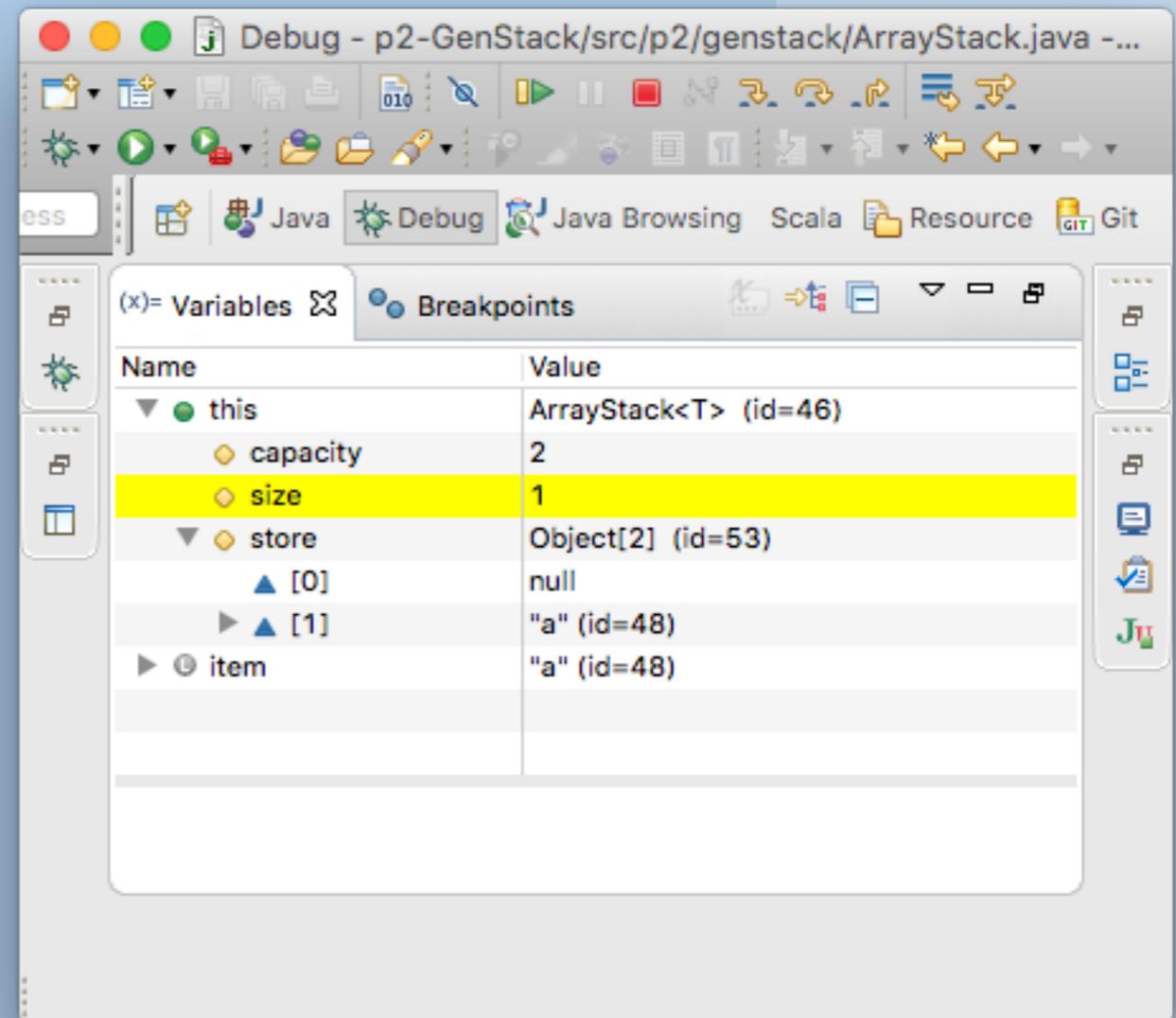


5. Testing and Debugging

Oscar Nierstrasz



Roadmap

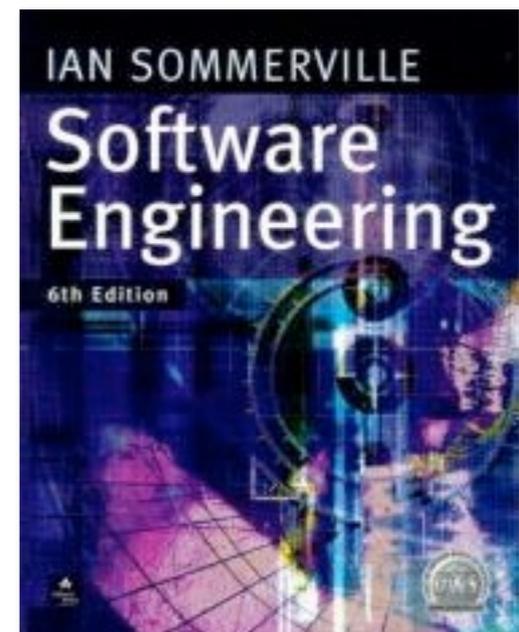


- > Testing — definitions and strategies
- > Test-Driven Development
- > Understanding the run-time stack and heap
- > Debuggers
- > Timing benchmarks
- > Profilers and other tools

Testing and Debugging

Sources

- > I. Sommerville, *Software Engineering*, Addison-Wesley, Sixth Edn., 2000.



Roadmap



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

Testing

<i>Unit testing:</i>	test <i>individual (stand-alone) components</i>
<i>Module testing:</i>	test a <i>collection of related components (a module)</i>
<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 fulfil <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.

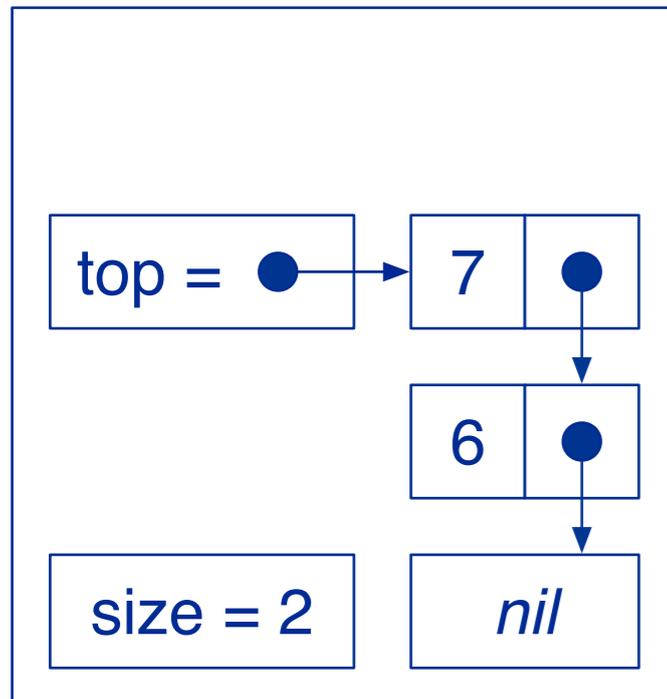
Roadmap



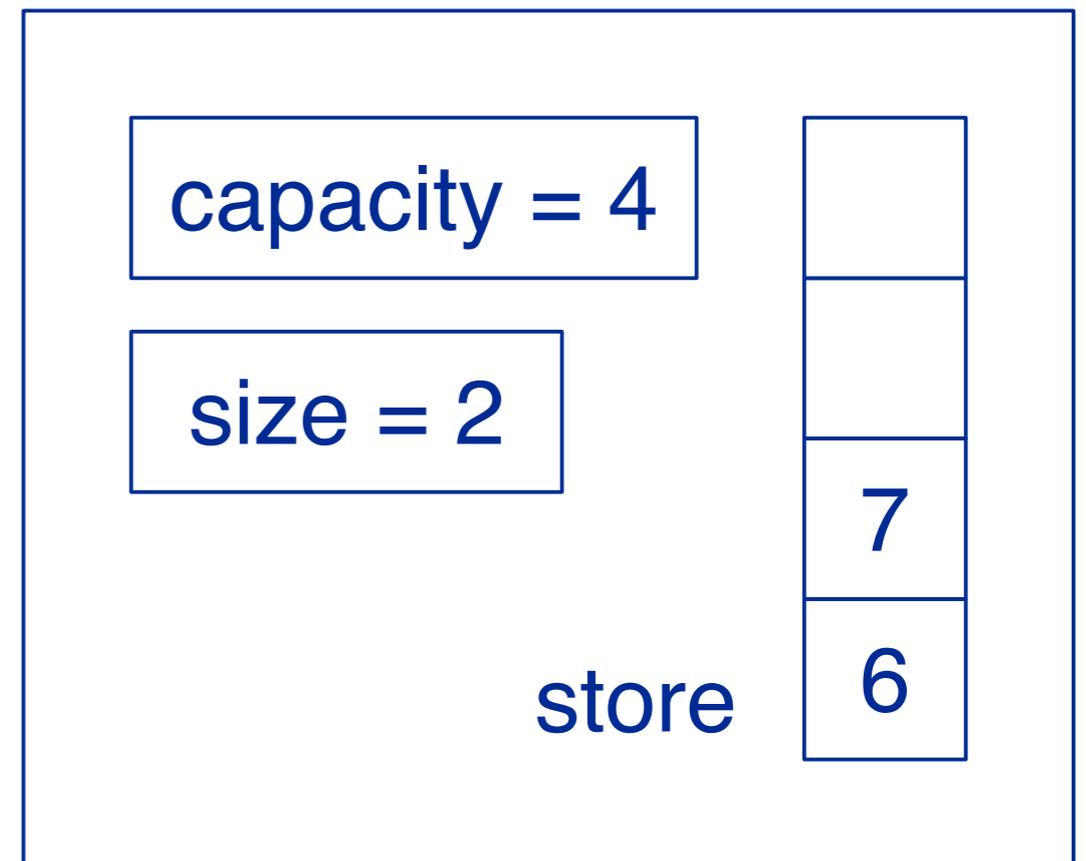
- > Testing — definitions and strategies
- > **Test-Driven Development**
- > Understanding the run-time stack and heap
- > Debuggers
- > Timing benchmarks
- > Profilers and other tools

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 `ArrayStack` only grows when it runs out of space. A push or pop is very cheap with an `ArrayStack`, 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 `ArrayStack` 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;

    @BeforeEach 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 {  
    @BeforeEach public void setUp() {  
        stack = new ArrayStack<String>();  
    }  
}
```

Test-driven development

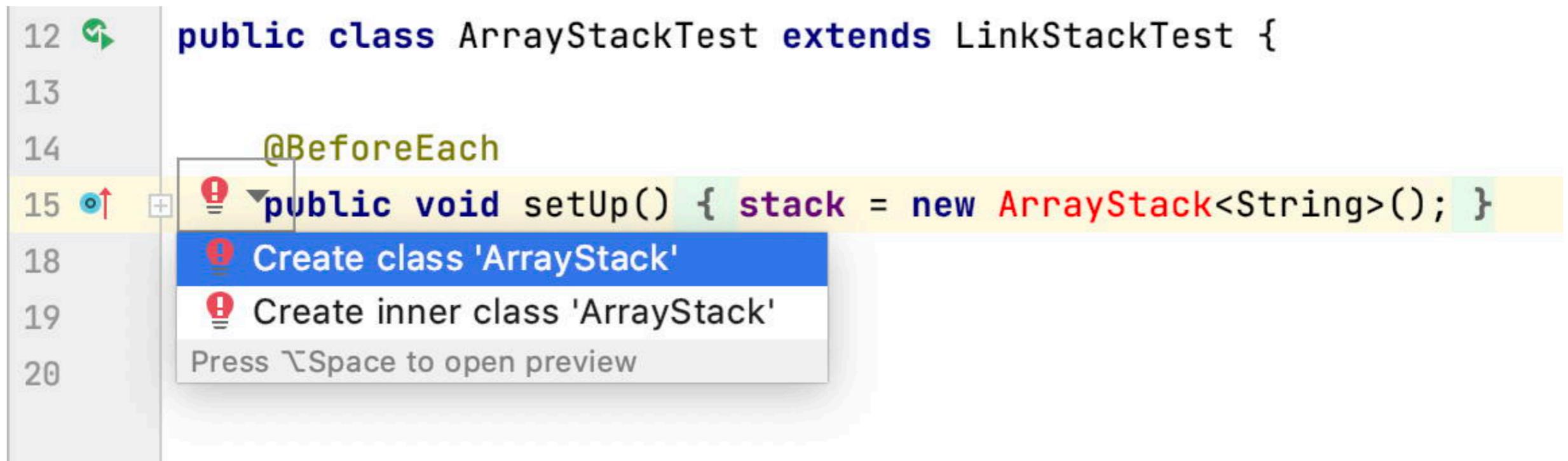
Instead of writing a class first and then writing the tests, TDD promotes a development approach in which you alternate in writing a test first, and then writing code that lets this test pass.

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 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.

Exploiting “quick fixes”

```
12  ↩️ public class ArrayStackTest extends LinkStackTest {
13
14      @BeforeEach
15  ⬆️  public void setUp() { stack = new ArrayStack<String>(); }
18
19
20
```



IDEs like IntelliJ 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 IntelliJ, 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() {
        return false;
    }

    @Override
    public int size() {
        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

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

```
18     stack = new LinkStack<String>();
19 }
20
21 @Test
22 public void empty() {
23     assertTrue(stack.isEmpty());
24     assertEquals( expected: 0, stack.size());
25 }
26
27 @Test
28 public void ...
29     asse
30 }
31
32 @Test
```

Run: ArrayStackTest

Tests failed: 9 of 9 tests

Test Results

- ArrayStackTest 18 ms
 - pushPopOneElement 11 ms
 - twoElement() 3 ms
 - brokenSequence() 2 ms
 - pushOneElement()
 - empty()
 - emptyTopFails()
 - emptyRemoveFails() 1 ms
 - firstInLastOut() 1 ms
 - pushNull()

org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
<4 internal calls>
at p2.genstack.LinkStackTest.empty(LinkStackTest.java:23) <19 internal calls>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal calls>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <21 internal calls>

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.)

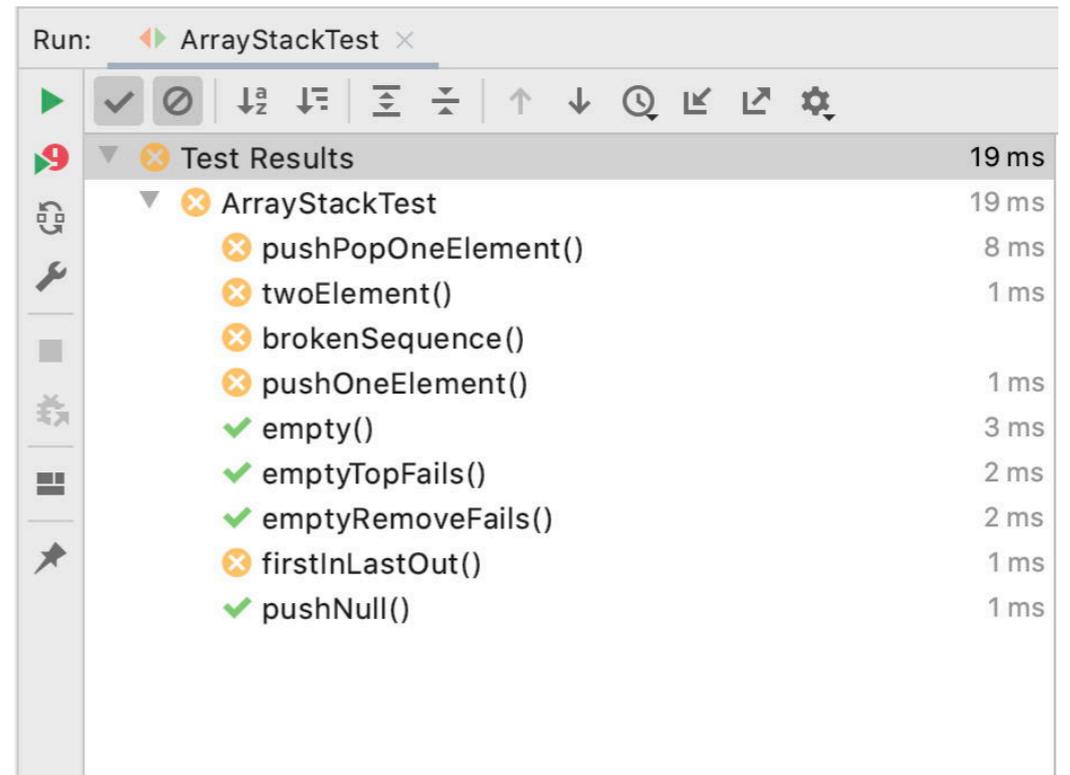
Recall that the *class invariant* formalizes the valid states of an object (see the lecture on Design by Contract.)

What are the valid states of an `ArrayStack`, and how would you express this as 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;  
}
```



The screenshot shows a test runner window titled "Run: ArrayStackTest". The interface includes a toolbar with icons for running, pausing, and refreshing tests. Below the toolbar, the "Test Results" section is expanded to show the following test results:

Test Name	Result	Duration
ArrayStackTest	Failed	19 ms
pushPopOneElement()	Failed	8 ms
twoElement()	Failed	1 ms
brokenSequence()	Failed	1 ms
pushOneElement()	Failed	1 ms
empty()	Passed	3 ms
emptyTopFails()	Passed	2 ms
emptyRemoveFails()	Passed	2 ms
firstInLastOut()	Failed	1 ms
pushNull()	Passed	1 ms

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 and other tools

Testing ArrayStack

When we test our ArrayStack, we get a surprise:

```
java.lang.AssertionError Create breakpoint
  at p2.genstack.ArrayStack.push(ArrayStack.java:37)
+ at p2.genstack.LinkStackTest.pushPopOneElement(LinkStackTest.java:47) <19 internal calls>
+ at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal calls>
+ at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <21 internal calls>
```

*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.*

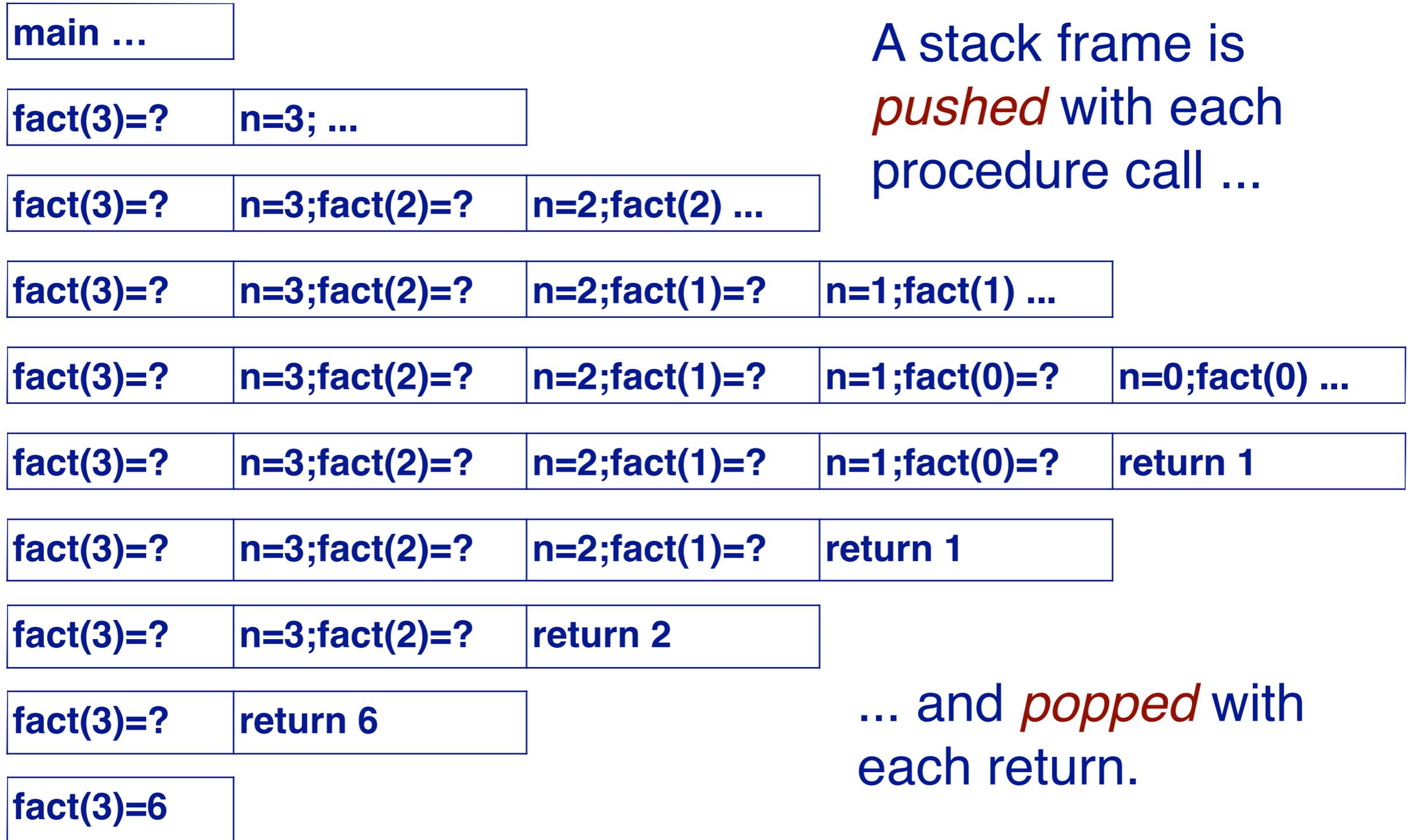
Practically all programming languages use a run-time stack:

```
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 ...



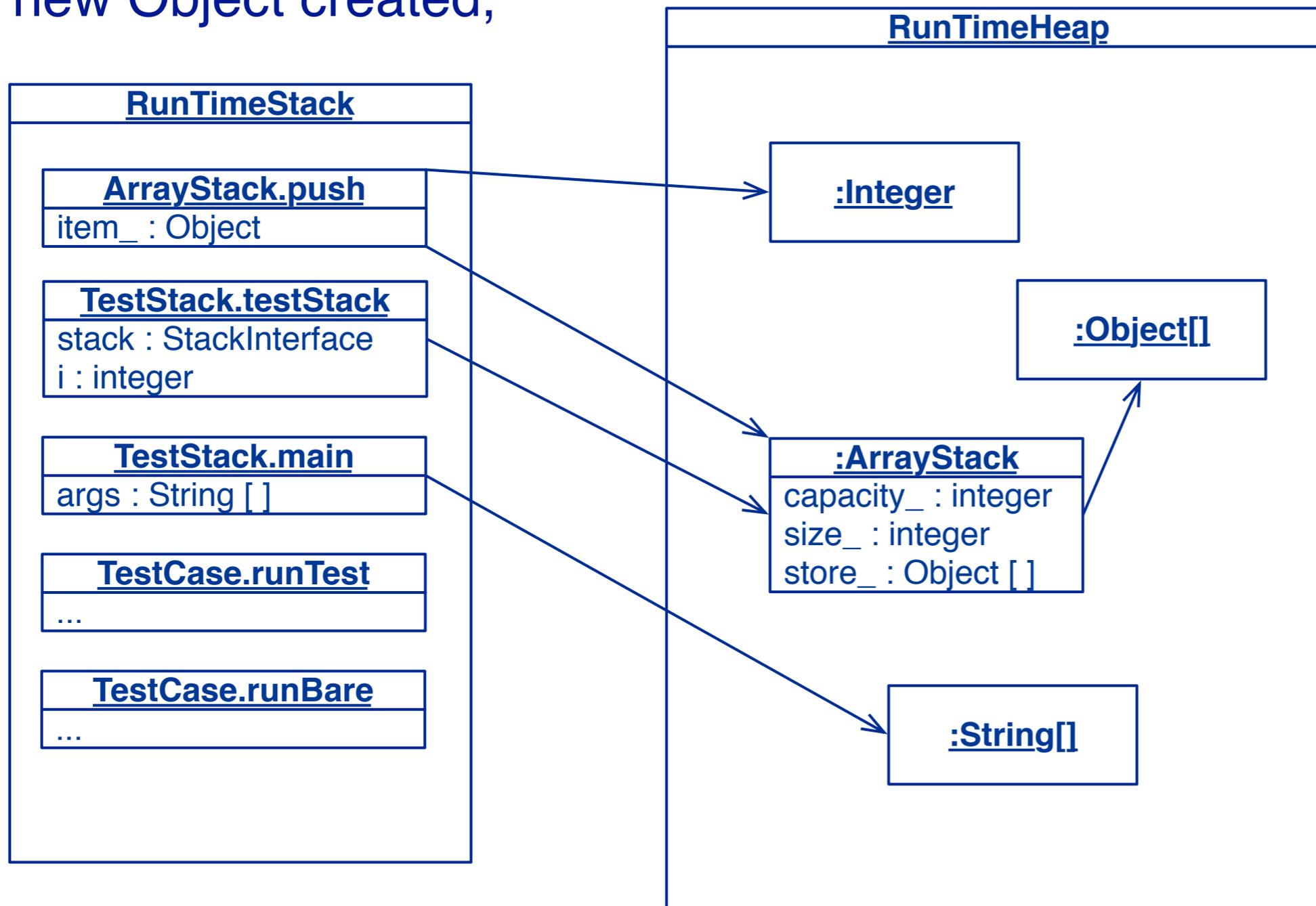
A stack frame is *pushed* with each procedure call ...

... 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 and other tools

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 IntelliJ

```
45  
46    
47    
48  
49  
50  
51 
```

```
@Test  
public void pushPopOneElement() {  
    stack.push( item: "a");  
    stack.pop();  
    assertTrue(stack.isEmpty());  
    assertEquals( expected: 0, stack.size());  
}
```

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 IntelliJ

The screenshot shows the IntelliJ IDEA IDE with a project named "p2-GenStack". The code editor displays the `ArrayStack.java` file. The `push` method is highlighted, and the debugger is paused at line 37, which contains the assertion `assert this.top() == item; item: "a"`. The Variables panel shows the state of the `ArrayStack` object, including its `store` array, `capacity`, and `size`.

```
20     return size;
21 }
22
23 @Override
24 public void push(T item) {
25     if (size == capacity) {
26         grow();
27     }
28     store[++size] = item; // NB: bug t
29     assert this.top() == item; item: "a"
30     assert invariant();
31 }
```

Debugger: `ArrayStackTest.pushPopOneElement`

Frames:

- "main"@1 in group "main": RUNNING
- push:37, ArrayStack (p2.genstack)
- pushPopOneElement:47, LinkStackTest (p2.genstack)
- invoke0:-1, NativeMethodAccessorImpl (jdk.internal.reflect)
- invoke:64, NativeMethodAccessorImpl (jdk.internal.reflect)
- invoke:43, DelegatingMethodAccessorImpl (jdk.internal.reflect)
- invoke:564, Method (java.lang.reflect)
- invokeMethod:628, ReflectionUtils (org.junit.platform.commons.util)
- invoke:117, ExecutableInvoker (org.junit.jupiter.engine.execution)
- lambda\$invokeTestMethod\$7:184, TestMethodTestDescriptor (org.junit.jupiter.engine.descriptor)
- execute:-1, TestMethodTestDescriptor\$\$Lambda\$246/0x0000000800be7d30 (org.junit.jup)
- execute:73, ThrowableCollector (org.junit.platform.engine.support.hierarchical)
- invokeTestMethod:180, TestMethodTestDescriptor (org.junit.jupiter.engine.descriptor)
- execute:127, TestMethodTestDescriptor (org.junit.jupiter.engine.descriptor)
- execute:68, TestMethodTestDescriptor (org.junit.jupiter.engine.descriptor)
- lambda\$executeRecursively\$5:135, NodeTestTask (org.junit.platform.engine.support.hierarc)

Variables:

- this = {ArrayStack@1654}
- store = {Object[2]@1656}
- Not showing null elements
- 1 = "a"
- capacity = 2
- size = 1
- item = "a"
- size = 1
- store = {Object[2]@1656}

Event Log: 1 Event Log

master

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.

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);  
}
```

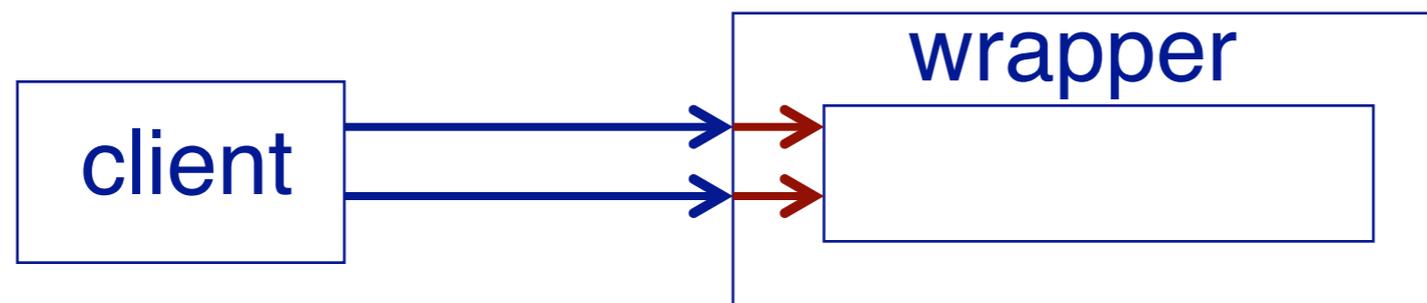
If we change our programs to work with the Java Stack, we won't be able to work with our own Stack implementations ...

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?*

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.empty(); }
    public int size() { return stack.size(); }
}
```

 Do you see any flaws with our wrapper class?

A contract mismatch

But running the test case yields:

```
org.opentest4j.AssertionFailedError: Unexpected exception type thrown ==> expected: <java.lang.AssertionError> but was: <java.util.EmptyStackException>
+ <3 internal calls>
+ at p2.genstack.LinkStackTest.emptyTopFails(LinkStackTest.java:29) <19 internal calls>
+ at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal calls>
+ at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <21 internal calls>
Caused by: java.util.EmptyStackException Create breakpoint
at java.base/java.util.Stack.peek(Stack.java:101)
at p2.genstack.SimpleWrappedStack.top(SimpleWrappedStack.java:31)
+ at p2.genstack.LinkStackTest.lambda$emptyTopFails$0(LinkStackTest.java:29) <1 internal call>
... 54 more
```

 *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 and other tools

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 {
    protected Date startTime; // Abstract from the
                               // 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>10M pushes</i>	<i>10M pops</i>
p2.stack.LinkStack	909	98
p2.stack.ArrayStack	342	24
p2.stack. WrappedStack	816	529
java.util.Stack	253	175

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

Every time I run these benchmarks on a different machine with a different JVM I get a different result!

Roadmap



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

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

...

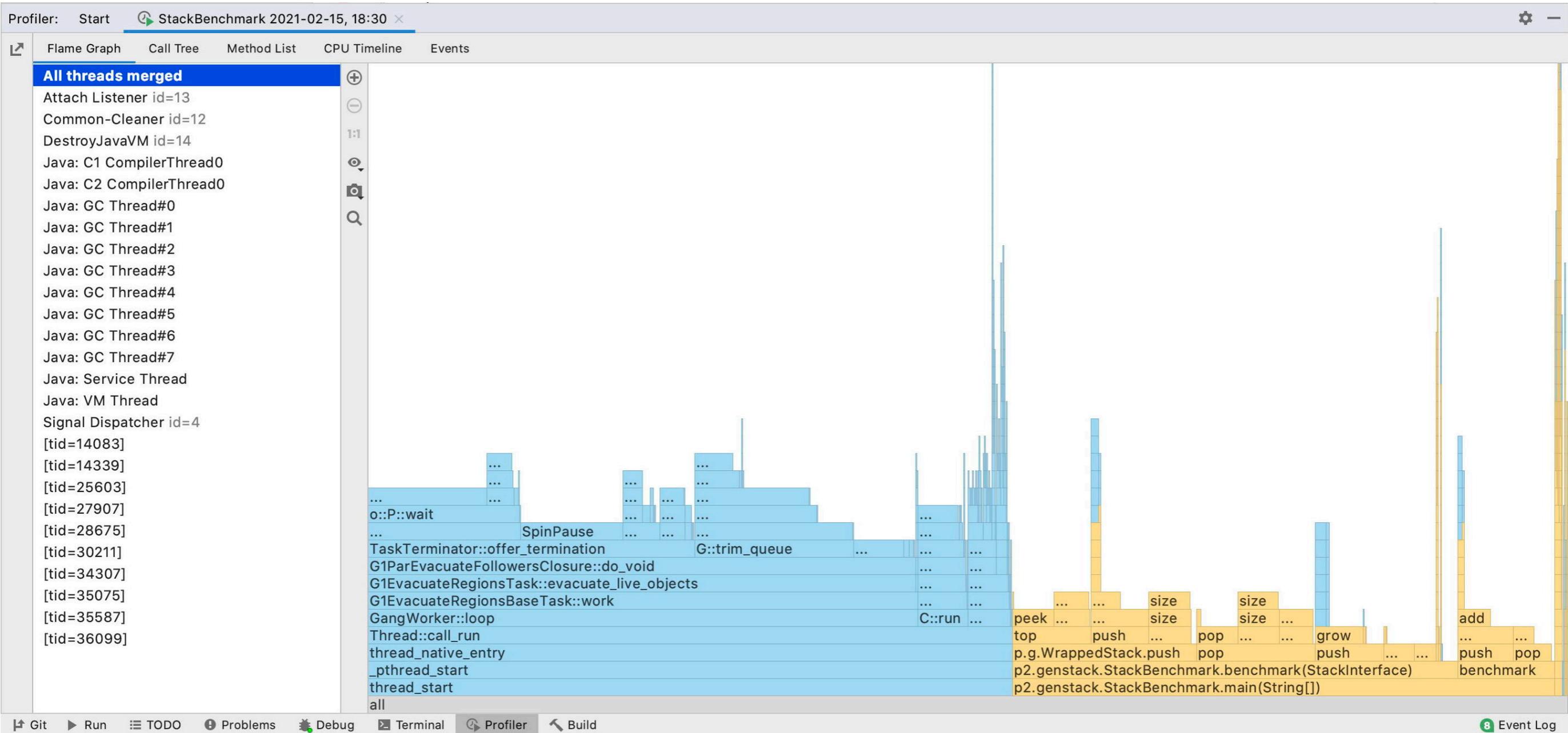
To use this, simply set the runtime (vm) flag `-Xprof`

IntelliJ profilers

The screenshot shows the IntelliJ IDEA IDE with a project named 'p2-GenStack [P2Stack]'. The left sidebar displays the project structure, including folders like '.idea', 'bin', and 'src', and various classes such as 'ArrayStack', 'LinkStack', 'ParenMatch', 'SimpleWrappedStack', 'StackBenchmark', 'StackInterface', 'Timer', 'WrappedStack', and 'WrappedStackTest'. The main editor window shows the source code for 'StackBenchmark.java'. A context menu is open over the line `static int iterations = 10000000; // 10M`. The menu items include 'New', 'Cut', 'Copy', 'Copy Path...', 'Paste', 'Path From Repository Root', 'Find Usages', 'Analyze', 'Refactor', 'Add to Favorites', 'Browse Type Hierarchy', 'Reformat Code', 'Optimize Imports', 'Delete...', 'Build Module 'P2Stack'', 'Run 'StackBenchmark.main()'', 'Debug 'StackBenchmark.main()'', 'More Run/Debug', 'Open in Right Split', and 'Open In'. The 'More Run/Debug' sub-menu is open, showing options like 'Run 'StackBenchmark.main()' with Coverage', 'Run 'StackBenchmark.main()' with 'CPU Profiler'', 'Run 'StackBenchmark.main()' with 'Allocation Profiler'', and 'Run 'StackBenchmark.main()' with 'Java Flight Recorder'.

```
5  /**
6  * Run timing benchmarks for various Stack implementations.
7  * @author Oscar.Nierstrasz@acm.org
8  * @version $Id: StackBenchmark.java 17003 2008-03-13 10:20:41Z oscar $
9  */
10 public class StackBenchmark {
11     static int iterations = 10000000; // 10M
12
13     static public void main(String args[]) {
14         benchmark(new LinkStack<Integer>());
15         // benchmark(new BrokenArrayStack<Integer>());
16         benchmark(new SimpleWrappedStack<Integer>());
17         benchmark(new WrappedStack<Integer>());
18         benchmark(new ArrayStack<Integer>());
19     }
20
21     /**
22     * Makes to do a large number of pushes/pops.
23     */
24     private void benchmark(StackInterface<Integer> stack) {
25         Timer timer = new Timer();
26         timer.start();
27         for (int i = 0; i < iterations; i++) {
28             stack.push("pushes");
29         }
30         timer.stop();
31         System.out.println("Timing " + stack.getClass().getName());
32     }
33 }
```

IntelliJ profile data



A modern profiler typically offers a graphical interface to navigate and explore the profile data.

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

Running tests with coverage

The screenshot displays the IntelliJ IDEA IDE interface during a test run with coverage. The main editor shows the `parenMatch` method in `ParenMatch.java`. The code is as follows:

```
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)); // Autoboxed to Character
        } else {
            if (isRightParen(c)) {
                // empty stack => missing left paren
                if (stack.isEmpty()) { return false; }
                if (stack.top().equals(c)) { // Autoboxed
                    stack.pop();
                } else { return false; } // mismatched paren
            }
        }
    }
    return stack.isEmpty(); // not empty => missing right paren
}
```

The coverage tool shows that 33% of classes and 27% of lines are covered in the scope of 'all classes in scope'. The table below summarizes the coverage data for the `p2.genstack` element:

Element	Class, %	Method, %	Line, %
p2.genstack	33% (4/12)	30% (23/...	27% (69/...

The Run tool window at the bottom shows the test results for `ParenMatchTest`, which passed all 7 tests in 14 ms. The output includes the IntelliJ IDEA coverage runner command and various configuration options:

```
Tests passed: 7 of 7 tests - 14 ms
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
p2\genstack\.*
exclude patterns:
Class transformation time: 0.017776063s for 670 classes or 2.653143731343284E-5s per class
```

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.

Other tools

- > FindBugs — static analysis of Java code
- > QuickCheck — generates tests
 - versions exist for many languages, including Java
- > Selenium — framework for testing web applications
- > Eclipse Memory Analyzer — analyzes Java heap to find memory leaks
- > VisualVM — visualizes memory consumption, thread usage, and many other features

What you should know!

-  *What is a regression test? Why is it important?*
-  *What strategies should you apply to design a test?*
-  *How does test-driven development work?*
-  *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?*



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/>