observation Tools

- Getting started fast – without altering the program code at hand
- Flexible observation of arbitrary events
- Transient sessions – no code is written



Debuggers

- Execute the program and make it stop under specific conditions
- Observe the state of the stopped program
- Change the state of the program



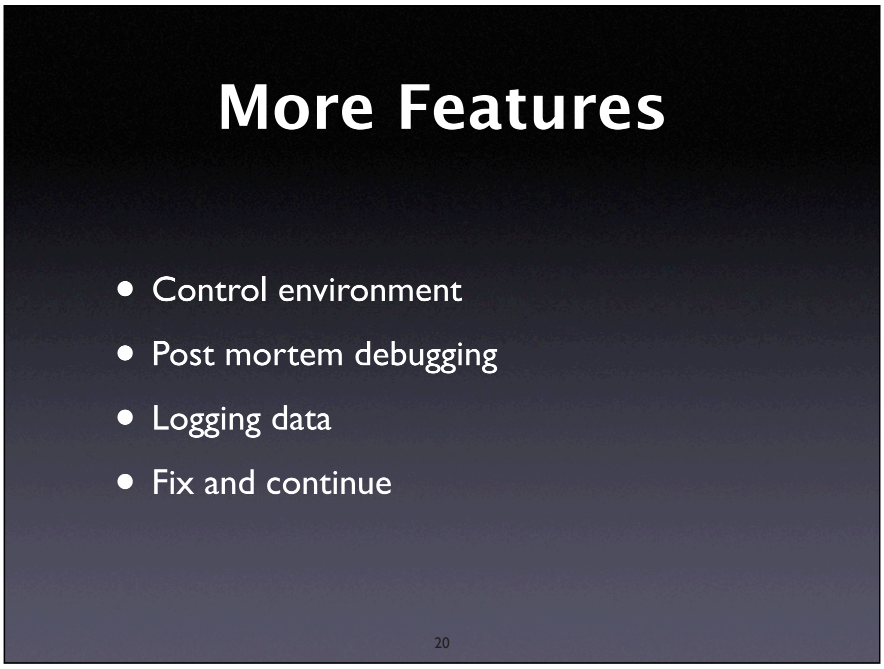
A Debugging Session

```
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```

19

Show this interactively with GDB or DDD

19

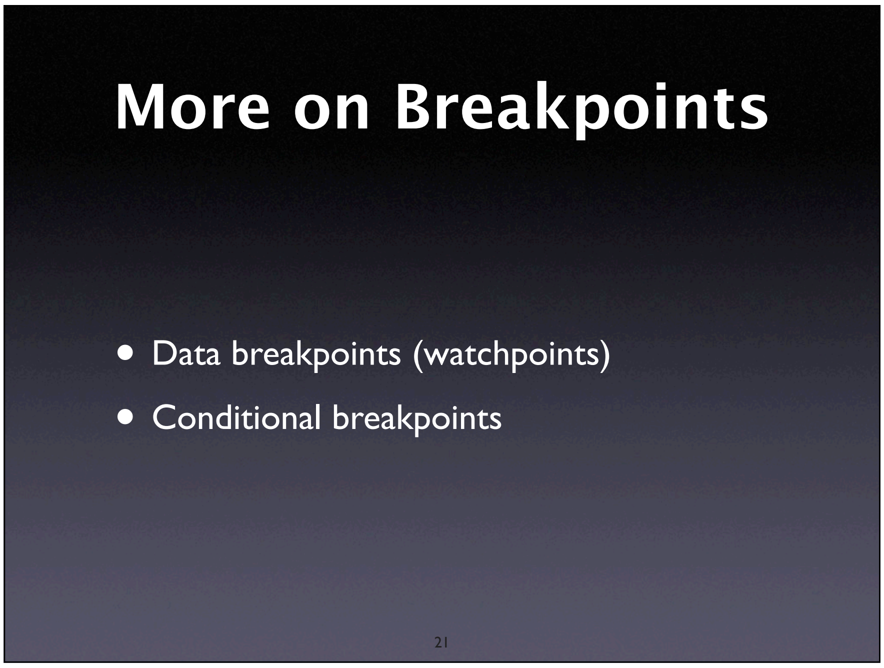


More Features

- Control environment
- Post mortem debugging
- Logging data
- Fix and continue

20

20



More on Breakpoints

- Data breakpoints (watchpoints)
- Conditional breakpoints

21

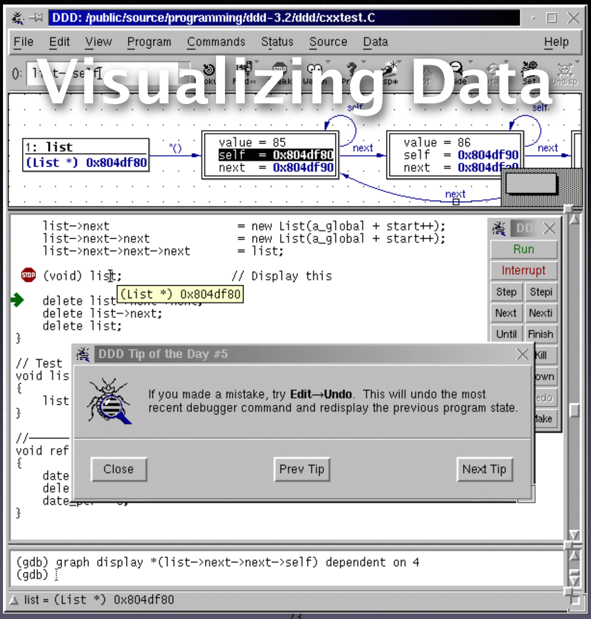
Demonstrate watchpoints and conditionals interactively

21

Debugger Caveats

- A debugger is a tool, not a toy!

Visualizing Data



```
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    (void) list;               // Display this

    delete list->next->next;
    delete list->next;
    delete list;
}
```

```
value = 85
self  = 0x804df80
next  = 0x804df90
```

```
value = 86
self  = 0x804df90
```

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

Again, demonstrate DDD interactively

Concepts

★ Logging functions (”printf debugging”) are easy to use, but clobber code and output

★ To encapsulate and reuse debugging code, use dedicated logging functions or aspects

Concepts (2)

- Logging functions can be turned on or off (and may even remain in the source code)
- Aspects elegantly keep all logging code in one place
- Debuggers allow flexible + quick observation of arbitrary events

Concepts (3)

- To observe the final state of a crashing program, use a debugger
- Advanced debuggers allow to query events in a declarative fashion…
- …as well as visualizing events and data