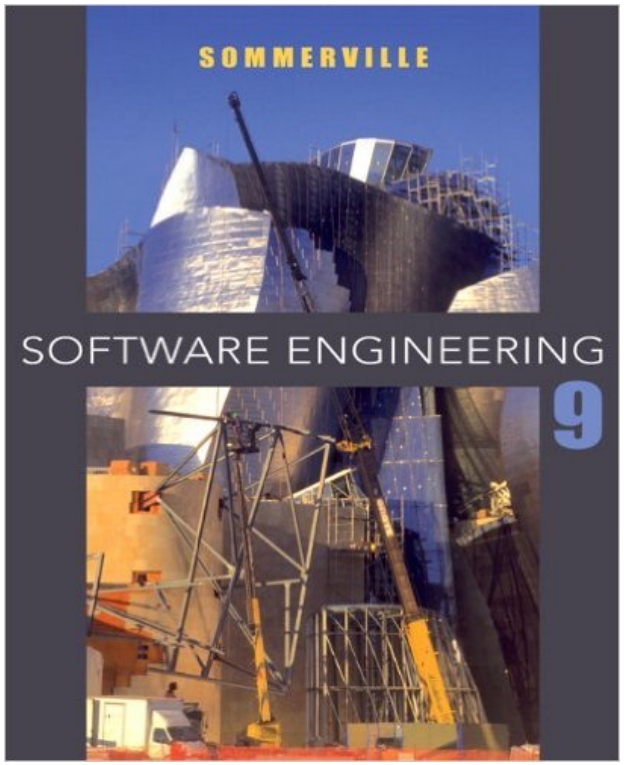
Testing and Debugging

Alexandre Bergel
http://bergel.eu
30/08/2017

Source

I. Sommerville, Software Engineering, Addison-Wesley, 9th Edn., 2015.

www.eclipse.org



Roadmap

1. Testing — definitions and strategies
2. Understanding the run-time stack and heap
3. Debuggers

Roadmap

1. **Testing — definitions and strategies**
2. Understanding the run-time stack and heap
3. Debuggers

Testing

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

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” functionalities

- every interface (black-box testing)
- all boundary situations
- every feature
- every line of code (white-box testing)
- everything that can conceivably go wrong!

Regression testing

Writing tests 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 and Correctness

This has a number of serious consequences

xUnit works well for testing your tareas and most industrial projects

However, critical software systems (e.g., medical, nuclear, air industry), software drivers requires complementary or different techniques

- Validation using Coq, FramaC, Esterel, …
- Mathematical proofs of a software, including theorem, offer guaranties that unit tests cannot offer
- But such mathematical proofs are much harder to write!

StackInterface

Interfaces let us *abstract* from concrete implementations:

```
public interface StackInterface {
   public boolean isEmpty();
   public int size();
   public void push(Object item);
   public Object top() ;
   public void pop();
}
```

How can clients accept multiple implementations of an Abstract Data Type?

Make them depend only on an interface or an abstract class.

Interfaces in Java

Interfaces reduce coupling between objects and their clients:

A class can implement *multiple interfaces* ... but can only *extend one parent class*

Clients should *depend on an interface*, not an implementation ... so implementations do not need to extend a specific class

Define an interface for any *concept* will have more than one implementation

As soon as different classes offer the same contract, then using interface is necessary

Testing a Stack

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

```
public class LinkStackTest {
   protected StackInterface stack;

   @Before public void setUp() {
      stack = new LinkStack();
   }

   @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() {
   int size;
   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 implements StackInterface {
   private Object[] 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(Object 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 Object top() {
   if(this.isEmpty())
      throw new AssertionError("Cannot be empty”);
   return store[size-1];
}
public void pop() {
   if(this.isEmpty())
      throw new AssertionError("Cannot be empty”);
   size--;
}
```

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

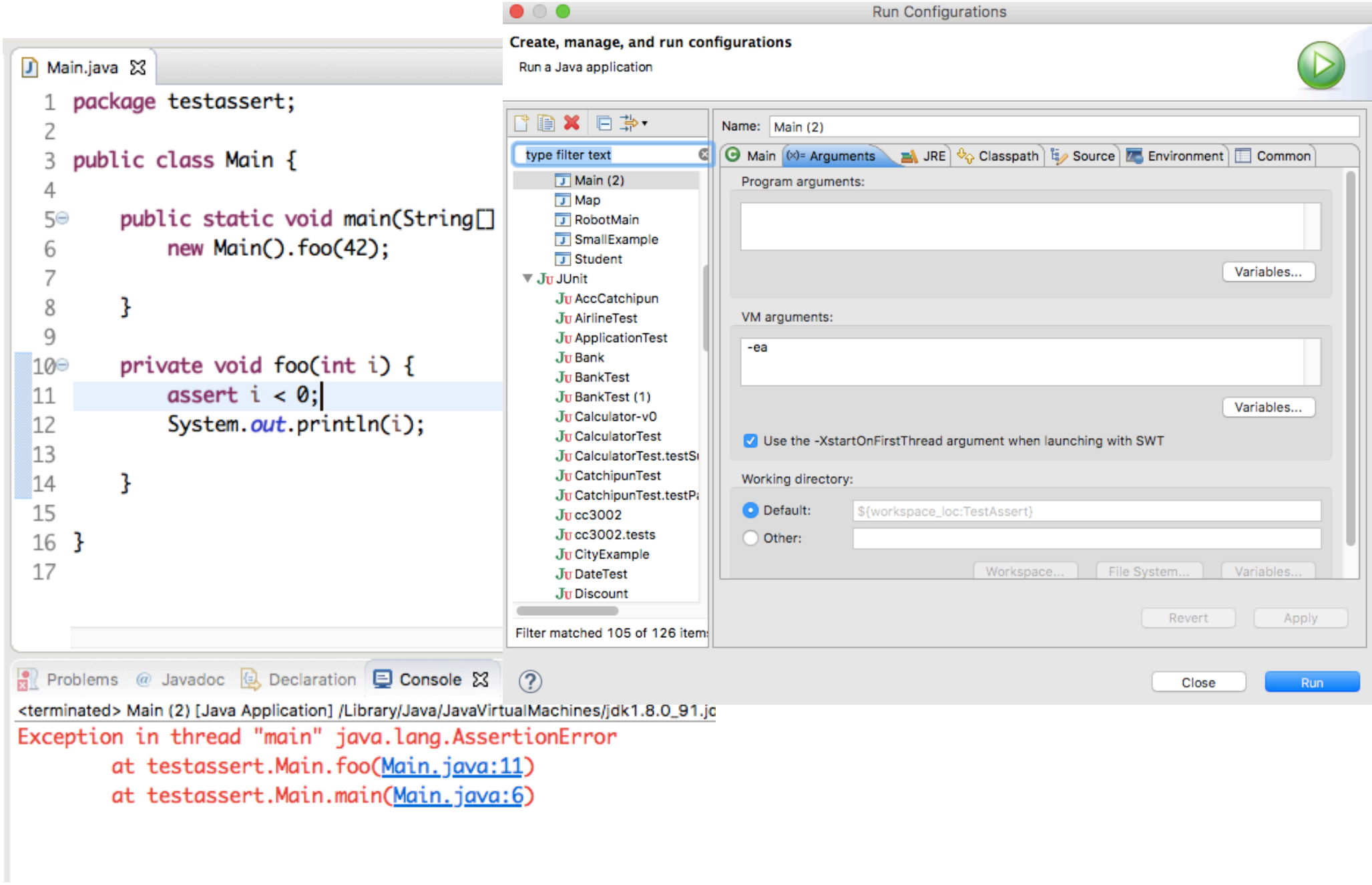
Checking pre-conditions

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

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

Equivalent notation: assert(Boolean) is available only with the proper compiler parameters

Enabling Assert on Eclipse



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();
   }
}
```

All the test methods defined in the superclass are inherited

Roadmap

1. Testing — definitions and strategies
2. **Understanding the run-time stack and heap**
3. Debuggers

Testing ArrayStack

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

```
java.lang.ArrayIndexOutOfBoundsException: 2
   at cc3002.stack.ArrayStack.push(ArrayStack.java:27)
   at cc3002.stack.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.

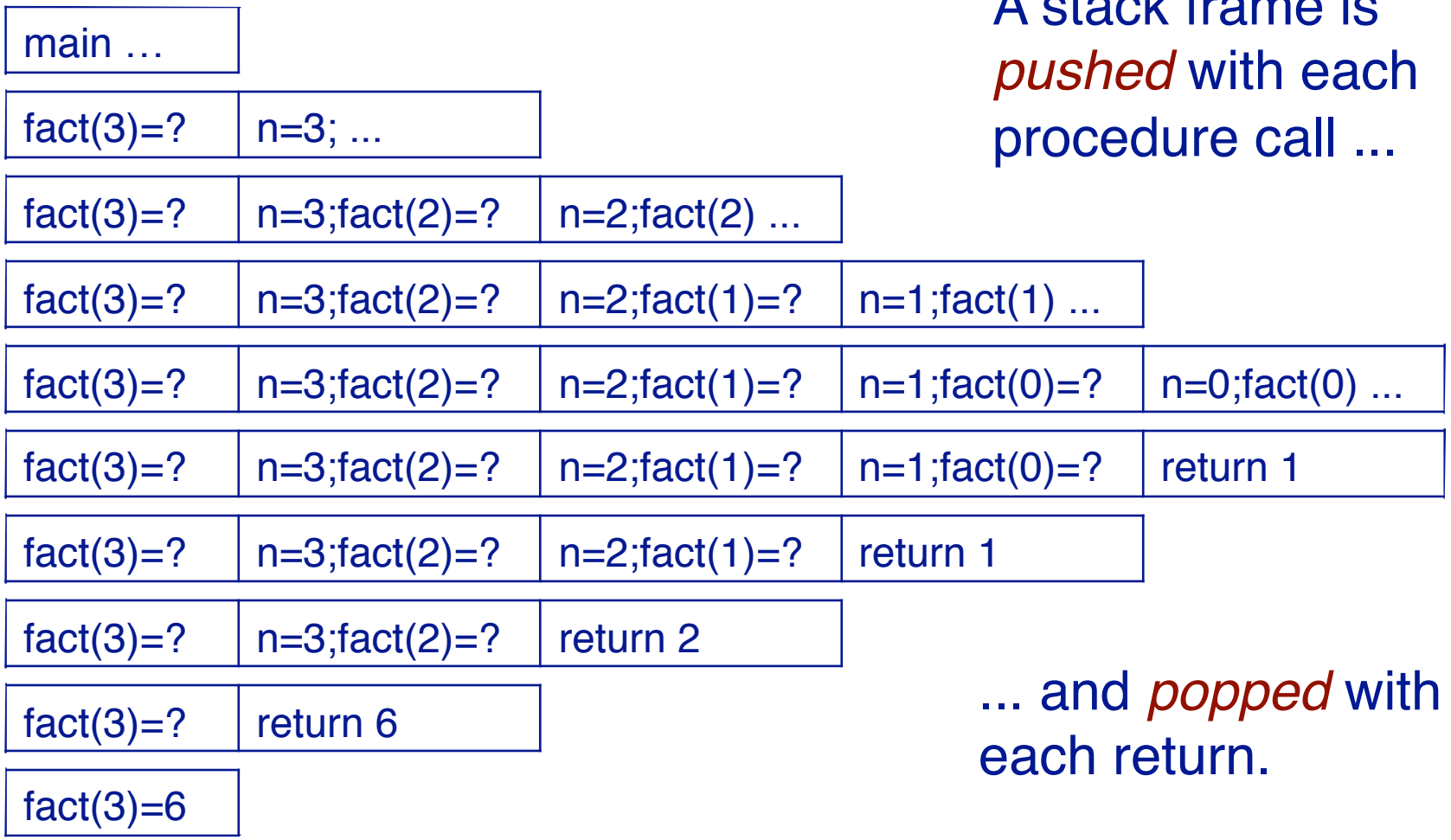
This *context* (AKA "*stack frame*") stores the arguments to the procedure and its local variables.

Practically all programming languages
use a run-time stack, in principle

The 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 ...

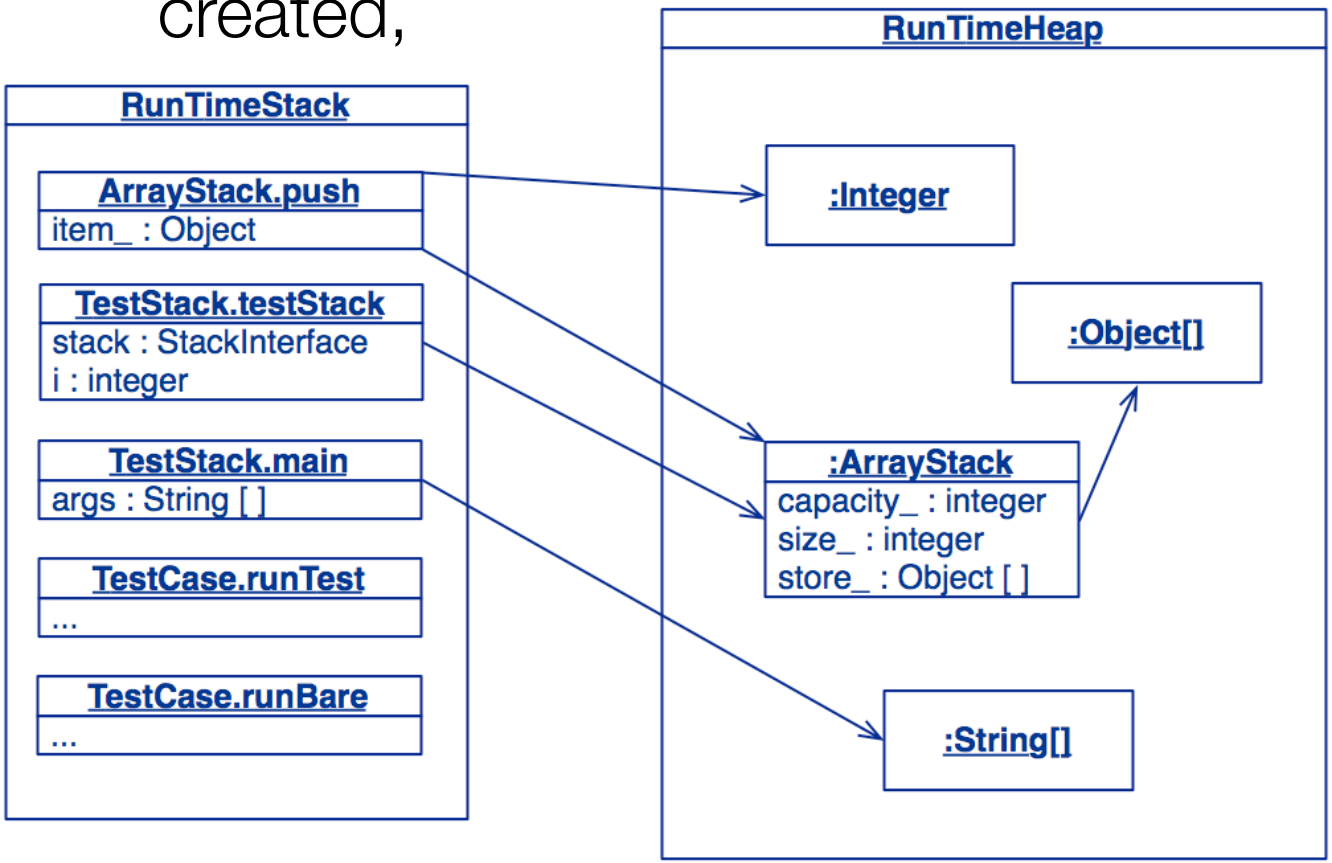


A stack frame is *pushed* with each procedure call ...

... 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

Cleaning the heap

A garbage collector is a form of automatic memory management

The basic principles of garbage collection are:

- Find data objects in a program that cannot be accessed in the future
- Reclaim the resources used by those objects

Objects that are not referenced anymore are simply removed from the memory

Roadmap

1. Testing — definitions and strategies
2. Understanding the run-time stack and heap
3. **Debuggers**

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.

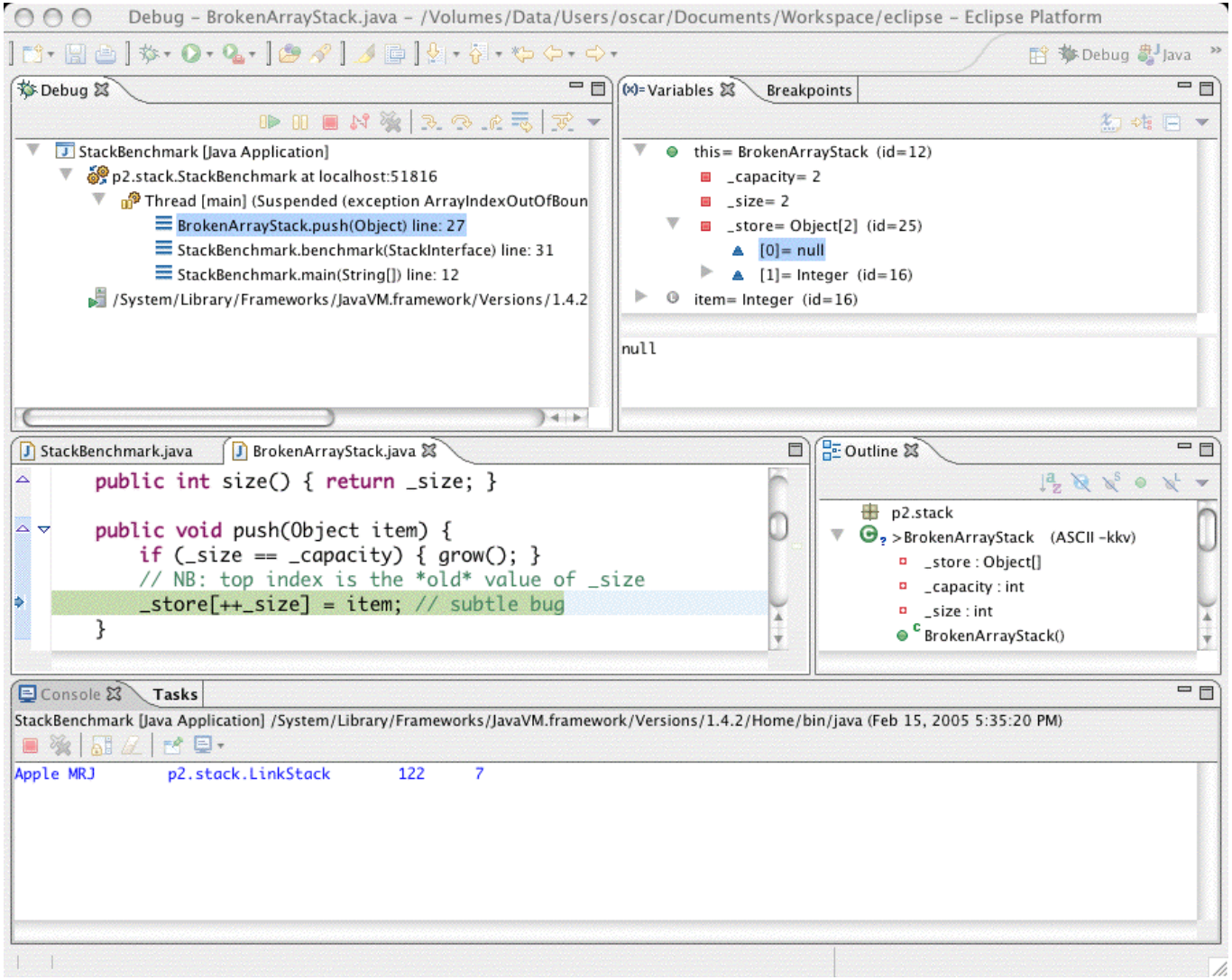
Classical debuggers are line-oriented (e.g., jdb); most modern ones are graphical.

When should you use a debugger?

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

Debugging in Eclipse

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



Debugging Strategy: development time

Develop tests as you program

- Develop *unit tests* to exercise all paths through your program
- You may apply *Design by Contract* to decorate classes with invariants and pre- and post-conditions, using the assert() keyword
- Use *assertions* (not print statements) to probe the program state
- After every modification, do regression testing!

Debugging Strategy: when testing

If errors arise during the unit tests execution

Use the test results to track down and fix the bug:

Test may be obsolete and need to be updated. The bug is therefore in the tests.

If the tests are right, the bug is therefore in the application.

If you can't tell where the bug is, *then use a debugger* to identify the faulty code

1 - identify and *add any missing tests*!

2 - fix the bug

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

Debugging Strategy: when running the application

If errors arise during the application execution

You first *need to understand* what is the running *scenario* that *caused* the *bug*

It is essential to be sure we got the problematic scenario. *Write a test that reproduce the scenario*

Run this new test to be sure that it fails

Fix the bug in the application

Run the test again to be sure we have fixed the bug

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(Object item) ... {
   if (size == capacity) { grow(); }
   store[size++] = item;
   assert(this.top() == item);
}
```

NB: perhaps it would be clearer to write:

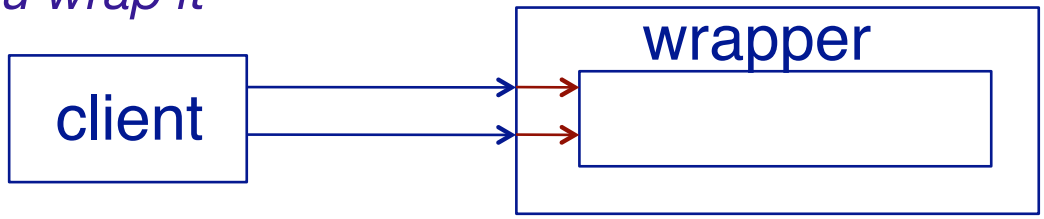
```
store[this.topIndex()] = item;
```

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 implements StackInterface {
   private Stack stack;
   public SimpleWrappedStack() { stack = new Stack(); }
   public boolean isEmpty() { return stack.empty(); }
   public int size() { return stack.size(); }
   public void push(Object item) { stack.push(item); }
   public Object top() { return stack.peek(); }
   public void pop() { stack.pop(); }
}
```

Stack is a standard Java class, contained in the package java.util

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 cc3002.stack.SimpleWrappedStack.top(SimpleWrappedStack.java:32)
   at cc3002.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 implements StackInterface {
   public Object top() {
      assert(!this.isEmpty());
      return super.top();
   }
   public void pop() {
      assert(!this.isEmpty());
      super.pop();
   }
   …
}
```

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?

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?

License

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

creative commons

COMMONS DEED

Attribution-ShareAlike 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.