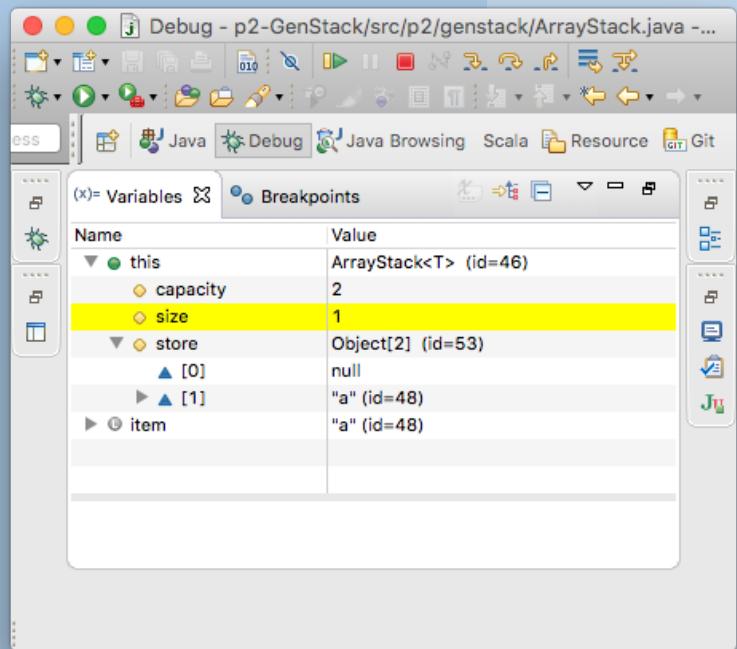


# 5. Testing and Debugging

Oscar Nierstrasz



# Roadmap

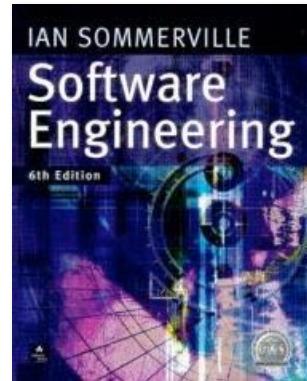


- > Testing — definitions and strategies
- > Understanding the run-time stack and heap
- > Debuggers
- > Timing benchmarks
- > Profilers

# Testing and Debugging

## Sources

- > I. Sommerville, *Software Engineering*, Addison-Wesley, Sixth Edn., 2000.
- > svnbook.red-bean.com
- > www.eclipse.org



# Roadmap



- > **Testing – definitions and strategies**
- > Test-Driven Development
- > 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!*

We focus in this course mainly on unit testing. A “unit” for us is a class. Even though JUnit is design mainly for unit testing, it can also be used to write arbitrary kinds of automated tests, for example, we also use it to test complete scenarios (i.e., module testing or even system testing).

# 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*

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

Consider the fact that you have to test all the functionality that you implement anyway. Rather than testing your code manually, you should write automated tests. This way you spend roughly the same effort as you would have anyway, but at the end you have an automated test that you can re-run any time later.

It is a mistake to think that once something works, you never need to test it again.

# Testing strategies

- > Tests should cover “all” functionality
  - every public method (black-box testing)
  - every feature
  - all boundary situations
  - common scenarios
  - exceptional scenarios
  - every line of code (white-box testing)
  - every path through the code

## Caveat: Testing and Correctness

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

*—Edsger Dijkstra, 1970*



Just because all your tests are green does not mean that your code is correct and free of bugs. This also does not mean that testing is futile!

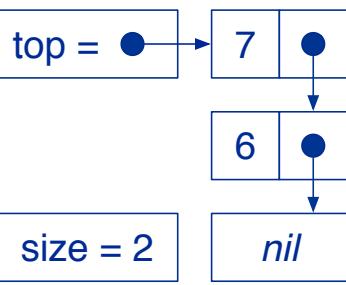
A good strategy is to add a new test whenever a new bug is discovered. The test should demonstrate the presence of the bug. When the test is green, you know that this particular instance of the bug is gone. Often bugs that arise despite testing are the trickiest to find, and they may easily be reintroduced. Writing a new test for the bug (i) documents the bug, (ii) helps you debug it, and (iii) ensures that the bug will be flagged if it ever appears again.

# Roadmap



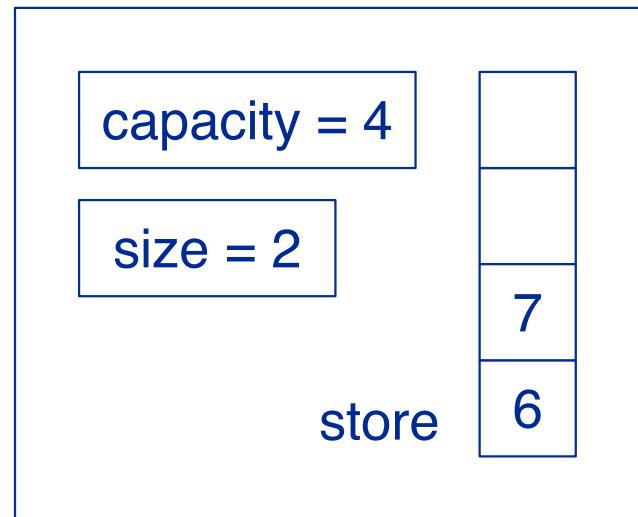
- > Testing — definitions and strategies
- > **Test-Driven Development**
- > Understanding the run-time stack and heap
- > Debuggers
- > Timing benchmarks
- > Profilers

# Multiple Stack implementations



Recall our LinkStack implementation

An alternative way to implement a stack is using an array. When the array runs out of space, we simply allocate a large one and copy all the elements



Note that it is not obvious which approach is better. The **LinkStack** grows and shrinks with every push or pop. An **ArrayList** only grows when it runs out of space. A push or pop is very cheap with an **ArrayList**, unless it runs out of space, when an expensive copy operation must be performed.

It is also not clear what size the initial capacity of an **ArrayList** should be, nor how much it should “grow” when a larger store is needed.

# Testing a stack interface

*Recall that we implemented tests for the interface of our LinkStack class.*

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

Since there was no complex code in the `LinkStack` implementation, we only write tests that focused on the interface. Since `ArrayStack` implements the same interface, we can actually *reuse* our older tests.

# 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>();  
    }  
}
```

# Test-driven development

Instead of writing a class first and then writing the tests, TDD promotes a development approach in which you alternate in writing a test first, and then writing code that lets this test pass.

*Since we already have our tests in place, we can use them to develop our `ArrayStack` in a test-driven way*

TDD forces you to think about how to test your code from the beginning. It can also influence your design in two ways: first, it helps you to specify the interface of your class up front, since that is what will be tested, and second, it will help you design your class in a way that allows it to be easily tested.

# Exploiting “quick fixes”

The screenshot shows a Java code editor in Eclipse. The cursor is at the end of the line 'stack = new ArrayStack<String>();'. A context menu is open with the following options:

- Create class 'ArrayStack<T>' (highlighted)
- Change to 'StackInterface' (p2.genstack)
- Change to 'Array' (java.lang.reflect)
- Change to 'Arrays' (java.util)
- Change to 'ArrayStackTest' (p2.genstack)
- Rename in file (⌘2 R)
- Fix project setup...

A tooltip on the right side of the menu provides additional information:

Opens the new class wizard to create the type.

Package: p2.genstack  
public class ArrayStack<T> implements StackInterface<String> {  
}

Press 'Tab' from proposal table or click for focus

```
12 public class ArrayStackTest extends LinkStackTest {  
13  
14     @Before public void setUp() {  
15         stack = new ArrayStack<String>();  
16     }  
17  
18 }  
19
```

IDEs like Eclipse recognize that the class you want to test does not exist yet, and can propose to generate it for you as a “quick fix”.

To view the quick fixes in Eclipse, just click on the red warning symbol in the left margin. Several possible ways to fix the problem will be proposed, including generating the missing class as the top solution.

# A generated class

```
package p2.genstack;

public class ArrayStack<T> implements StackInterface<String> {

    @Override
    public boolean isEmpty() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public int size() {
        // TODO Auto-generated method stub
        return 0;
    }

    ...
}
```

The generated class contains empty method stubs for the declared interface. Of course the generated code will not be correct, but it will compile and can be tested.

# Failing tests as “to do” items

The screenshot shows the Eclipse IDE interface with the following components:

- Left Panel:** A code editor window titled "LinkStackTest.java" containing Java test code. A specific line of code, `assertTrue(stack.isEmpty());`, is highlighted in blue.
- Right Panel:** A JUnit perspective window showing the test results.
  - Summary Bar:** Displays "Runs: 9/9", "Errors: 0", and "Failures: 9".
  - Test Tree:** Shows the test class "p2.genstack.ArrayStackTest" and its methods: pushPopOneElement, twoElement, brokenSequence, pushOneElement, empty, emptyToCfails, emptyRemoveFails, firstInLastOut, and pushNull. The "empty" method is expanded, showing it failed.
  - Failure Trace:** A detailed stack trace for the failure in the "empty" method, listing classes like java.lang.AssertionError, org.junit.Assert, and p2.genstack.LinkStackTest.

*Each failing test can be seen as a “to do” item. When all the tests are green, you are done.*

```
41
42  @Test public void pushPopOneElement() {
43      stack.push("a");
44      stack.pop();
45      assertTrue(stack.isEmpty());
46      assertEquals(0, stack.size());
47  }
48
49  @Test public void pushNull() {
50      stack.push(null);
51      assertFalse(stack.isEmpty());
```

```
59
60      stack.push("b");
61      assertEquals("b", stack.top());
62      stack.pop();
63      assertEquals("a", stack.top());
64      stack.pop();
65      assertTrue(stack.isEmpty());
```

# ArrayStack

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

```
public class ArrayStack<T> implements StackInterface<T> {  
    protected T store [];  
    protected int capacity;  
    protected 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?

Here we decide on an initial capacity of zero, and we do not allocate any initial store. This means that the store will be initialized *lazily* the first time it is needed. (Lazy initialization is a common programming idiom to save time and space when an application starts, by delaying the cost to a later point in the execution.)

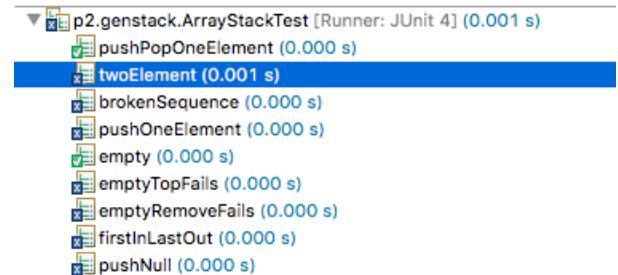
Recall that the *class invariant* formalizes the valid states of an object (see the lecture on Design by Contract.)

*What are the valid states of an `ArrayStack`, and how would you express this is a Boolean predicate?*

# Passing tests

Some of the more trivial tests pass quite quickly

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



# Handling overflow

```
public void push(T item) {  
    if (size == capacity) {  
        grow();  
    }  
    store[++size] = item;  
    assert this.top() == item;  
    assert invariant();  
}  
  
public T top() {  
    assert !this.isEmpty();  
    return store[size-1];  
}  
  
public void pop() {  
    assert !this.isEmpty();  
    size--;  
    assert invariant();  
}
```

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



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

The `grow()` method must allocate a (strictly) larger array, copy all the elements, and ensure that the old array can be garbage-collected (i.e., make sure no variables refer to it any more).

*What would be a suitable larger size?*

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

# Roadmap



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

# Testing ArrayStack

When we test our `ArrayStack`, we get a surprise:

```
java.lang.AssertionError  
at p2.genstack.ArrayStack.push(ArrayStack.java:39)  
at p2.genstack.LinkStackTest.pushPopOneElement  
  (LinkStackTest.java:43)  
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) ; }  
}
```

Each JVM thread has a private Java virtual machine stack, created at the same time as the thread. A JVM stack stores *frames*, that hold local variables and partial results, and play a part in method invocation and return.

Because the Java VM stack is never manipulated directly except to push and pop frames, frames may actually be heap-allocated. The memory for a Java virtual machine stack does not need to be contiguous. `OutOfMemoryError - CANNOT ALLOCATE STACK.`

# The run-time stack in action ...

main ...

fact(3)=? n=3; ...

fact(3)=? n=3;fact(2)=? n=2;fact(2) ...

fact(3)=? n=3;fact(2)=? n=2;fact(1)=? n=1;fact(1) ...

fact(3)=? n=3;fact(2)=? n=2;fact(1)=? n=1;fact(0)=? n=0;fact(0) ...

fact(3)=? n=3;fact(2)=? n=2;fact(1)=? n=1;fact(0)=? return 1

fact(3)=? n=3;fact(2)=? n=2;fact(1)=? return 1

fact(3)=? n=3;fact(2)=? return 2

fact(3)=? return 6

fact(3)=6

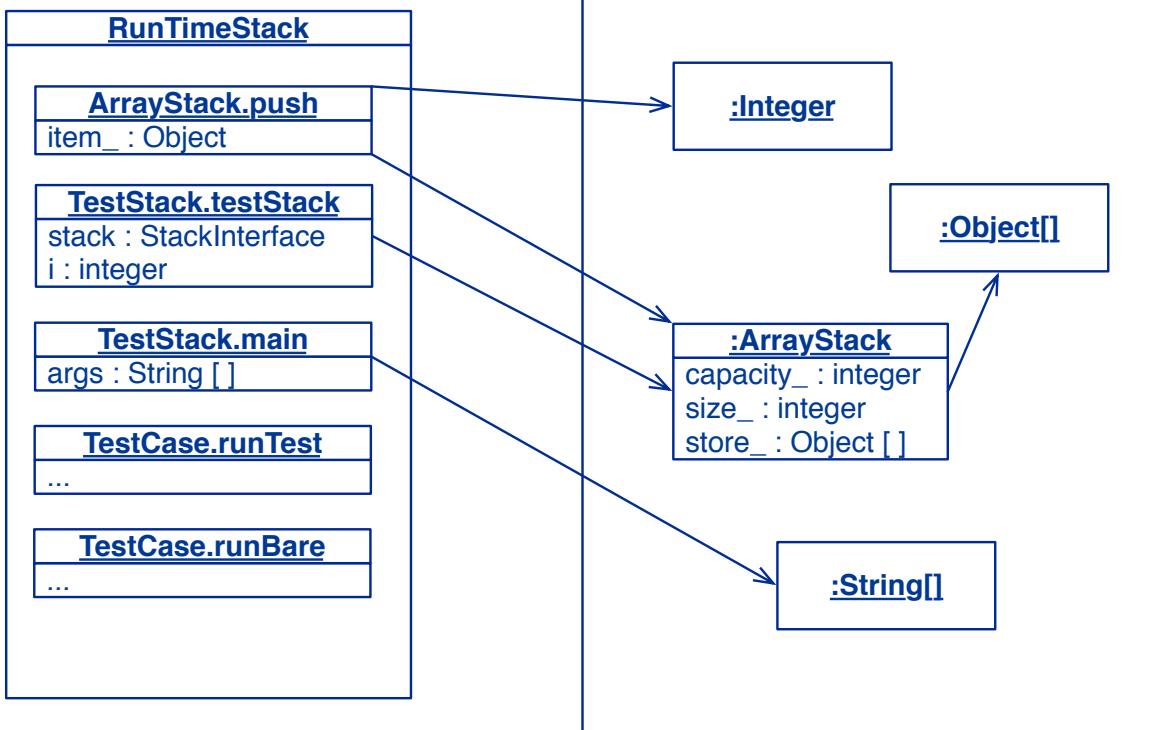
A stack frame is  
*pushed* with each  
procedure call ...

... and *popped* with  
each return.

The caller pushes the return address onto the stack, and the called subroutine, when it finishes, pops the return address off the call stack (and transfers control to that address). If a called subroutine calls on to yet another subroutine, it will push its return address onto the call stack, and so on, with the information stacking up and unstacking as the program dictates. If the pushing consumes all of the space allocated for the call stack, an error called a stack overflow occurs.

# 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*

The JVM has a heap that is shared among all Java virtual machine threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated. The heap is created on VM start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector); objects are never explicitly deallocated. The JVM assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.

# Roadmap



- > Testing — definitions and strategies
- > Test-Driven Development
- > 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”)

A debugger offers the ability to perform step through execution, to set breakpoints and values, to inspect variables and values, and to suspend and resume threads. Additionally, you can debug applications that are running on a remote machine.

*Whenever you have the temptation to add a “print” statement to your code to find out what is happening, instead you should set a breakpoint and directly inspect the program state.*

Aside: A *core dump* is the recorded state of the working memory of a computer program at a specific time, generally when the program has terminated abnormally (crashed). (This quaint term refers to the time when computer storage consisted of “magnetic core memory”, up to about 1975.) It is possible to force a JVM to “dump core.”

# 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!*

# Setting a breakpoint in Eclipse

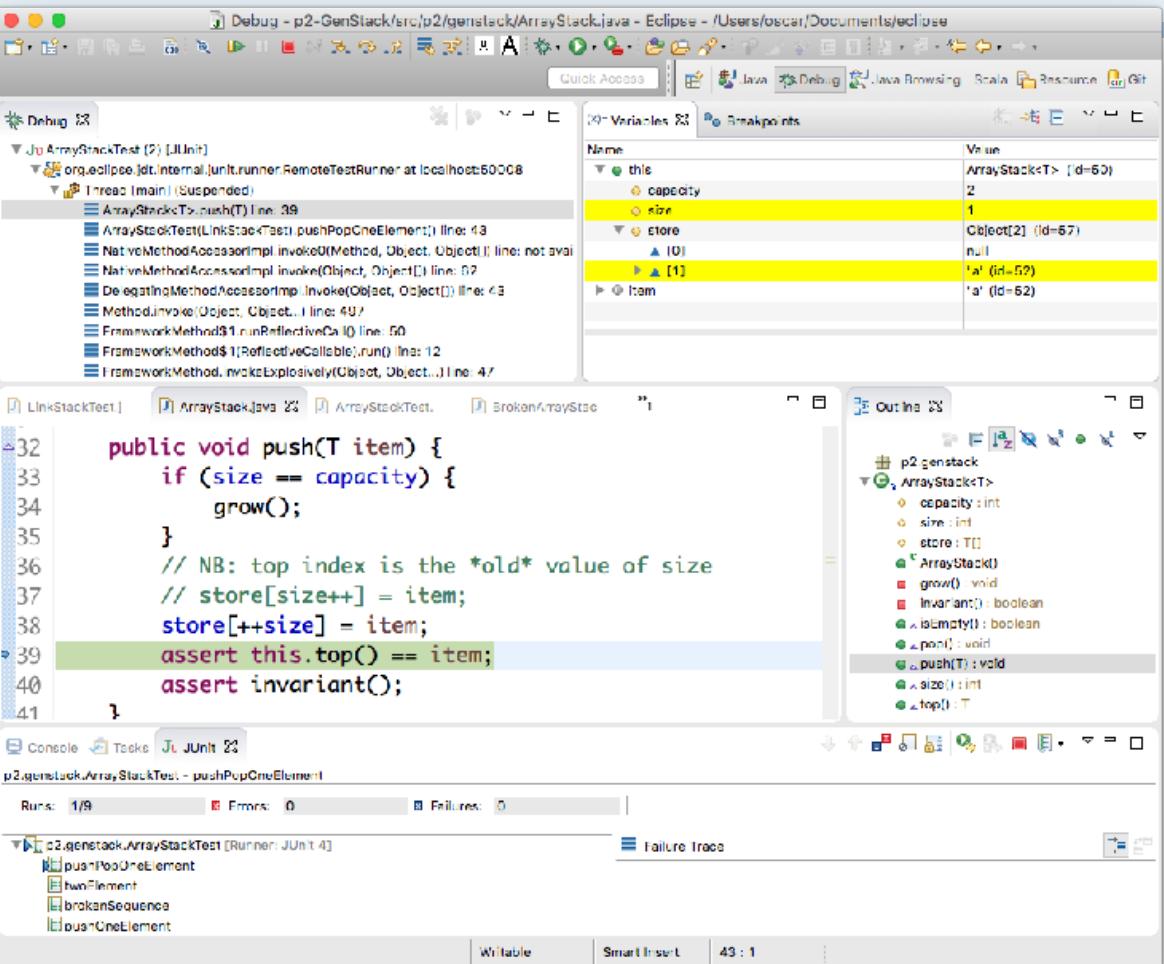
The screenshot shows the Eclipse IDE interface with the following details:

- Top Bar:** Shows the title "Java - p2 GenStack/src/p2/genstack/LinkStackTest.java - Eclipse - /Users/ocesar/Documents/eclipse".
- Toolbar:** Standard Eclipse toolbar with various icons for file operations, selection, and navigation.
- Left Side:** Project Explorer and Package Explorer panes are visible.
- Central Area:** The code editor displays the `LinkStackTest.java` file. A red circle highlights the line number 43, which contains the code `stack.pop();`. The code is annotated with several assertions using JUnit annotations like `@Test`.
- Right Side:** The **JUnit View** shows the test results:
  - Summary: "Finished after 0.018 seconds"
  - Runs: 9/9, Errors: 0, Failures: 4
  - Test Suite: `p2 genstack.ArrayStackTest [Runner: JUnit: 4] (0.001 s)`
    - Test Cases:
      - `pushPopOneElement (0.001 s)` (highlighted in blue)
      - `twoElement (0.000 s)`
      - `brokenSequence (0.000 s)`
      - `pushOneElement (0.000 s)`
      - `empty (0.000 s)`
      - `emptyTopFails (0.000 s)`
      - `emptyRemoveFails (0.000 s)`
      - `firstInLastOut (0.000 s)`
      - `pushNull (0.000 s)`
  - Failure Trace:** Shows the stack trace for a failure, starting with `java.lang.AssertionError`.

Set a breakpoint within the failing test method (double-click in the margin of the code browser to the left of the line number so a dot representing the breakpoint appears). Then execute “Debug As” JUnit test (i.e., rather than “Run As”).

# Debugging in Eclipse

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



The code will run up to the breakpoint and then start the debugger. You may then step into, step over and step return out of code, inspecting the state of the objects and methods as you navigate.

To remove the breakpoint, just double-click again on the dot.

# 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*

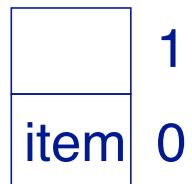
- > Identify and *add any missing tests!*
- > Use the test results to track down and fix the bug
- > If you can't tell where the bug is, *then use a debugger* to identify the faulty code

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:*

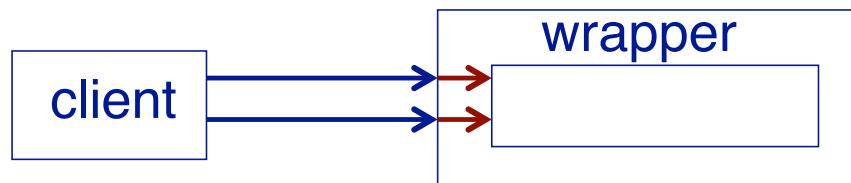
```
store[this.topIndex()] = item;
```

This is a classic example of an “off by one” error. This is one of the most common (and notorious) bugs in software systems. Typically code with such bugs works fine until a certain limit case is encountered. For this reason it is important to *write tests for boundary conditions*, i.e., where arguments to methods are on or next to boundary values, such as minimum or maximum values of ranges, or values next to them.

# 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<E> extends Vector<E> {  
    public Stack();  
    public Object push(E item);  
    public synchronized E pop();  
    public synchronized E 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> {  
    protected 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
- > Test-Driven Development
- > 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>Stack Implementation</i>	<i>100K pushes</i>	<i>100K pops</i>
p2.stack.LinkStack	126	6
p2.stack.ArrayStack	138	3
<b>p2.stack.WrappedStack</b>	104	154

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

# Roadmap



- > Testing — definitions and strategies
- > Test-Driven Development
- > 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

Profilers are used to find out what parts of the code have been executed and how much time was spent in each part. Running a profiler should be the first step whenever you discover that performance is not good enough and you want to optimize parts of your code. The profiler will tell you where the program is spending most of its time.

# 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

...

To use this, simply set the runtime (vm) flag -Xprof

# Example of Profiler Features

JProfiler [BEA Weblogic 8.1 on localhost]

Session Edit Profiler Views Window Help

Memory views      Heap walker      CPU views      Thread views      VM telemetry views

Thread selection: All thread groups      Thread status: Runnable

Aggregation level: Methods

Call tree

View Filters:  Reset view filters Show global filters

Call tree Hot spots Call Graph

The screenshot displays the JProfiler interface for BEA Weblogic 8.1 running on localhost. The main window shows a detailed call tree. At the top, the method `weblogic.kernel.ExecuteThread.run()` is highlighted with a red bar, indicating it is the current target. Below it, the tree branches into several other methods, each with its execution time and a corresponding colored bar. The methods include `com.bea.medrec.actions.BaseAction.execute()`, `com.bea.medrec.controller.RecordSessionEB.getRecord()`, and various JDBC calls like `SELECT WLO.id, WLO.record_date, WLO.diagnosis, WLO.notes, WLO.pat_id`. The left sidebar lists five views: Memory views, Heap walker, CPU views, Thread views, and VM telemetry views. The bottom navigation bar includes tabs for Call tree, Hot spots, and Call Graph.

This is an example of a nice tool that visually displays the call graph as a browsable tree. you can navigate through the tree to inspect further details.

# 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 interface with several open windows:

- Package Explorer:** Shows the project structure with various source files like `co-expendableArray`, `co-inAspects`, etc.
- Editor:** Displays the `ParenMatch.java` file containing code for matching parentheses. The code is annotated with green highlights indicating covered lines.
- Context Menu:** A context menu is open over the code, with the "Coverage As" option highlighted. Other options include "Java Application", "JUnit Test", and "Coverage Configurations...".
- Right-hand pane:** Shows the "PareMatchTest (1) (Mar 21, 2017 4:41:08 PM)" coverage report. It lists all methods and their coverage percentages. Most methods are at 100% coverage, except for `isLeftParen(char)` which is at 41.7%.

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {
        char c = line.charAt(i);
        if (isLeftParen(c)) {
            // expect matching right paren later
            stack.push(matchingRightParen(c)); // Autobalance
        } else {
            if (isRightParen(c)) {
                // empty stack => missing left paren
                if (stack.isEmpty()) { return false; }
                if (stack.top().equals(c)) { // Autobalance
                    stack.pop();
                } else { return false; } // mismatched
            }
        }
    }
    return !stack.isEmpty(); // not empty => missing right paren
}
```

Element	Coverage	Covered Lines
p2-GenStack	26.3 %	26.3
src	26.3 %	26.3
p2-GenStack	0.0 %	0
LinkStackTest.java	0.0 %	0
LinkStack.java	0.0 %	0
PareMatch.java	61.7 %	61.7
PareMatch	41.7 %	41.7
isLeftParen(char)	0.0 %	0
isRightParen(char)	0.0 %	0
isMatch(String)	0.0 %	0
isMatchingRightParen(char)	0.0 %	0
isNotMatch(String, StackInterface)	100.0 %	100.0
isLeftParen(char)	100.0 %	100.0
isRightParen(char)	100.0 %	100.0
parenMatch()	100.0 %	100.0
wrappedStack	0.0 %	0
Stack3Invariance.java	0.0 %	0
SimpleWrappedStack.java	0.0 %	0
LinkStack.java	80.7 %	80.7
Timer.java	0.0 %	0
ArrayStackTest.java	0.0 %	0
WrappedStackTest.java	0.0 %	0
PareMatchUtilTest.java	100.0 %	100.0

Here we see in green which code has been exercised and in red which code has not. By running your tests with the coverage tool you can quickly discover which code has not been covered by your test suite.

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

- 1 ⓘ *What is a regression test? Why is it important?*
- 2 ⓘ *What strategies should you apply to design a test?*
- 3 ⓘ *How does test-driven development work?*
- 4 ⓘ *What are the run-time stack and heap?*
- 5 ⓘ *How can you adapt client/supplier interfaces that don't match?*
- 6 ⓘ *When are benchmarks useful?*

1)

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*

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

Just because all your tests are green does not mean that your code is correct and free of bugs. This also does not mean that testing is futile!

A good strategy is to add a new test whenever a new bug is discovered. The test should demonstrate the presence of the bug. When the test is green, you know that this particular instance of the bug is gone. Often bugs that arise despite testing are the trickiest to find, and they may easily be reintroduced. Writing a new test for the bug (i) documents the bug, (ii) helps you debug it, and (iii) ensures that the bug will be flagged if it ever appears again.

2)

- > Tests should cover “all” functionality
  - every public method (black-box testing)
  - every feature
  - all boundary situations
  - common scenarios
  - exceptional scenarios
  - every line of code (white-box testing)
  - every path through the code

3)

TDD forces you to think about how to test your code from the beginning. It can also influence your design in two ways: first, it help you to specify the interface of your class up front, since that is what will be tested, and second, it will help you design your class in a way that allows it to be easily tested.

#### 4) Stack :

Each JVM thread has a private Java virtual machine stack, created at the same time as the thread. A JVM stack stores *frames*, that hold local variables and partial results, and play a part in method invocation and return.

Because the Java VM stack is never manipulated directly except to push and pop frames, frames may actually be heap-allocated. The memory for a Java virtual machine stack does not need to be contiguous. OutOfMemoryError - CANNOT ALLOCATE STACK.

#### Heap:

The JVM has a heap that is shared among all Java virtual machine threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated. The heap is created on VM start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector); objects are never explicitly deallocated. The JVM assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.

#### 6)

profilers are used to find out what parts of the code have been executed and how much time was spent in each part. Running a profiler should be the first step whenever you discover that performance is not good enough and you want to optimize parts of your code. The profiler where tell you where the program is spending most of its time.

##### ⌚ When should you use a profiler?

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

##### ⌚ How early should you start worrying about performance?

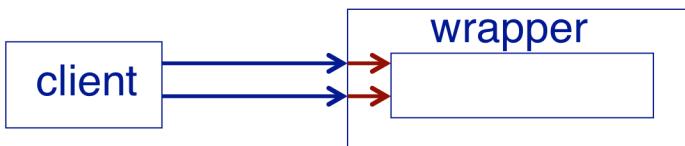
- ✓ *Only after you have a clean, running program with poor performance.*

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

#### 5) Use a wrapper.

*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.**



## ***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?*



## Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

### You are free to:

**Share** — copy and redistribute the material in any medium or format

**Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

### Under the following terms:



**Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



**ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>