Test Report - ArrayStack

Tim Pizey
Oxford University
Tim.Pizey@gmail.com

Page count 21

March 9, 2013

Part I
Test Report

1 Test Environment

The tests should be run on all target hardware and operating systems, given below are the versions actually run on due to limitations of available testing setup :

Processor	OS	Memory	Java™Version
Intel Core 2 Duo	OSX 10.6.8	4 GB	HotSpot(TM) 64-Bit version 1.6.0_37
Intel i7	Ubuntu 12.0.4	8 GB	OpenJDK 64-Bit version 1.7.0_09

Current versions of Java™are able to simulate earlier versions and so the tests have been run with simulated Java™versions 1.3, 1.4, 1.5, 1.6 and 1.7.

The tests were run with JUnit version 3.8.1.

2 Additional Test Machinery

The defects in the SUT prevent the full tests being run successfully, as **JUnit** fails at the first failing assertion in a test. These defects would normally be fixed, to enable the next defect to be detected. For the purposes of this example a new class `FixedStackArray` is supplied on which all the tests run to completion, in a real world development the fixes would be applied directly to the SUT.

Some machinery is introduced to enable the tests to be run on both classes, exposing another design smell: the method `atPosition(int p)` is not specified in the interface `jdslcomp.simple.api.Stack` so we cannot test cleanly to the `Stack` interface.

3 Test Design

The tests are designed using a combination of categorisation, analysis and inspection .

3.1 Method Categories

Creation	Transforming	Non-transforming
Default constructor	push	size
Sized constructor	pop	isEmpty
		top
		atPosition
		toString

Table 1: Method Categories

3.2 Analysis of Argument Types

The semantic type of the `size` argument to the constructor is **Positive Integer Greater Than Zero** which ranges from `1` to `Infinity` however it is implemented using the Java™primitive `int` which ranges from `Integer.MIN_VALUE` to `Integer.MAX_VALUE` ie it includes negative integers and zero and it has an upper bound. We need to test that the SUT correctly handles all cases where the range of the implementation does not coincide with the range of the semantic type.

Similarly the `position` argument has the same semantic type, **positive Integer Greater Than Zero**, but is implemented as `int` so the same checks need to be made. In addition we know that the SUT is implemented using a java `Object` array which uses a zero based index so we need to test for a mis-match between the `position` in the abstract `Stack` and the concrete `index` of the array.

3.3 State Transition Analysis

The state model in Figure 1 shows the effect on the System Under Test (SUT) of the two state transforming methods `push` and `pop`, the creation and non-transforming methods are excluded for clarity.

Edge	Operation	Before	After	Transition	Return	testName
r1	pop	S1	S1	no	Exception	testPopFromEmpty
r2	push	S1	S2	yes	void	testPushToEmpty
r3	pop	S2	S1	yes	value	testPopEmptying
r4	push	S2	S2	no	void	testPushCentral
r5	pop	S2	S2	no	value	testPopCentral
r6	push	S2	S3	yes	void	testPushFilling
r7	pop	S3	S2	yes	value	testPopFromFull
r8	push	S3	S3	no	Exception	testPushToFull

Table 2: State Table

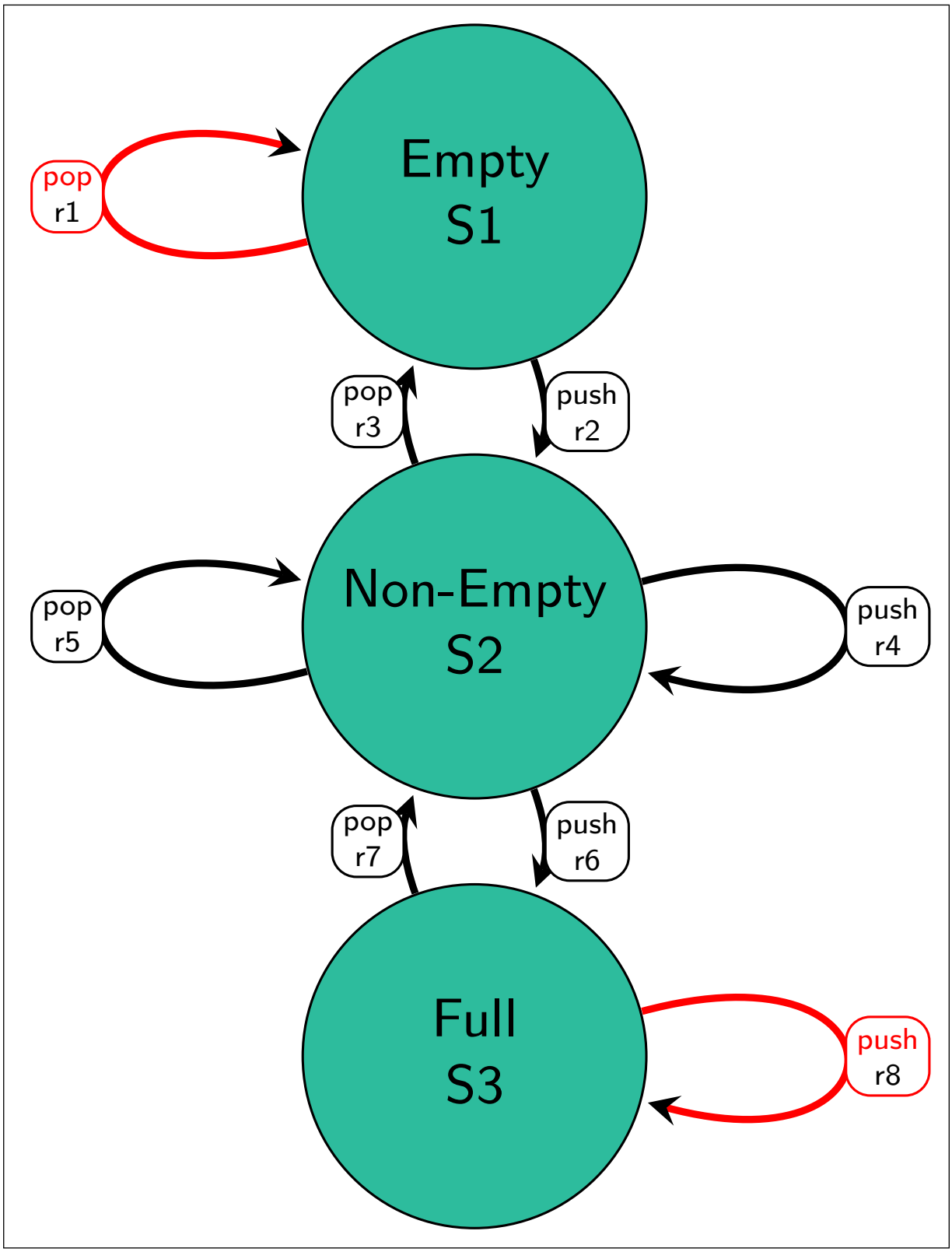


Figure 1: Stack States, exception transitions are coloured in red

3.4 Thread Safety Analysis

To use the `ArrayStack` in a multi-threaded setting it is necessary to `synchronize` on the stack. It is not sufficient to `synchronize` the `push` and `pop` methods as shown in `testUnsafeThreadedAccess()`.

3.5 Memory Leak Analysis

findbugs does not report any anti-patterns.

3.6 Failure Modes Analysis

Two modes of failure have been highlighted: memory exhaustion, where the initial size of the stack is greater than the available memory, exposed by `testMaxConstructor()`, the correct exception is thrown.

The second failure, highlighted in `testUnsafeThreadedAccess()`, arises from misuse: failure to `synchronize` upon the stack object in a multithreaded environment will lead to thread clashes and unexpected results. This is not a defect and cannot be changed but should be documented.

3.7 Security Analysis

No security issues are considered at this level.

4 Tests

Listing 1: Tests

```
package ArrayStack;

import jdslcomp.simple.api.Stack;
import jdslcomp.simple.api.StackEmptyException;
import jdslcomp.simple.api.StackFullException;
import jdslcomp.simple.api.StackOutOfScopeException;
import junit.framework.TestCase;

/**
 * Tests for the ArrayStack.
 *
 * @author timp
 */
public abstract class ArrayStackSpec extends TestCase {

  /** Our ArrayStack has an additional method. */
```

```
public interface InspectableStack extends Stack {
  Object atPosition(int position);
}

public class Sut implements InspectableStack {
  Stack stack;

  public Sut(Stack s) {
    stack = s;
  }

  @Override
  public int size() {
    return stack.size();
  }

  @Override
  public boolean isEmpty() {
    return stack.isEmpty();
  }

  @Override
  public Object top() throws StackEmptyException {
    return stack.top();
  }

  @Override
  public void push(Object element) {
    stack.push(element);
  }

  @Override
  public Object pop() throws StackEmptyException {
    return stack.pop();
  }

  /**
   * Machinery to work around method not included
   * in Stack interface.
   */
  @Override
  public Object atPosition(int position) {
    if (stack instanceof ArrayStack) {
      return ((ArrayStack) stack).atPosition(position);
```

```
    } else if (stack instanceof FixedArrayStack) {
      return ((FixedArrayStack) stack).atPosition(position);
    } else
      throw new RuntimeException("Unexpected class "
          + stack.getClass().getName());
  }

  @Override
  public String toString() {
    return stack.toString();
  }
}

// Creation Tests
public void testNegativeSizeConstructor() {
  try {
    getSizedStack(-1);
  } catch (NegativeArraySizeException e) {
    fail(failMessage(e)); // bug
  } catch (IllegalArgumentException e) {
    e = null; // expected
  }
}

public void testZeroSizedConstructor() {
  @SuppressWarnings("unused")
  Stack zero = null;
  try {
    zero = getSizedStack(0);
    fail("Should have bombed with IllegalArgumentException");
  } catch (IllegalArgumentException e) {
    e = null; // expected
  }
}

public void testOneConstructor() {
  Sut s = getSizedStack(1);
  exerciseSized(s, 1);
}

public void testThreeConstructor() {
  Sut s = getSizedStack(3);
  exerciseSized(s, 3);
}
```

```
public void testCapacityConstructor() {
  Sut s = getSizedStack(ArrayStack.CAPACITY);
  exerciseSized(s, ArrayStack.CAPACITY);
}

// Possible design flaw,
public void testMaxConstructor() {
  try {
    getSizedStack(Integer.MAX_VALUE);
    fail("Should have bombed");
  } catch (OutOfMemoryError e) {
    e = null; // expected
  }
}

public void testDefaultConstructor() {
  Sut s = getDefaultStack();
  exerciseSized(s, ArrayStack.CAPACITY);
}

private void exerciseSized(Sut s, int size) {
  assertTrue(s.isEmpty());
  assertEquals("", s.toString());
  try {
    s.top();
  } catch (StackEmptyException e) {
    e = null;
  }
  StringBuffer expected = new StringBuffer();
  for (int i = 0; i < size; i++) {
    s.push(new Integer(i));
    assertFalse(s.isEmpty());
    assertEquals(new Integer(i), s.top());
    assertEquals(s.top(), s.atPosition(i + 1));
    try {
      s.atPosition(i + 2);
    } catch (StackOutOfScopeException e) {
      e = null;
    }
    try {
      s.atPosition(0);
    } catch (IllegalArgumentException e) {
      e = null;
```

```
      } catch (ArrayIndexOutOfBoundsException e) {
        fail(failMessage(e)); // bug
      }
      expected.append(i + " ");
      assertEquals(expected.toString().trim(), s.toString());
    }
    try {
      s.push("breaker");
    } catch (ArrayIndexOutOfBoundsException e) {
      fail("Should have bombed with StackFullException");
    } catch (StackFullException e) {
      e = null; // expected
    }

    for (int i = 0; i < size; i++) {
      s.pop();
    }
    try {
      s.pop();
    } catch (ArrayIndexOutOfBoundsException e) {
      fail("Should have bombed with StackEmptyException");
    } catch (StackEmptyException e) {
      e = null; // expected
    }
  }

  // Argument Tests

  public void testPushNull() throws Exception {
    Sut s = getDefaultStack();
    s.push(null);
    assertEquals(1, s.size());
    assertNull(s.top());
    assertNull(s.pop());
    assertEquals(0, s.size());
  }

  // Fails with ArrayIndexOutOfBounds
  public void testBadPosition() throws Exception {
    Sut s = getDefaultStack();
    try {
      s.atPosition(-1);
    } catch (IllegalArgumentException e) {
      e = null; //expected
```

```
    }
    try {
      s.atPosition(0);
    } catch (IllegalArgumentException e) {
      e = null; //expected
    }
    try {
      s.atPosition(1);
    } catch (StackOutOfScopeException e) {
      e = null; //expected
    }
  }

  // State Transition Tests

  public void testPopFromEmpty() {
    Sut s = getDefaultStack();
    try {
      s.pop();
    } catch (StackEmptyException e) {
      e = null;
    }
  }

  public void testPushToEmpty() {
    Sut s = getDefaultStack();
    s.push("1");
    assertEquals("1", s.toString());
    assertEquals("1", s.top());
    assertEquals(1, s.size());
    assertFalse(s.isEmpty());
    assertEquals("1", s.atPosition(1));
  }

  public void testPopEmptying() {
    Sut s = getDefaultStack();
    s.push("1");
    s.pop();
    assertEquals("", s.toString());
    try {
      s.top();
    } catch (StackEmptyException e) {
      e = null;
    }
```

```
    assertEquals(0, s.size());
    assertTrue(s.isEmpty());
  }

  public void testPushCentral() {
    Sut s = getDefaultStack();
    s.push("1");
    s.push("2");
    assertEquals("1 2", s.toString());
    assertEquals("2", s.top());
    assertEquals(2, s.size());
    assertFalse(s.isEmpty());
    assertEquals("1", s.atPosition(1));
    assertEquals("2", s.atPosition(2));
  }

  public void testPopCentral() {
    Sut s = getDefaultStack();
    s.push("1");
    s.push("2");
    s.pop();
    assertEquals("1", s.toString());
    assertEquals("1", s.top());
    assertEquals(1, s.size());
    assertFalse(s.isEmpty());
    assertEquals("1", s.atPosition(1));
  }

  public void testPushFilling() {
    Sut s = getSizedStack(1);
    s.push("1");
    assertEquals("1", s.toString());
    assertEquals("1", s.top());
    assertEquals(1, s.size());
    assertFalse(s.isEmpty());
    assertEquals("1", s.atPosition(1));
  }

  public void testPopFromFull() {
    Sut s = getSizedStack(1);
    s.push("1");
    s.pop();
    assertEquals("", s.toString());
    try {
```

```
      s.top();
    } catch (StackEmptyException e) {
      e = null;
    }
    assertEquals(0, s.size());
    assertTrue(s.isEmpty());
  }

  public void testPushToFull() {
    Sut s = getSizedStack(1);
    s.push("1");
    try {
      s.push("2");
    } catch (StackFullException e) {
      e = null;
    }
    assertEquals("1", s.toString());
    try {
      s.top();
    } catch (StackEmptyException e) {
      e = null;
    }
    assertEquals(1, s.size());
    assertFalse(s.isEmpty());
  }

  // Thread Safety Tests

  Sut as = getSizedStack(200);
  static boolean finished;
  static Error firstError;

  public void testThreadedAccess() throws Exception {
    finished = false;
    firstError = null;
    for (int i = 0; i < 100; i++)
      new PushPopper().start();
    while(!finished)
     Thread.sleep(10);
    if (firstError != null)
      throw firstError;
  }
  // Test fails, but is marked as unsafe usage
  public void testUnsafeThreadedAccess() throws Exception {
```

```
    finished = false;
    firstError = null;
    for (int i = 0; i < 100; i++)
      new UnsafePushPopper().start();
    while(!finished)
     Thread.sleep(10);
    if (firstError != null)
      throw firstError;
  }

  /** Thread safe, synchronized on stack. */
  private class PushPopper extends Thread {
    public void run() {
      for (int i = 0; i < 100; i++) {
        synchronized (as) {
          as.push("A");
          try {
            assertEquals("A", as.pop());
            assertTrue(valid(as));
          } catch (Throwable t) {
            firstError = new Error("Failed on iteration " + i, t);
            finished = true;
            Thread.currentThread().notifyAll();
            throw firstError;
          }
        }
      }
      finished = true;
    }
  }


  /** Thread unsafe, not synchronized on stack. */
  private class UnsafePushPopper extends Thread {
    public void run() {
      for (int i = 0; i < 2000; i++) {
        as.push("A");
        try {
          assertEquals("A", as.pop());
          assertTrue(valid(as));
        } catch (Throwable t) {
          firstError = new Error("Failed on iteration " + i, t);
          finished = true;
          throw firstError;
```

```
        }
      }
      finished = true;
    }
  }

  abstract Sut getDefaultStack();

  abstract Sut getSizedStack(int size);

  private String failMessage(Exception e) {
    return e.getClass().getName()
        + (e.getMessage() == null ? "" : ":" + e.getMessage());
  }

  public boolean valid(Sut s) {
    for (int i = 0; i < s.size(); i++) {
      if (s.atPosition(i + 1) == null)
        return false;
    }
    return true;
  }
}
```

Listing 2: ArrayStackTest

```
package ArrayStack;

public class ArrayStackTest extends ArrayStackSpec {

  @Override
  Sut getDefaultStack() {
    return new Sut(new ArrayStack());
  }

  @Override
  Sut getSizedStack(int size) {
    return new Sut(new ArrayStack(size));
  }

}
```

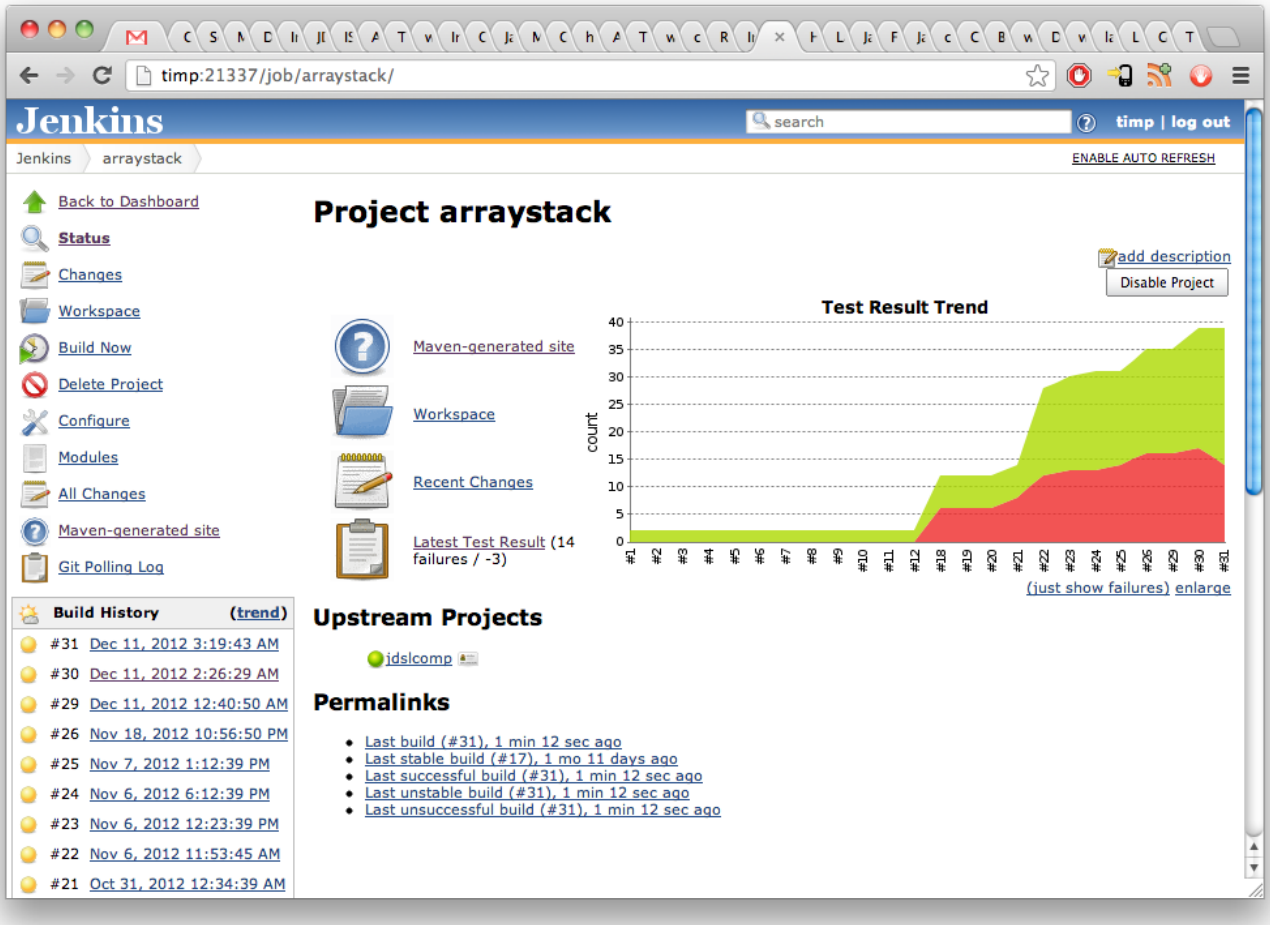
Listing 3: FixedArrayStackTest

```
package ArrayStack;
```

```
public class FixedArrayStackTest extends ArrayStackSpec {

  @Override
  Sut getDefaultStack() {
    return new Sut(new FixedArrayStack());
  }

  @Override
  Sut getSizedStack(int size) {
    return new Sut(new FixedArrayStack(size));
  }

}
```

5 Results

5.1 Jenkins Results



5.2 Static Analysis Reports

5.2.1 Cobertura

Cobertura shows an uncovered line in `push(Object o)` caused by defect AS02.

```
25          public void push(Object e) throws StackFullException {
26   32727    if (size() == CAPACITY)
27       0      throw new StackFullException("Push to a full stack");
28   32727    tos++;
29   32727    S[tos] = e;
30   32725  }
31
```

5.2.2 Findbugs

Findbugs discovers two performance issues which can be safely ignored.

ArrayStack.ArrayStack

Bug	Category	Details	Line	Priority
ArrayStack.ArrayStack.toString() invokes inefficient new String() constructor	PERFORMANCE	DM_STRING_VOID_CTOR	63	Medium
ArrayStack.ArrayStack.toString() concatenates strings using + in a loop	PERFORMANCE	SBSC_USE_STRINGBUFFER_CONCATENATION	65	Medium

5.2.3 Checkstyle

Checkstyle finds 40 style issues, from the trivial "File does not end with a newline", through design pattern checks "Method 'isEmpty' is not designed for extension - needs to be abstract, final or empty.".

Checkstyle needs to be configured carefully but can be useful, each issue is worth considering, even if to positively configure Checkstyle not to perform that check.

5.2.4 PMD

PMD found no issues.

5.2.5 CPD

Not surprisingly picked up the similarities between ArrayStack.java and FixedArayStack.java.

5.2.6 JDepend

JDepend produces a summary of its quality metrics, and an explanation of the code quality model it uses.

ArrayStack

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
1	2	0.0%	67.0%	33.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	ArrayStack.ArrayStack ArrayStack.FixedArrayStack	Default	java.lang jdslcomp.simple.api

5.3 Surefire Results

ArrayStackTest			
Test	Status	Reason	Time
testNegativeSizeConstructor	Failed	Threw NegativeArraySizeException	0.004
testZeroSizedConstructor	Failed	Should have bombed	0
testOneConstructor	Failed	Attempt to go beyond top of stack	0.004
testThreeConstructor	Failed	Attempt to go beyond top of stack	0
testCapacityConstructor	Failed	Attempt to go beyond top of stack	0
testMaxConstructor	Passed		0
testDefaultConstructor	Failed	Attempt to go beyond top of stack	0
testPushNull	Passed		0
testBadPosition	Failed	ArrayIndexOutOfBounds -2	0
testPopFromEmpty	Passed		0
testPushToEmpty	Failed	expected:<1> but was:<>	0
testPopEmptying	Passed		0
testPushCentral	Failed	expected:<1 2> but was:<2 >	0.001
testPopCentral	Failed	expected:<1> but was:<>	0
testPushFilling	Failed	expected:<1> but was:<>	0
testPopFromFull	Passed		0
testPushToFull	Failed	ArrayIndexOutOfBounds	0.001
testThreadedAccess	Passed		0.043
testUnsafeThreadedAccess	Failed	Failed on iteration 0	0.111

FixedArrayStackTest			
Test	Status	Reason	Time
testNegativeSizeConstructor	Passed		0
testZeroSizedConstructor	Passed		0
testOneConstructor	Passed		0
testThreeConstructor	Passed		0
testCapacityConstructor	Passed		3.098
testMaxConstructor	Passed		0.042
testDefaultConstructor	Passed		2.849
testPushNull	Passed		0
testBadPosition	Passed		0
testPopFromEmpty	Passed		0.001
testPushToEmpty	Passed		0
testPopEmptying	Passed		0
testPushCentral	Passed		0
testPopCentral	Passed		0
testPushFilling	Passed		0
testPopFromFull	Passed		0
testPushToFull	Passed		0
testThreadedAccess	Passed		0.039
testUnsafeThreadedAccess	Failed	Failed on iteration 116	0.117

5.4 Differences between ArrayStack and FixedArrayStack

Listing 4: AS01, AS02. Capacity bug fix; parameter range check

```
20,21c20,24
<   public ArrayStack(int cap) {
<     capacity = CAPACITY;
---
>   public FixedArrayStack(int cap) {
>     if (cap < 1)
>       throw new IllegalArgumentException(
>           "A stack must be at least one element big.");
>     capacity = cap;
```

Listing 5: AS03, AS04. Synchronize push; Capacity bug fix.

```
25,26c28,29
<   public void push(Object e) throws StackFullException {
<     if (size() == CAPACITY)
---
>   public synchronized void push(Object e) throws StackFullException {
>     if (size() == capacity)
```

Listing 6: AS05. Synchronize pop

```
32c35
<   public Object pop() throws StackEmptyException {
---
>   public synchronized Object pop() throws StackEmptyException {
```

Listing 7: AS06, AS07. Argument checking; tos bug fix

```
56,57c59,64
<     if (i > tos)
<       throw new StackOutOfScopeException(
>                 "Attempt to go beyond top of stack");
---
>     if (i < 1)
>       throw new IllegalArgumentException(
>           "Stack position must be greater than one.");
>     if (i > (tos + 1))
>       throw new StackOutOfScopeException(
>                 "Attempt to go beyond top of stack ("
>           + i + ">" + (tos + 1) + ")");
```

Listing 8: AS08. Array index bug fix

```
64c71
<       for (int i = 1; i < size(); i++)
---
>       for (int i = 0; i < size(); i++)
```

Listing 9: AS09. Trim toString()

```
66c73
<       return Sout;
---
>       return Sout.trim();
```

6 Defects

ID	Method	Issue	Recommendation
AS01	ArrayStack(int cap)	cap not checked for less than zero.	Add parameter check.
AS02	ArrayStack(int cap)	capacity always set to CAPACITY.	Set capacity to cap.
AS03	push(Object e)	Not synchronized.	Add synchronized keyword.
AS04	push(Object e)	CAPACITY instead of capacity.	Change to capacity.
AS05	pop()	Not synchronized.	Add synchronized keyword.
AS06	atPosition(int i)	No parameter lower bounds checking.	Check i greater than zero.
AS07	atPosition(int i)	Position compared to index.	Add one to index to make comparable.
AS08	toString()	First element of array missed.	Start array index from zero.
AS09	toString()	Always ends in a space.	Should be trimmed.
AS10	atPosition(int i)	Method not in interface.	Add to interface or delete method.

7 Test Assessment

The tests have revealed some serious defects which must be remedied. Once remedied, as per FixedArrayStack, the class is fit for purpose.

7.1 Code Style

The stylistic flaws identified by Checkstyle should be addressed if the class is to be part of a larger suite.

7.2 Code Quality

The threaded tests revealed that the allocation of a very large stack takes an appreciable length of time. There might be performance advantages to allocating the stack space when it was needed, rather than allocating the maximum that might be needed, depending upon whether the maximum stack size is known.

It might be an improvement to create a `Position` type and have the bounds checking in one place rather than using `int` for position parameters.

7.3 Test Improvement

The test `testUnsafeThreadedAccess()` should arguably be moved to a separate file where it can be easily excluded from running on the Continuous Integration server, as it is a proof that misuse is harmful, rather than a failing test.

7.4 Documentation Improvement

The need for synchronization against the stack in multithreaded use should be documented in the class javadoc.

The fact that it is possible to create a stack larger than available memory probably does not need to be documented, as the same can be said for the creation of any array.