Testing and Debugging

Alexandre Bergel
Nancy Hitschfeld

26/10/2020
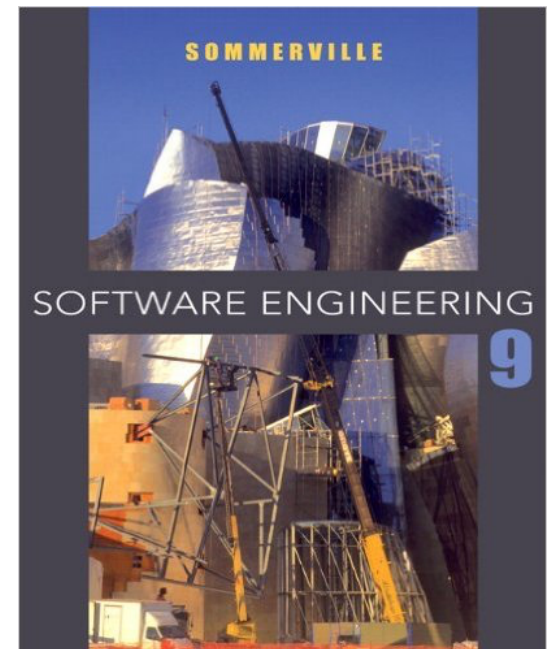
Source

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:

```java
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:

```java
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:

```java
@Test public void oneElement() {
    stack.push("a");
    assertFalse(stack.isEmpty());
    assertEquals(1, stack.size());
    stack.pop();
    assertEquals(0, 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?
With JUnit4, you need to write the following:

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


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

Check that failures are caught

How do we check that an assertion fails when it should?
With JUnit5, you need to write the following:

```java
@Test
public void emptyTopFails() {
   boolean shouldPass = false;
   try {
      stack.top();
   }
   catch(AssertionError e) {
      shouldPass = true;
   }
   assertTrue(shouldPass);
}
```

```java
@Test
public void emptyRemoveFails() {
   boolean shouldPass = false;
   try {
      stack.pop();
   }
   catch(AssertionError e) {
      shouldPass = true;
   }
   assertTrue(shouldPass);
}
```

Check that failures are caught

How do we check that an assertion fails when it should?
Alternatively, with JUnit5, you can write the following:

```java
@Test
public void emptyTopFails() {
  assertThrows(AssertionError.class,
      () -> stack.top() );
}

@Test
public void emptyRemoveFails() {
  assertThrows(AssertionError.class,
      () -> stack.pop() );
}
```

ArrayStack

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

```java
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.

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

How would you implement the grow() method?

Making the Array grow

```java
private void grow() {
    if(capacity == 0) {
        capacity = 1;
        store = new Object[1];
        return;
    }

    Object[] newStore = new Object[capacity * 2];
    for(int i = 0; i < capacity; i++)
        newStore[i] = store[i];
    store = newStore;
}
```

Checking pre-conditions

```java
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

```java
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 IntelliJ

The keyword assert() is enabled per default.

To manually enable it, you need to provide the parameter `-ea` to the JVM when being run
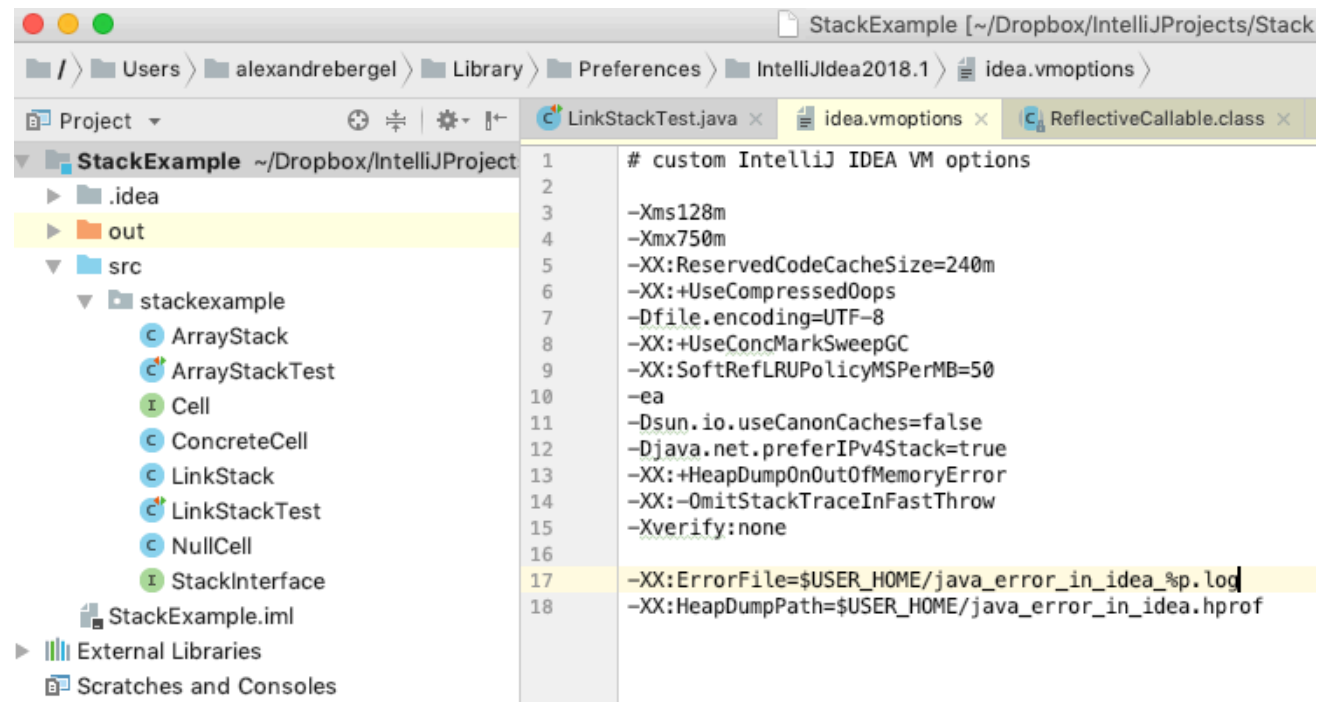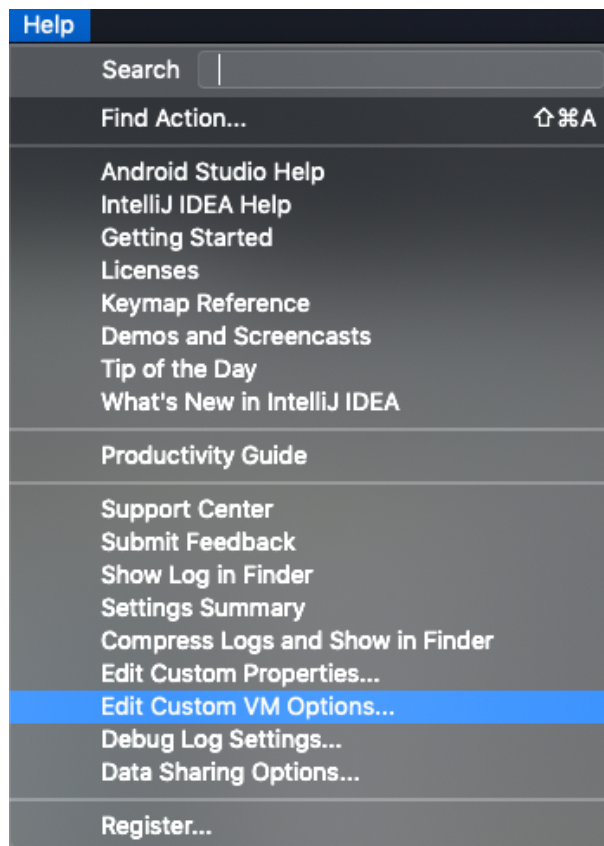
Adapting the test case

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

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

All the test methods defined in
the superclass are inherited

Roadmap

Testing ArrayStack

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

Testing ArrayStack

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

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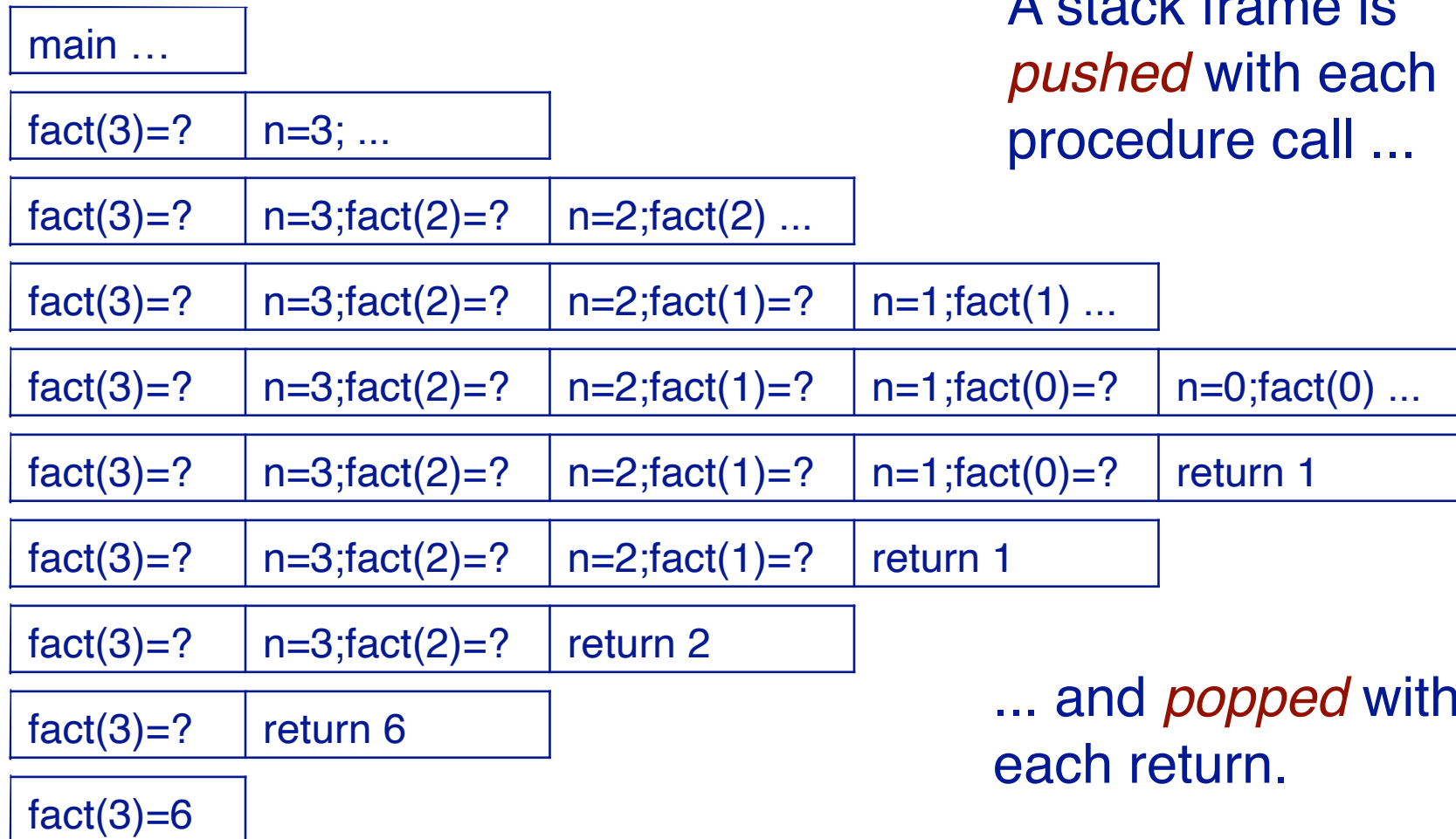
Practically all programming languages use a run-time stack, in principle

The Run-time Stack

```java
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 ...

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				

... and *popped* with each return.

The Stack and the Heap

The Heap grows with each new Object created,
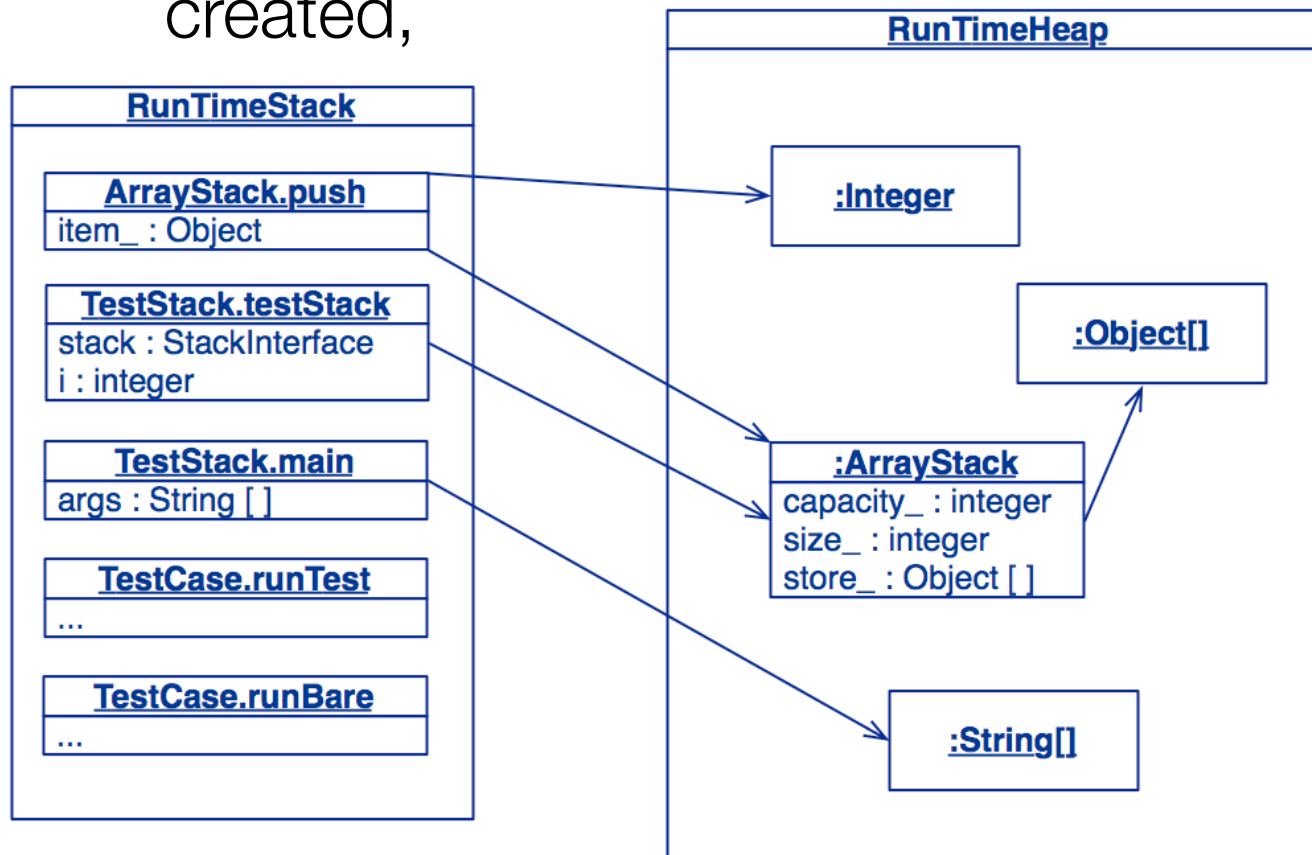
and shrinks when Objects are garbage-collected

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 IntelliJ

The stack indicating the error is:
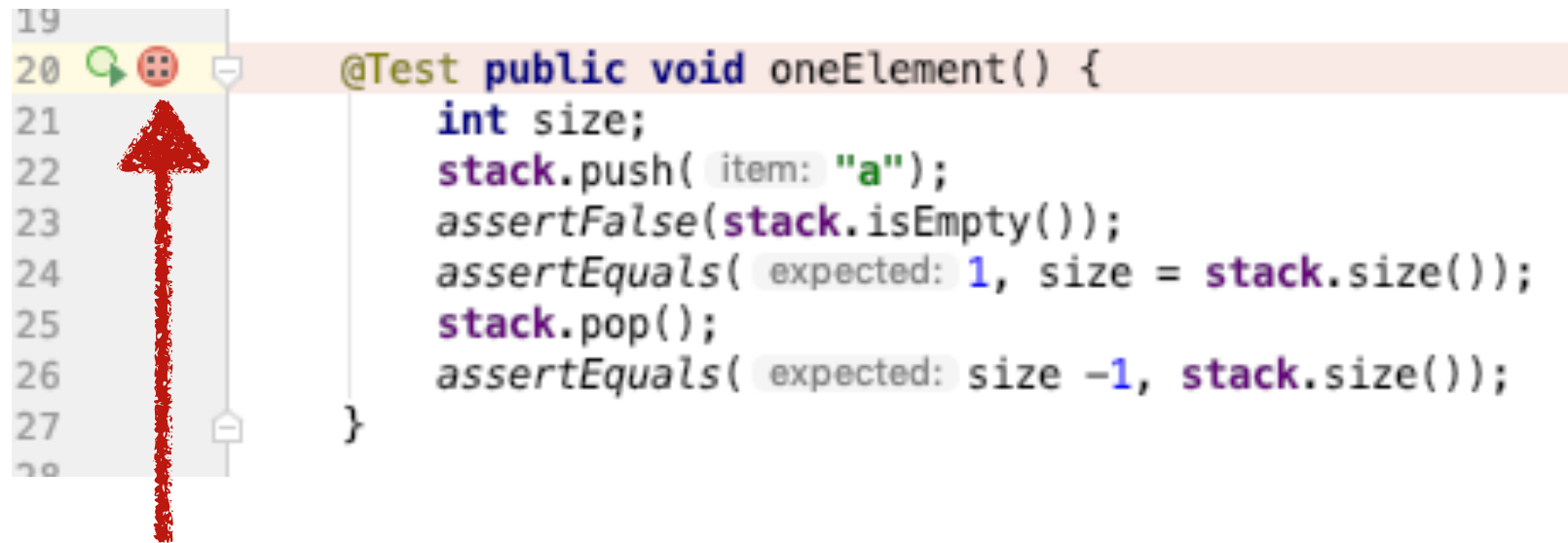
```
java.lang.ArrayIndexOutOfBoundsException: 1

    at stackexample.ArrayStack.push(ArrayStack.java:19)
    at stackexample.LinkStackTest.oneElement(LinkStackTest.java:22)
```

We see that the error is (indirectly) caused by `oneElement()`

We could trace the execution (i.e., executing step by step) starting from `oneElement()`
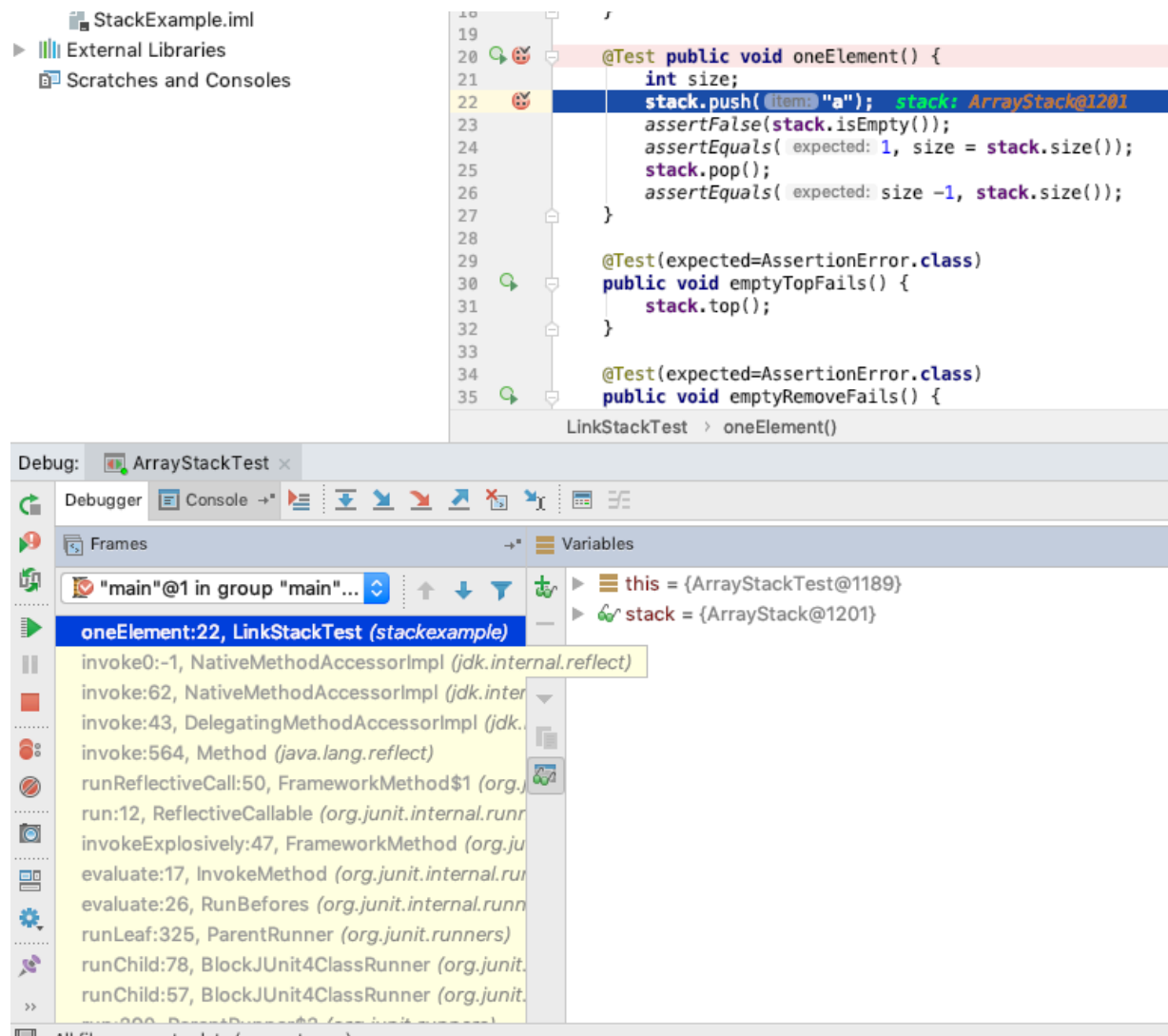
Debugging in IntelliJ

```
19
20          @Test public void oneElement() {
21              int size;
22              stack.push( item: "a");
23              assertFalse(stack.isEmpty());
24              assertEquals( expected: 1, size = stack.size());
25              stack.pop();
26              assertEquals( expected: size -1, stack.size());
27          }
28
```
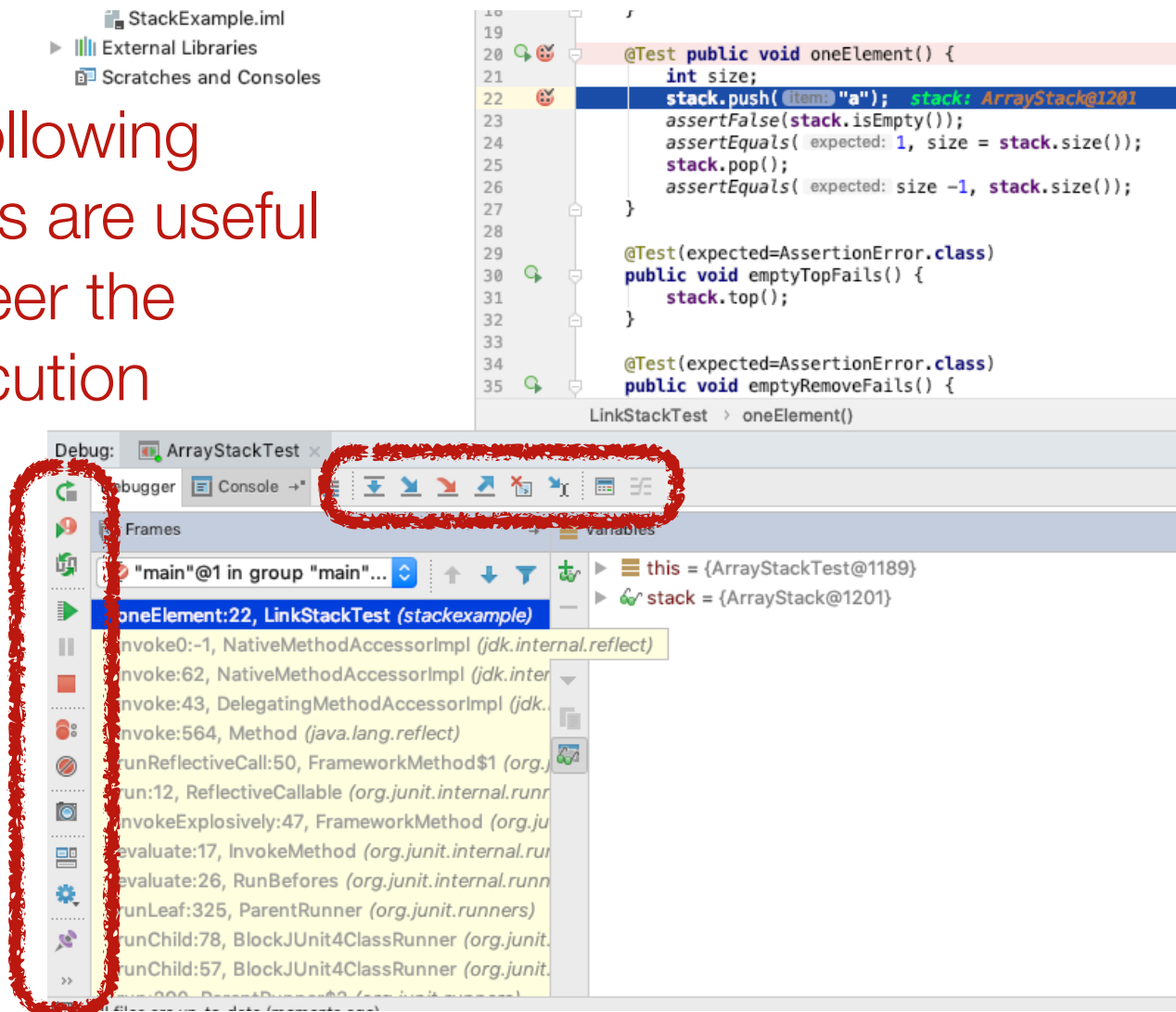
Clicking next to the method insert a breakpoint.
A *breakpoint* is a location in the source code that forces the runtime to suspend the execution when the program execution reaches that particular point

Debugging in IntelliJ
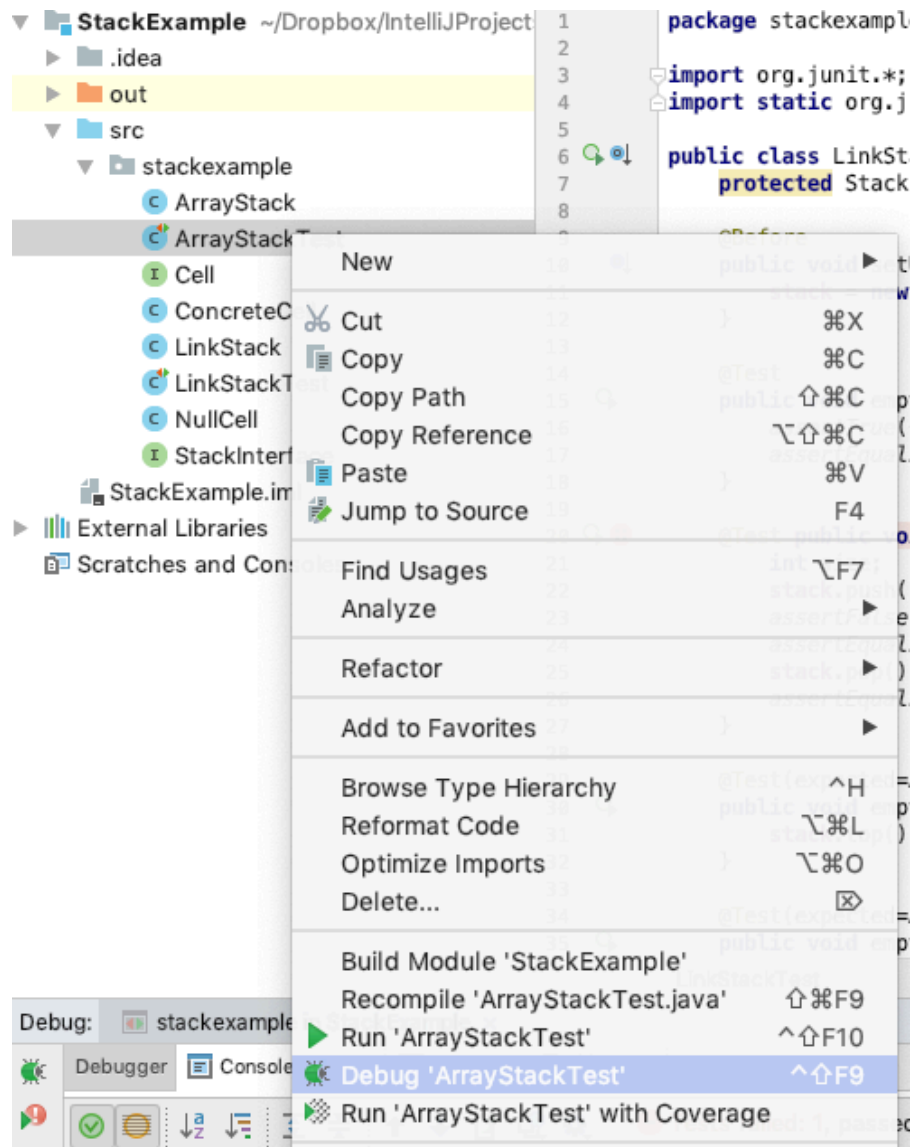
We can run again our test, but in a debug mode

Debugging in IntelliJ

Debugging in IntelliJ

The following commands are useful to steer the execution

Debugging in IntelliJ

The state of the
application
can be inspected

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);
}
```

item	1
	0

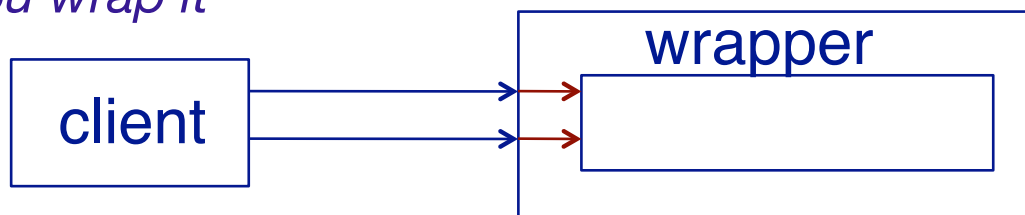NB: perhaps it would be clearer to write:

```
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:

```java
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:

```java
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*!

```java
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

Original version of this lecture is from Prof. Oscar Nierstrasz

dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

f ⃝ in 🐦 / DCCUCHILE