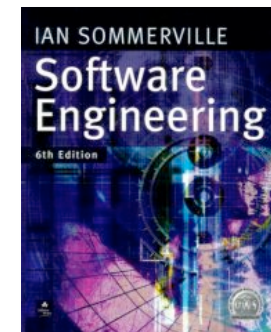


## 6. Testing and Debugging

# Testing and Debugging

## **Sources**

- > I. Sommerville, *Software Engineering*, Addison-Wesley, Sixth Edn., 2000.
- > [svnbook.red-bean.com](http://svnbook.red-bean.com)
- > [www.eclipse.org](http://www.eclipse.org)



# Roadmap

- > Testing — definitions and strategies
- > Understanding the run-time stack and heap
- > Debuggers
- > Timing benchmarks
- > Profilers



# Roadmap

- > **Testing — definitions and strategies**
- > Understanding the run-time stack and heap
- > Debuggers
- > Timing benchmarks
- > Profilers



# Testing

<i>Unit testing:</i>	test <i>individual (stand-alone) components</i>
<i>Module testing:</i>	test a <i>collection of related components</i> (a module)
<i>Sub-system testing:</i>	test <i>sub-system interface mismatches</i>
<i>System testing:</i>	(i) test <i>interactions between sub-systems</i> , and (ii) test that the complete systems fulfils <i>functional and non-functional requirements</i>
<i>Acceptance testing (alpha/beta testing):</i>	test system with <i>real rather than simulated data</i> .

*Testing is always iterative!*

# Regression testing

Regression testing means testing that *everything that used to work still works* after changes are made to the system!

- > tests must be *deterministic and repeatable*
- > should test “all” functionality
  - every interface (black-box testing)
  - all boundary situations
  - every feature
  - every line of code (white-box testing)
  - everything that can conceivably go wrong!

*It costs extra work to define tests up front, but they more than pay off in debugging & maintenance!*

## Caveat: Testing and Correctness

*“Program testing can be used to show the presence of bugs, but never to show their absence!”*  
—Edsger Dijkstra, 1970



# Testing a Stack

*We define a simple regression test that exercises all StackInterface methods and checks the boundary situations:*


```
public class LinkStackTest {  
    protected StackInterface<String> stack;  
    private int size;  
  
    @Before public void setUp() {  
        stack = new LinkStack<String>();  
    }  
  
    @Test public void empty() {  
        assertTrue(stack.isEmpty());  
        assertEquals(0, stack.size());  
    }  
    ...  
}
```



## Build simple test cases

*Construct a test case and check the obvious conditions:*

```
@Test public void oneElement() {  
    stack.push("a");  
    assertFalse(stack.isEmpty());  
    assertEquals(1, size = stack.size());  
    stack.pop();  
    assertEquals(size - 1, stack.size());  
}
```

 *What other test cases do you need to **fully exercise** a Stack implementation?*

## Check that failures are caught

*How do we check that an assertion fails when it should?*

```
@Test(expected=AssertionError.class)
public void emptyTopFails() {
    stack.top();
}

@Test(expected=AssertionError.class)
public void emptyRemoveFails() {
    stack.pop();
}
```

# ArrayStack

We can also implement a (variable) Stack using a (fixed-length) array to store its elements:

```
public class ArrayStack<E> implements StackInterface<E> {  
    private E store [];  
    private int capacity;  
    private int size;  
  
    public ArrayStack() {  
        store = null;           // default value  
        capacity = 0;          // available slots  
        size = 0;              // used slots  
    }  
}
```

 *What would be a suitable class invariant for ArrayStack?*

# Handling overflow

Whenever the array runs out of space, the Stack “**grows**” by allocating a larger array, and copying elements to the new array.

```
public void push(E item) {  
    {  
        if (size == capacity) {  
            grow();  
        }  
        store[++size] = item;    // NB: subtle error!  
    }  
}
```

 How would you implement the **grow()** method?

# Checking pre-conditions

```
public boolean isEmpty() { return size == 0; }
public int size() { return size; }

public E top() {
    assert(!this.isEmpty());
    return store[size-1];
}
public void pop() {
    assert(!this.isEmpty());
    size--;
}
```

*NB: we only check pre-conditions in this version!*

 *Should we also shrink() if the Stack gets too small?*

## Adapting the test case

We can easily adapt our test case by overriding the `setUp()` method in a subclass.

```
public class ArrayStackTest extends LinkStackTest {  
    @Before public void setUp() {  
        stack = new ArrayStack<String>();  
    }  
}
```

# Roadmap

- > Testing — definitions and strategies
- > **Understanding the run-time stack and heap**
- > Debuggers
- > Timing benchmarks
- > Profilers



# Testing ArrayStack

When we test our ArrayStack, we get a surprise:

```
java.lang.ArrayIndexOutOfBoundsException: 2  
    at p2.genstack.ArrayStack.push(ArrayStack.java:27)  
    at p2.genstack.LinkStackTest.twoElement(LinkStackTest.java:46)  
    at ...
```

*The stack trace tells us exactly **where** the exception occurred ...*



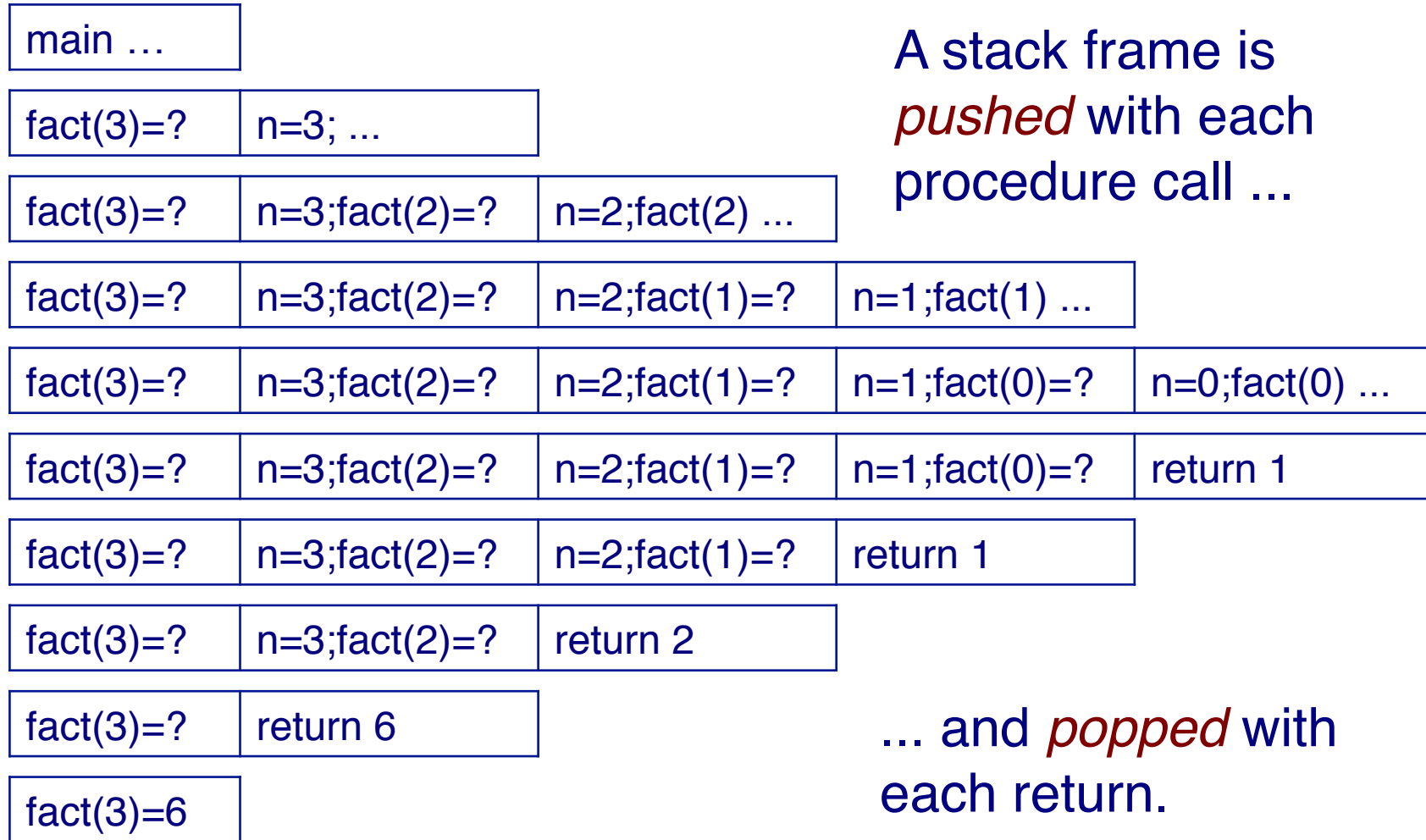
# The Run-time Stack

The **run-time stack** is a fundamental data structure used to record the *context* of a procedure that will be returned to at a later point in time. This **context** (AKA “**stack frame**”) *stores the arguments to the procedure and its local variables*.

*Practically all programming languages use a run-time stack:*

```
public static void main(String args[]) {  
    System.out.println( "fact(3) = " + fact(3));  
}  
public static int fact(int n) {  
    if (n<=0) { return 1; }  
    else { return n*fact(n-1) ; }  
}
```

# The run-time stack in action ...

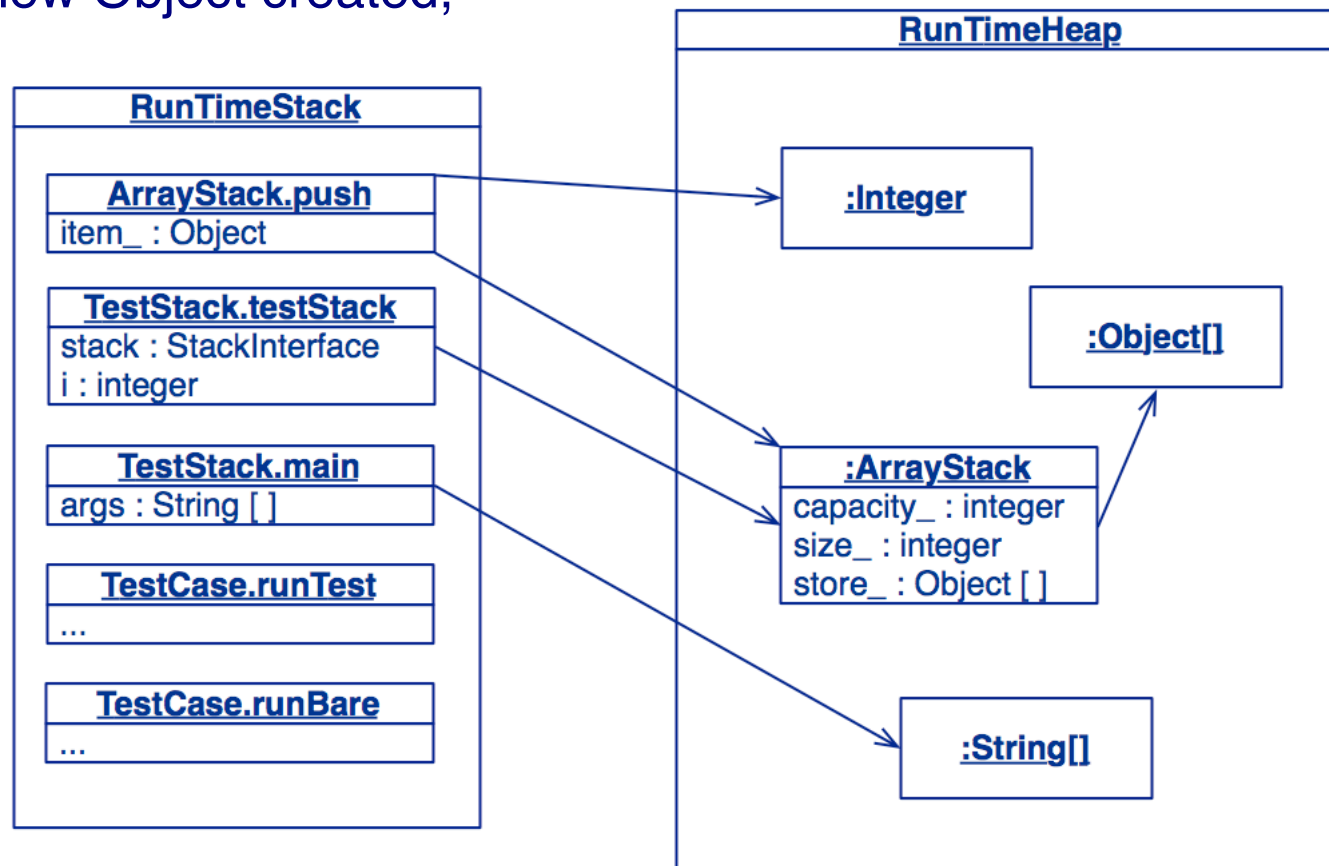


A stack frame is  
*pushed* with each  
procedure call ...

... and *popped* with  
each return.

# The Stack and the Heap

The **Heap** grows with each new Object created,



and shrinks  
when Objects  
are **garbage-**  
**collected.**

*NB: allocating  
objects is cheap  
on modern VMs*

# Roadmap

- > Testing — definitions and strategies
- > Understanding the run-time stack and heap
- > **Debuggers**
- > Timing benchmarks
- > Profilers



# Debuggers

A **debugger** is a tool that allows you to examine the state of a running program:

- > step through the program instruction by instruction
- > view the source code of the executing program
- > inspect (and modify) values of variables in various formats
- > set and unset breakpoints anywhere in your program
- > execute up to a specified breakpoint
- > examine the state of an aborted program (in a “core file”)

# Using Debuggers

Interactive debuggers are available for most mature programming languages and integrated in IDEs.

Classical debuggers are *line-oriented* (e.g., jdb); most modern ones are *graphical*.

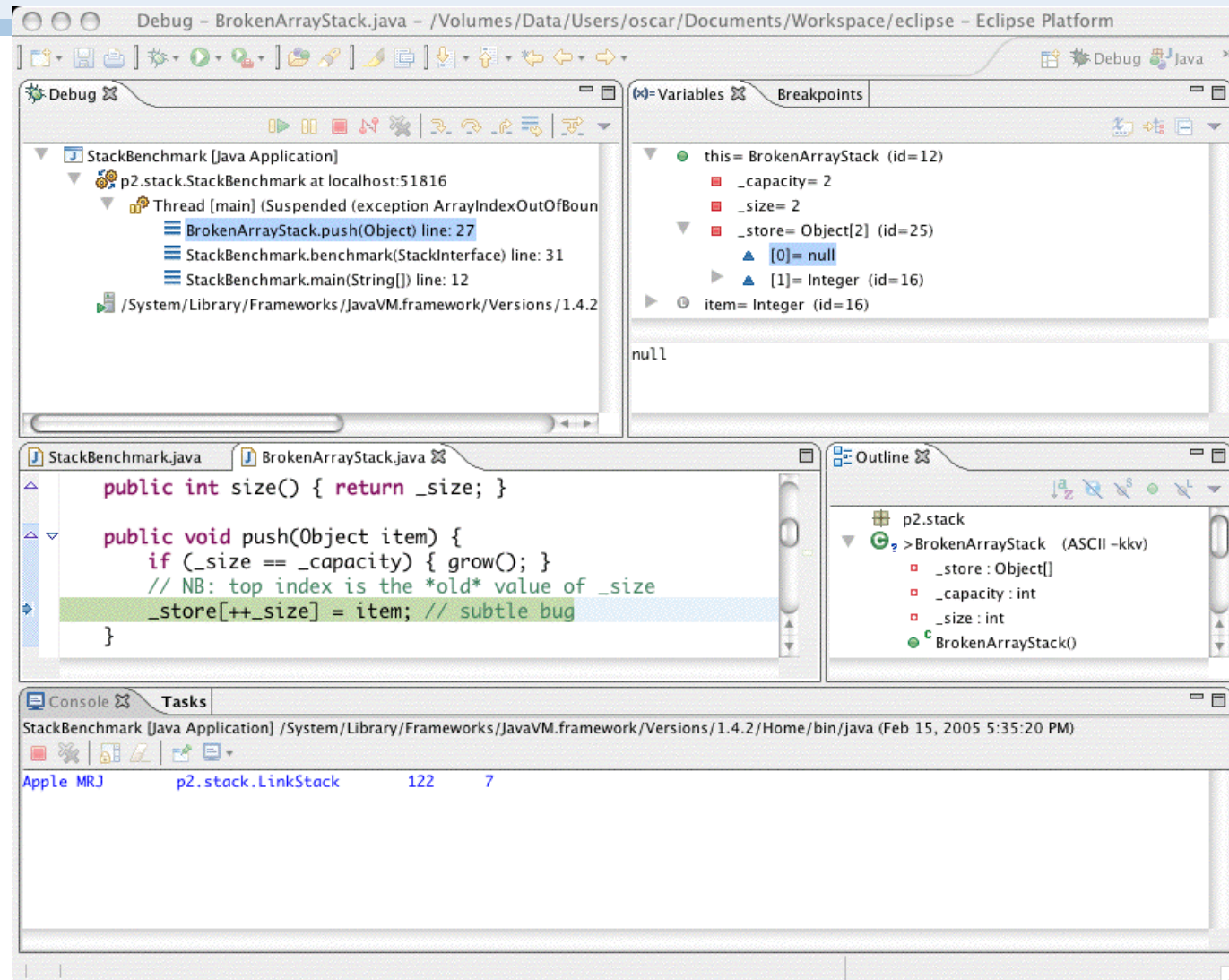
 When should you use a debugger?

✓ *When you are unsure why (or where) your program is not working.*

*NB: debuggers are object code specific — pick the right one for your platform!*

# Debugging in Eclipse

When unexpected exceptions arise, you can use the debugger to **inspect** the program **state** ...



# Debugging Strategy

## *Develop tests as you program*

- > Apply *Design by Contract* to decorate classes with **invariants** and **pre-** and **post-conditions**
- > Develop *unit tests* to exercise all paths through your program
  - use **assertions** (not print statements) to probe the program state
  - print the state **only** when an assertion fails
- > After every modification, do regression testing!

## *If errors arise during testing or usage*

- > Use the test results to track down and fix the bug
- > If you can't tell where the bug is, *then use a debugger* to identify the faulty code
  - fix the bug
  - identify and *add any missing tests!*

All software bugs are a matter of *false assumptions*. If you make your assumptions *explicit*, you will find and stamp out your bugs!



## Fixing our mistake

We erroneously used the *incremented size* as an index into the store, instead of the new size of the stack - 1:

```
public void push(E item) ... {  
    if (size == capacity) { grow(); }  
    store[size++] = item;  
    assert(this.top() == item);  
    assert(invariant());  
}
```



*NB: perhaps it would be clearer to write:*

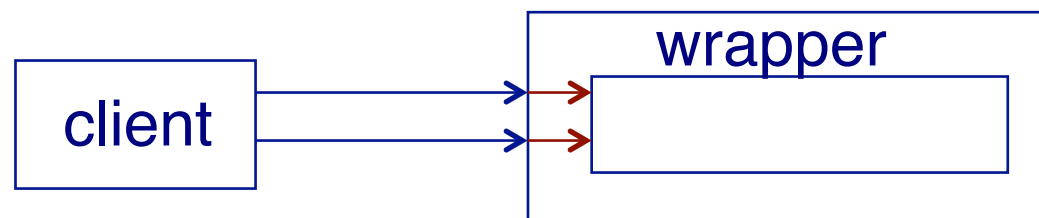
```
store[this.topIndex()] = item;
```

# Wrapping Objects

*Wrapping* is a fundamental programming technique for systems integration.

✎ What do you do with an object whose interface doesn't fit your expectations?

✓ *You wrap it.*



✎ What are possible **disadvantages** of wrapping?

# java.util.Stack

Java also provides a Stack implementation, but it is not compatible with our interface:

```
public class Stack extends Vector {  
    public Stack();  
    public Object push(Object item);  
    public synchronized Object pop();  
    public synchronized Object peek();  
    public boolean empty();  
    public synchronized int search(Object o);  
}
```

*If we change our programs to work with the Java Stack, we won't be able to work with our own Stack implementations ...*

# A Wrapped Stack

A wrapper class implements a required interface, by *delegating requests* to an instance of the wrapped class:

```
public class SimpleWrappedStack<E> implements StackInterface<E> {  
    private java.util.Stack<E> stack;  
    public SimpleWrappedStack() { this(new Stack<E>()); }  
    public SimpleWrappedStack(Stack<E> stack) { this.stack = stack; }  
    public void push(E item) { stack.push(item); }  
    public E top() { return stack.peek(); }  
    public void pop() { stack.pop(); }  
    public boolean isEmpty() { return stack.isEmpty(); }  
    public int size() { return stack.size(); }  
}
```



*Do you see any flaws with our wrapper class?*

## A contract mismatch

But running the test case yields:

```
java.lang.Exception: Unexpected exception,  
expected<java.lang.AssertionError> but  
was<java.util.EmptyStackException>  
...  
Caused by: java.util.EmptyStackException  
    at java.util.Stack.peek(Stack.java:79)  
    at p2.stack.SimpleWrappedStack.top(SimpleWrappedStack.java:32)  
    at p2.stack.LinkStackTest.emptyTopFails(LinkStackTest.java:28)  
    ...
```

 *What went wrong?*

## Fixing the problem ...

Our tester **expects** an empty Stack to throw an exception when it is popped, but `java.util.Stack` doesn't do this — *so our wrapper should check its preconditions!*

```
public class WrappedStack<E> implements StackInterface<E> {
    public E top() {
        assert !this.isEmpty();
        return super.top();
    }
    public void pop() {
        assert !this.isEmpty();
        super.pop();
        assert invariant();
    }
    ...
}
```

# Roadmap

- > Testing — definitions and strategies
- > Understanding the run-time stack and heap
- > Debuggers
- > **Timing benchmarks**
- > Profilers



# Timing benchmarks

Which of the Stack implementations performs better?

```
timer.reset();
for (int i=0; i<iterations; i++) {
    stack.push(item);
}
elapsed = timer.timeElapsed();
System.out.println(elapsed + " milliseconds for "
    + iterations + " pushes");
...
```

-  Complexity aside, how can you tell which implementation strategy will perform best?
- ✓ *Run a benchmark.*



# Timer

```
import java.util.Date;
public class Timer {           // Abstract from the
    protected Date startTime;  // details of timing
    public Timer() {
        this.reset();
    }
    public void reset() {
        startTime = new Date();
    }
    public long timeElapsed() {
        return new Date().getTime() - startTime.getTime();
    }
}
```

## Sample benchmarks (milliseconds)

<i><b>Stack Implementation</b></i>	<i><b>100K pushes</b></i>	<i><b>100K pops</b></i>
p2.stack.LinkStack	126	6
p2.stack.ArrayStack	138	3
p2.stack. <b>WrappedStack</b>	104	154

 *Can you explain these results? Are they what you expected?*

# Roadmap

- > Testing — definitions and strategies
- > Understanding the run-time stack and heap
- > Debuggers
- > Timing benchmarks
- > **Profilers**



# Profilers

A profiler tells you where a terminated program has *spent its time*.

1. your program must first be *instrumented* by
  - I. setting a compiler (or interpreter) option, or
  - II. adding instrumentation code to your source program
2. the program is run, generating a profile data file
3. the profiler is executed with the profile data as input

The profiler can then display the call graph in various formats

***Caveat:*** the technical details vary from compiler to compiler

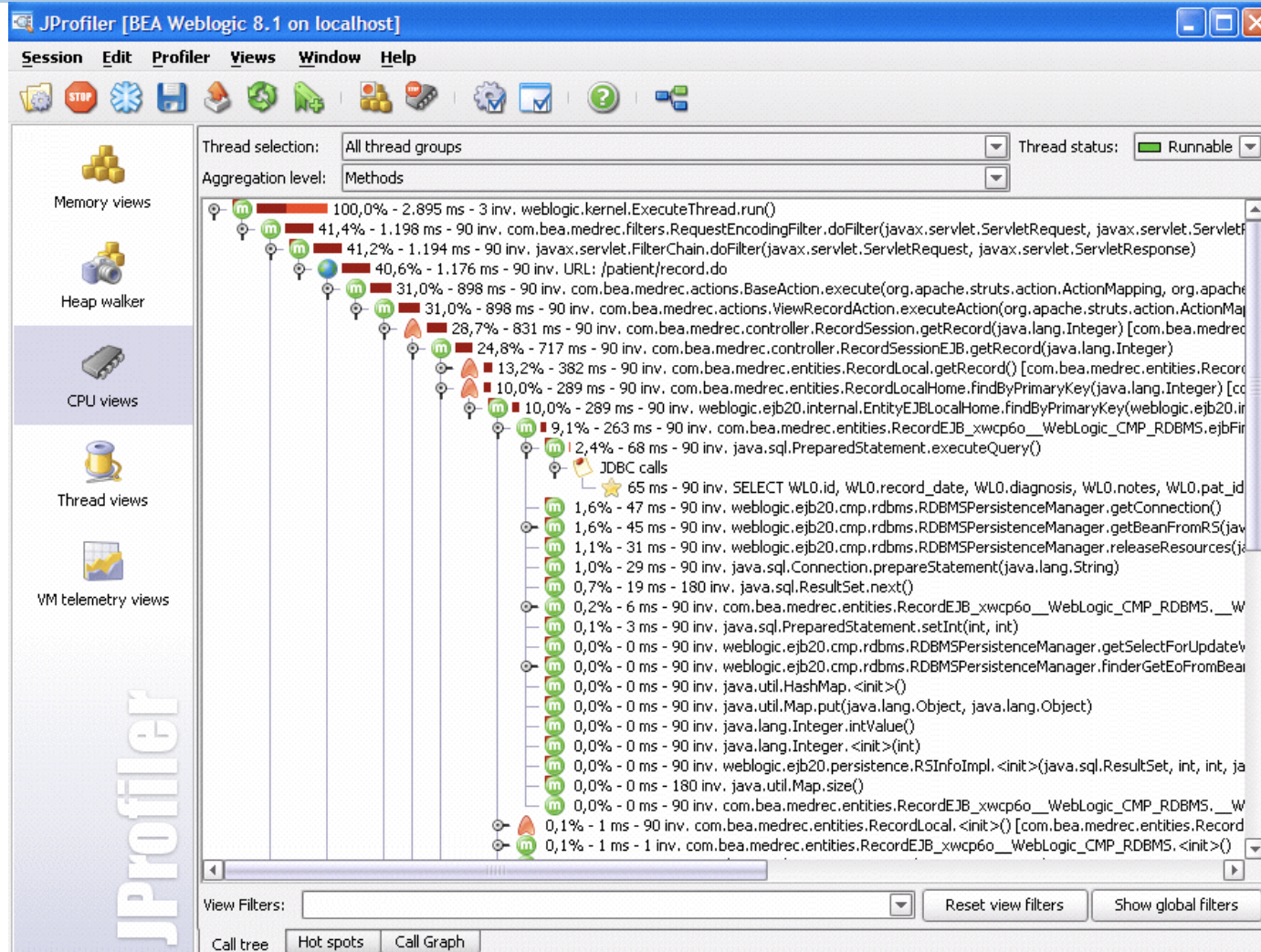
# Using java -Xprof

Flat profile of 0.61 secs (29 total ticks): main

Interpreted + native				Method
20.7%	0	+	6	java.io.FileOutputStream.writeBytes
3.4%	0	+	1	sun.misc.URLClassPath\$FileLoader.<init>
3.4%	0	+	1	p2.stack.LinkStack.push
3.4%	0	+	1	p2.stack.WrappedStack.push
3.4%	0	+	1	java.io.FileInputStream.open
3.4%	1	+	0	sun.misc.URLClassPath\$JarLoader.getResource
3.4%	0	+	1	java.util.zip.Inflater.init
3.4%	0	+	1	p2.stack.ArrayStack.grow
44.8%	1	+	12	Total interpreted

...

# Example of Profiler Features



# Using Profilers

 When should you use a profiler?

✓ *Always run a profiler before attempting to tune performance.*

 How early should you start worrying about performance?

✓ *Only after you have a clean, running program with poor performance.*

*NB: The call graph also tells you which parts of the program have (not) been tested!*

<http://www.javaperformancetuning.com/resources.shtml#ProfilingToolsFree>

# Coverage tools

- > A coverage tool can tell you what part of your code has been exercised by a test run or an interactive session. This helps you to:
  - identify dead code
  - missing tests








# EclEmma is a free Java coverage tool for Eclipse

The screenshot shows the Eclipse IDE with the EclEmma coverage tool active. The Package Explorer on the left shows the project structure, including the 'src' folder with files like 'Money.java', 'MoneyBag.java', and 'MoneyTest.java'. The main editor displays the source code of 'MoneyBag.java', which includes methods like 'appendBag', 'equals', and 'hashCode'. The bottom panel shows the 'Coverage' view, which displays a table of coverage data for the project and its components.








Element	Coverage	Covered Instructions	Total Instructions
P2Money	73.9 %	961	1301
src	73.9 %	961	1301
money.v1	87.1 %	121	139
money.v2	78.0 %	234	300
money.v3	66.1 %	275	416
money.v4	74.2 %	331	446
Money.java	86.7 %	78	90
MoneyBag.java	51.9 %	111	214
MoneyTest.java	100.0 %	142	142

At the bottom of the IDE, the status bar shows 'Writable', 'Smart Insert', and '31 : 1'.

## ***What you should know!***

-  *What is a **regression test**? Why is it important?*
-  *What **strategies** should you apply to design a test?*
-  *What are the **run-time** stack and **heap**?*
-  *How can you adapt client/supplier **interfaces** that don't **match**?*
-  *When are **benchmarks** useful?*

## *Can you answer these questions?*

-  *Why can't you use tests to demonstrate absence of defects?*
-  *How would you implement `ArrayStack.grow()`?*
-  *Why doesn't Java allocate objects on the run-time stack?*
-  *What are the advantages and disadvantages of wrapping?*
-  *What is a suitable class invariant for `WrappedStack`?*
-  *How can we learn where each `Stack` implementation is spending its time?*
-  *How much can the same benchmarks differ if you run them several times?*

# License

<http://creativecommons.org/licenses/by-sa/2.5/>



**You are free:**

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

**Under the following conditions:**



**Attribution.** You must attribute the work in the manner specified by the author or licensor.



**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

**Your fair use and other rights are in no way affected by the above.**