[18pts] 1. Near the top of page 4 is the first example of implementing a Stack class, in this case by extending the ArrayList class.

(5pts) 1a. This code implements a stack of Objects, which is very generic but not very safe. Explain why it is generic and also why it is not very safe.

- ➔ It is generic because it is using ArrayList to implement a stack Object through inheritance.
- ➔ It is not safe because of fragile base-class and base-class tight coupling issue. For example when trying to use a_stack.clear(), the base class doesn't know anything about the stack pointer, and this will make the Stack object an undefined state. The variable stack_pointer is a member of the Stack class so the ArrayList superclass have no idea about it.

(3pts) 1b. This article is fairly old, and Java has evolved since it was written. What would be a better way of implementing this now, instead of using Object as the stack item class?

- ➔ Composition using instance Node variables that refers to other objects is another alternative.

(10pts) 1c. For each of the five SOLID principles, decide if this class obeys it or not, and explain why you chose this answer

- ➔ Single Responsibility Principal: Does not obey,
  There should never be more than one reason for a class to change. The stack class does not relate to the single responsibility as there are responsibility for ArrayList and Stack.
- ➔ Open Closed Principle: Does not obey.
  Open Closed Principle is open for extension but closed for modification. The Stack class in page 4 requires modification to resolve fragile base class issues.
- ➔ Liskov Substitution Principle: Does not obey,
  Because Subtypes must be substitutable for their base types. Lists must contain methods meaningful to a List and Stack must contains methods meaningful to a Stack.
- ➔ Interface-Segregation Principle: Does not obey,
  Mostly applies to Interfaces instead of classes. No interface implemented in this stack class. Only inheritance is introduced, and the derived class is tightly coupling to the base class and forces to use ArrayList methods that are not required in Stack implementation.

➔ Dependency Inversion Principle: Does not obey,
The Stack class contradicts dependency Inversion Principle core concept. High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions. Stack class is dependent on ArrayList class which resulted a tight coupling and if we try to make a change in one class it can break the other class which is risky.

[14pts] 2. Near the top of page 5 is a second implementation of the Stack class.

(4pts) 2a. There is a bug in this code. When during development or testing would it likely be found? If your answer needs it, provide a test case that uncovers the bug.

Bug 1:
```
Stack s = new Stack();
        s.push("a");
        s.push("b");
      int num =(int)s.pop();
```
Exception occurs due to use of raw arraylist.

Bug 2:
```
Object articles [] = {1,2,3};

        s.push_many(articles);
```

Unresolved compilation problem:
o cannot be resolved to a variable

Bug 3: Bad performance: push_many() works by calling push() method

(10pts) 2b. Assuming the bug is fixed, for each of the five SOLID principles, decide if this class obeys it or not, and explain why you chose this answer.

➔ Single Responsibility Principal: Yes it obey:
Because everything in the class are related to that single purpose in the stack class. This only applies after the bug is fixed.

➔ Open Closed Principle: Yes, it obey:
The Stack class is now open for extension at list from Monitorable_stack class. However, the class still needs modification to fix above mentioned bugs.
If we assume that above mentioned bugs are fixed then it obeys this principle.

➔ Liskov Substitution Principle Yes, it obey:
Through encapsulation, lists contains methods meaningful to a List and Stack contains methods meaningful to a Stack.

➔ Interface-Segregation:, No, it does not obey
No Interface implemented in this stack class. Only inheritance is avoided through encapsulating the data structure.

➔ Dependency Inversion Principle: Yes, it obey:
The Stack classes is loosely coupled through encapsulation.

[20pts] 3. In the second half of page 5 is a class Monitorable_stack that extends Stack.

(4pts) 3a. What is a minimal test that achieves 100% statement coverage?

```
import static org.junit.Assert.assertEquals;
import org.junit.jupiter.api.Test;

/**
 * Statement coverage
 */

public class StackStatementCoverageTest {

	@Test
	public void TestMaximum_size_so_far() {

		Monitorable_stack s = new Monitorable_stack();
		assertEquals(0, s.maximum_size_so_far());
	}

	@Test
	public void TestPush() {

		Monitorable_stack s = new Monitorable_stack();
		s.push(1);
		s.push(2);
		assertEquals(2, s.maximum_size_so_far());
	}
	@Test
	public void TestPop() {

		Monitorable_stack s = new Monitorable_stack();
		s.push(1);
	    s.push(2);
	    assertEquals((Integer) 2, s.pop());
	    assertEquals((Integer) 1, s.pop());

	}


}
```

(3pts) 3b. What is a minimal test that achieves 100% branch coverage?

```
import static org.junit.Assert.assertEquals;
import java.util.NoSuchElementException;
import org.junit.Assert;
import org.junit.jupiter.api.Test;


/**
 * Branch coverage
 */
public class StackBranchCoverageTest {

	@Test
	public void TestEmptyMaximum_size_so_far() {

		Monitorable_stack s = new Monitorable_stack();
		assertEquals(0, s.maximum_size_so_far());
	}

	@Test
	public void TestPushMaximum_size_so_far() {

		Monitorable_stack s = new Monitorable_stack();
		s.push(1);
		assertEquals(1, s.maximum_size_so_far());
	}

	@Test
	public void TestPopMaximum_size_so_far() {

		Monitorable_stack s = new Monitorable_stack();
		s.push(1);
		s.pop();
		assertEquals(0, s.maximum_size_so_far());
	}

	@Test
	public void TestPush() {

		Monitorable_stack s = new Monitorable_stack();
		s.push(1);
		s.push(2);
		assertEquals(2, s.maximum_size_so_far());
	}
	@Test
	public void TestPop() {

		Monitorable_stack s = new Monitorable_stack();
		s.push(1);
	    s.push(2);
	    assertEquals((Integer) 2, s.pop());
	    assertEquals((Integer) 1, s.pop());

	}
```

```
	@Test
	public void TestPopEmpty() {

		Monitorable_stack s = new Monitorable_stack();
		try {
			s.pop();
			Assert.fail();
			} catch (NoSuchElementException e) { System.out.print(e);
			}

	}
}
```

(3pts) 3c. Do either or both tests uncover any errors?

Yes, branch coverage discover that “high_water_mark” is not updating correctly when popping out. Infact, it is not being tracked when we are popping out.

(10pts) 3d. For each of the five SOLID principles, decide if this class obeys it or not, and explain why you chose this answer.

- ➔ Single Responsibility Principal: Yes it obey:
  Because everything in the class are related to a single purpose.

- ➔ Open Closed Principle: Yes it obey :
  Open for extension and changes shouldn’t ripple out to dependent classes/objects/modules

- ➔ Liskov Substitution Principle: Yes it obey:
  through inheritance, Lists contains methods meaningful to a List and Stack contains methods meaningful to a Stack.

- ➔ Interface-Segregation: No it does not obey:
  No interface implemented in this stack class to segregate responsibility. Only inheritance is introduced.

- ➔ Dependency Inversion Principle: No it does not obey:
  Monitorable_stack is defendant on Stack class while Both should depend upon abstractions.

[13pts] 4. At the top of page 6 is a third implementation of Stack. Note that it has "assert" statements which essentially embody "black box" specifications, or at least boundaries, for the methods.

(3pts) 4a. What new boundary or boundaries does this version of Stack have that the previous ones did not? How would you test it?

It checks stack size during popping and pushing using assert.

```
Stack s = new Stack();

              if (s.stack_pointer < s.stack.length)
              s.push("a");
              s.push("b");

              if(s.stack_pointer >=0)
                s.pop();

              Object articles [] = {1,2,3};
              if(s.stack_pointer + articles.length < s.stack.length)

              s.push_many(articles);
```

(4pts) 4b. Page 6 explains why the third implementation of Stack breaks the behavior of the Monitorable_stack subclass. What is the underlying issue in why it breaks (not just the new mechanics)?

We no longer call push() method in push_many and the MonitorableStack did not override pushMany(). Therefore, a call to pushMany() method will not update highwater-mark.

(6pts) 4c. Is it possible to design a Monitorable_stack subclass that works properly for both versions of Stack (p5 and p)? How, or why not?

Yes, it can be fixed by introducing interface and encapsulation.

Introduce an interface and split the classes. In the base class implement the interface in Simple_stack class then Implement stack interface in Monitorable_stack class.

[25pts] 5. The numbered code listing on p6-7 (mostly 7) contains a solution.

(3pts) 5a. What is the fundamental change to inheritance usage that the solution contains?

➔ An Interface inheritance is introduced and we delegated Simple_stack to use pushmany() method.

(4pts) 5b. Give one test case that would achieve 100% branch coverage.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class Simple_stackTest {


	@Test
	public void TestPush() {

		Simple_stack s = new Simple_stack();
		s.push(1);
		s.push(2);
		assertEquals(2, s.stack_pointer);
	}
	@Test
	public void TestPop() {

		Simple_stack s = new Simple_stack();
		s.push(1);
	    s.push(2);
	    assertEquals((Integer) 2, s.pop());
	    assertEquals((Integer) 1, s.pop());

	}

	@Test
	public void TestPopEmpty() {

		Simple_stack s = new Simple_stack();

			s.pop();

	}

	@Test
	public void TestPush_many() {

		Object source [] =  {1,2,3};

		Simple_stack s = new Simple_stack();

			s.push_many(source);

		assertEquals(3, s.stack_pointer);

	}

}
```

(3pts) 5c. What are the definitions of the variable "high_water_mark" (use line numbers)?

Defs of high_water_mark : 37,43,55

(3pts) 5d. What are the uses of the variable "high_water_mark"?

Uses of high_water_mark: 42,54,61

(4pts) 5e. What are the FEASIBLE def-use pairs for the variable "high_water_mark"?

(37,42), (37,54), (37,61), (43,42), (43,54), (43,61), (55,42), (55,54), (55,61)

(8pts) 5f. Give one test case that achieves 100% feasible def-use coverage for the variable "high_water_mark". (since a test case is a sequence, a properly constructed sequence can cover all of them)

```
public class high_water_mark_Test {

	// Covers Def use pair 37,42 and 43,42 and 43,61
	@Test
	public void TestPush() {

		Monitorable_Stack s = new Monitorable_Stack();
		s.push(1);
		s.push(2);
		assertEquals(2, s.maximum_size());
		Assert.assertTrue(s.current_size  > s.high_water_mark );
	}

	// covers Def use pair 37,54  and 55,54 and 55,61
	@Test
	public void TestPush_many() {

		Object source [] =  {1,2,3};

		Monitorable_Stack s = new Monitorable_Stack();

			s.push_many(source);

		assertEquals(3, s.maximum_size());
	}

	// covers Def use pair 37,61
	@Test
	public void TestMaximum_Size() {

		Monitorable_Stack s = new Monitorable_Stack();

		assertEquals(0, s.maximum_size());
	}

}
```

[10pts] 6. The author's last paper, "Why Getter and Setter Methods Are Evil" addresses another common issue in object oriented programming. Write a short paragraph on whether the existence of getters and setters in a class would make testing the class easier or harder. Address this both from the perspective of black box testing, and white box testing.

The existence of getters and setters most of the time violets the core pillars of OOP such as encapsulation and abstraction. Setter and Getters provide access to implementation details. Having a setter/getter for every member means the code using that class can have just as many dependencies on the internal state as if the members were all public. A simple change to accessed field type will require to apply the change in several places across the code. This will create all sorts of unnecessary complexity that the code has to handle. Same complexity will be reflected when we came Testing part such as “white box testing” and “black box testing”. It will make white box testing harder as the tester has to handle the above-mentioned unnecessary complexity while performing white box testing. Black box testing is performed without internal implementation details. As the result, the existence of the getters and setters would make no effect from the perspective of black-box testing.