## Question 1. [5 MARKS]

### Part (a) [1 MARK]

In assignment 2, one of the operations you implemented was called join. It's job was to produce the Cartesian product of two tables, but only including rows where specified columns match. Suppose we have a table `t` with $r1$ rows and $c1$ columns, and a table `u` with $r2$ rows and $c2$ columns. Consider the table produced by a join expression involving these two tables.

What is the minimum number of rows that the resulting table would have? | 0 |

What is the maximum number of rows that the resulting table would have? | $r1 \times r2$ |

### Part (b) [1 MARK]

What is the name of the design pattern that could be paraphrased as follows: "Please notify me whenever you change yourself."

**Solution:** Observer

### Part (c) [1 MARK]

Complete the line of code below to make it print out whether or not a string called `input` consists only of 0's and/or 1's, and is not empty.

```
System.out.println(       Pattern.matches("[01]+", input)       );
```

### Part (d) [1 MARK]

Suppose class `Sneetch` implements `Observer`. Which statement is true? Check one.

- ☐ `Sneetch` inherits a method called `update` that it must not override
- ☐ `Sneetch` inherits a method called `update` that it has the option of overriding
- ☑ `Sneetch` must implement a method called `update`
- ☐ Trick question! This is impossible because _____

### Part (e) [1 MARK]

Suppose class `Smurf` implements `Observable`. Which statement is true? Check one.

- ☐ `Smurf` must implement a method called `NotifyObservers`
- ☐ `Smurf` must not `extend` anything
- ☐ `Smurf` must not `implement` anything else
- ☑ Trick question! This is impossible because `Observable` is a class and you can only implement interfaces

## Question 2. [6 MARKS]

Consider the following code:

```
public class ExceptionQuestion {
    // Rest of class Omitted.
    public int helper(int n) throws InvalidArgumentException {
        // Body omitted.
    }
    public boolean doSomething(int n) {
        return (helper(n) > 0);
    }
}
```

### Part (a) [1 MARK]

This code can compile in its current form: is that true or false?

☐ True ☑ False

### Part (b) [4 MARKS]

Change method `doSomething` on the copy below so that it returns false if method `helper` reports that it was given an invalid argument.

**Solution:**

```
public boolean doSomething(int n) {
    int answer;
    try {
        answer = helper(n);
    } catch (InvalidArgumentException e) {
        return false;
    }
    return (answer >= 0);
}
```

Suppose that instead of dealing with any possible `InvalidArgumentException` that `helper` may throw, we wanted method `doSomething` to just pass the exception along. Modify the code on the copy below so that it will do that.

```
public boolean doSomething(int n) {

    return (helper(n) > 0);

}
```

**Solution:** We just need to declare that the method might throw an InvalidArgumentException:

```
public boolean doSomething(int n) throws InvalidArgumentException {
    return (helper(n) > 0);
}
```

### Part (c) [1 MARK]

Write class `InvalidArgumentException` below.

**Solution:**

```
public class InvalidArgumentException extends Exception {

}
```

## Question 3. [7 MARKS]

Consider this very simple stack class. Method bodies have been omitted.

```
public class Stack {

    /**
     * Construct an empty stack.
     */
    public Stack() {
    }

    /**
     * Make o the new top item on this stack.
     *
     * @param o the new top item.
     */
    public void push(Object o) {
    }

    /**
     * Remove and return the top item on this stack.
     *
     * @return the top item.
     */
    public Object pop() {
    }
}
```

Below is the JUnit code that Netbeans generates for it:

```
public class StackTest {

    public StackTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }
```

```
    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of push method, of class Stack.
     */
    @Test
    public void testPush() {
        // Body omitted
    }

    /**
     * Test of pop method, of class Stack.
     */
    @Test
    public void testPop() {

        System.out.println("pop");

        Stack instance = new Stack();

        Object expResult = null;

        Object result = instance.pop();

        assertEquals(expResult, result);

        // TODO review the generated test code and remove the default call to fail.

        fail("The test case is a prototype.");

    }
}
```

**Part (a)** [3 MARKS]

Modify the code for method `testPop` so that it tests the behaviour of pop on a stack with one element.

**Solution:**

```
/**
 * Test of pop method, of class Stack. Testing a stack of one element.
 */
@Test
public void testPopOneElement() {
    System.out.println("pop");
    Stack instance = new Stack();
    instance.push(1);
    Object expResult = 1;
    Object result = instance.pop();
    assertEquals(expResult, result);
}
```

**Part (b)** [4 MARKS]

Suppose you wanted to create the following fixture to be available and in the following state for every test case: a stack of three integers, 10, 20 and 30, with 10 on the bottom and 30 on the top. Modify the code above to make this happen.

**Solution:**

Two things are needed:

1. A new instance variable to hold the stack must be defined:

   ```
   private Stack fixture;
   ```

   This must *not* be simply a local variable in one of the methods.

2. The setUp method must ensure that the contents of that stack are as described.

   ```
   fixture = new Stack();
   fixture.push(10);
   fixture.push(20);
   fixture.push(30);
   ```

   Another approach would be to reuse the same stack object rather than construct a new one each time. For that to work, the stack would have to be cleared of all elements before pushing 10, 20 and 30. This could be done either in tearDown or at the beginning of setUp. However, this would require the Stack class to offer a method that reports when the stack is empty, or the JUnit code would have to directly acces the instance variables of Stack.