Lab 8.2

Objectives

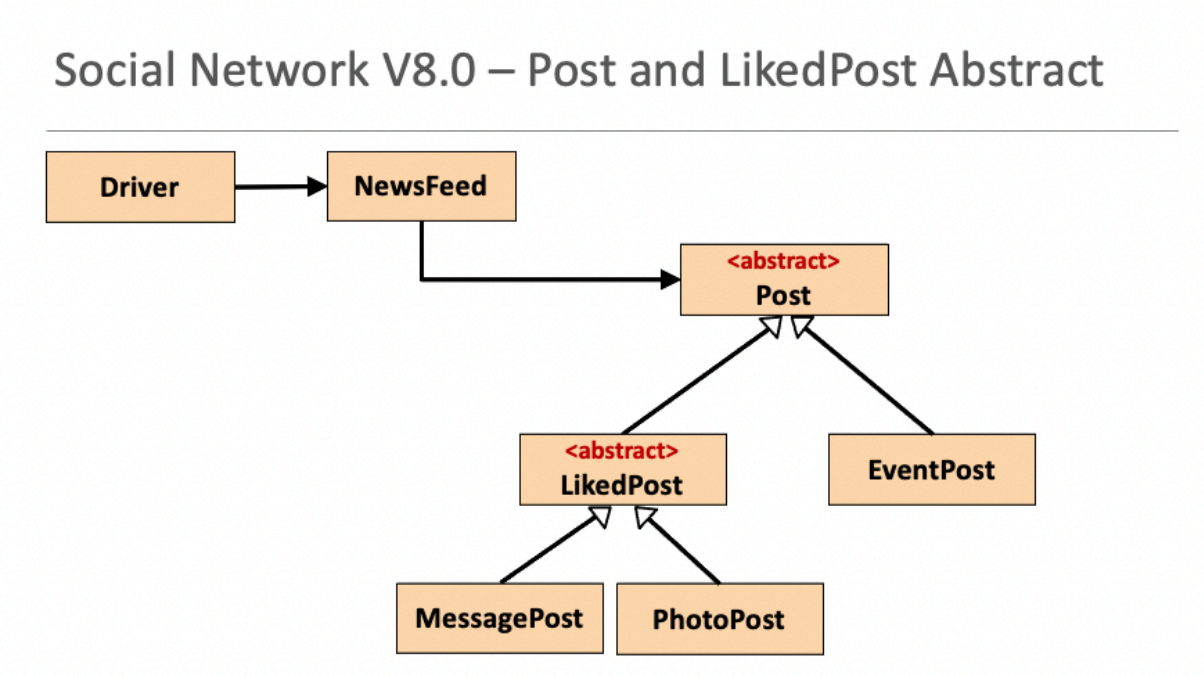
Interfaces · Social Network V9.0 · Interfaces and Collections\

Using Pre-defined Interfaces

In this practical, you will use the Collections Framework Interfaces in our Social Network App.

Creating Social Network V9.0 Project

Use either your V8.0 of the project or the downloadable version on Moodle.



Open Windows Explorer / Mac Finder and locate where V8.0 is. Copy the SocialNetworkV8.0 folder and paste it with the new name SocialNetworkV9.0.

Open this new project in IntelliJ. Once in IntelliJ, you will notice that the Project name is still SocialNetworkV8.0. Right click on it and select **Refactor…Rename**. Call the project SocialNetworkV9.0.

Now we are ready to refactor our app so that our *posts* ArrayList is defined using interfaces instead of concrete classes. This is how we have currently defined our ArrayList:

```
    private ArrayList<Post> posts;
```

```
    public NewsFeed() {
        posts = new ArrayList<>();
    }
```

NewsFeed - Adding Getters and Setters for post ArrayList

You might have noticed that we don't have getters and setters for our *posts* ArrayList.

Add them in (generate them in IntelliJ) now. We want to add them in so that we really see the effects of defining our ArrayList using interfaces instead of concrete classes.

Your generated code should look like this:

```
    public ArrayList<Post> getPosts() {
        return posts;
    }

    public void setPosts(ArrayList<Post> posts) {
        this.posts = posts;
  }
```

Using the List Interface

In NewsFeed:

- define the **posts** ArrayList to be a List.
- import java.util.List;
- change the **posts** getter return type to List
- change the **posts** setter parameter type to List
- change the load method to cast to List and not ArrayList

Your code changes should look like this now:

```
    private List<Post> posts;

    public NewsFeed() {
        posts = new ArrayList<>();
    }

    public List<Post> getPosts() {
        return posts;
    }

    public void setPosts(List<Post> posts) {
        this.posts = posts;
    }
    public void load() throws Exception {
```

```
        //list of classes that you wish to include in the
serialisation, separated by a comma
        Class<?>[] classes = new Class[]{EventPost.class,
MessagePost.class, PhotoPost.class, Post.class};

        //setting up the xstream object with default security and
the above classes
        XStream xstream = new XStream(new DomDriver());
        XStream.setupDefaultSecurity(xstream);
        xstream.allowTypes(classes);

        //doing the actual serialisation to an XML file
        ObjectInputStream in = xstream.createObjectInputStream(new
FileReader("posts.xml"));
        posts = (List<Post>) in.readObject();
        in.close();
    }
```

Run you app and test your code to make sure your refactoring did not break anything.

Changing to LinkedList

Now we will see the benefit of defining collections at Interface level.

We can now swap out our concrete implementation in one line of code. This means that we can change from using ArrayList to say, a LinkedList.

Try this now…your code change should look like this:

```
    public NewsFeed() {
        posts = new LinkedList<>();
    }
```

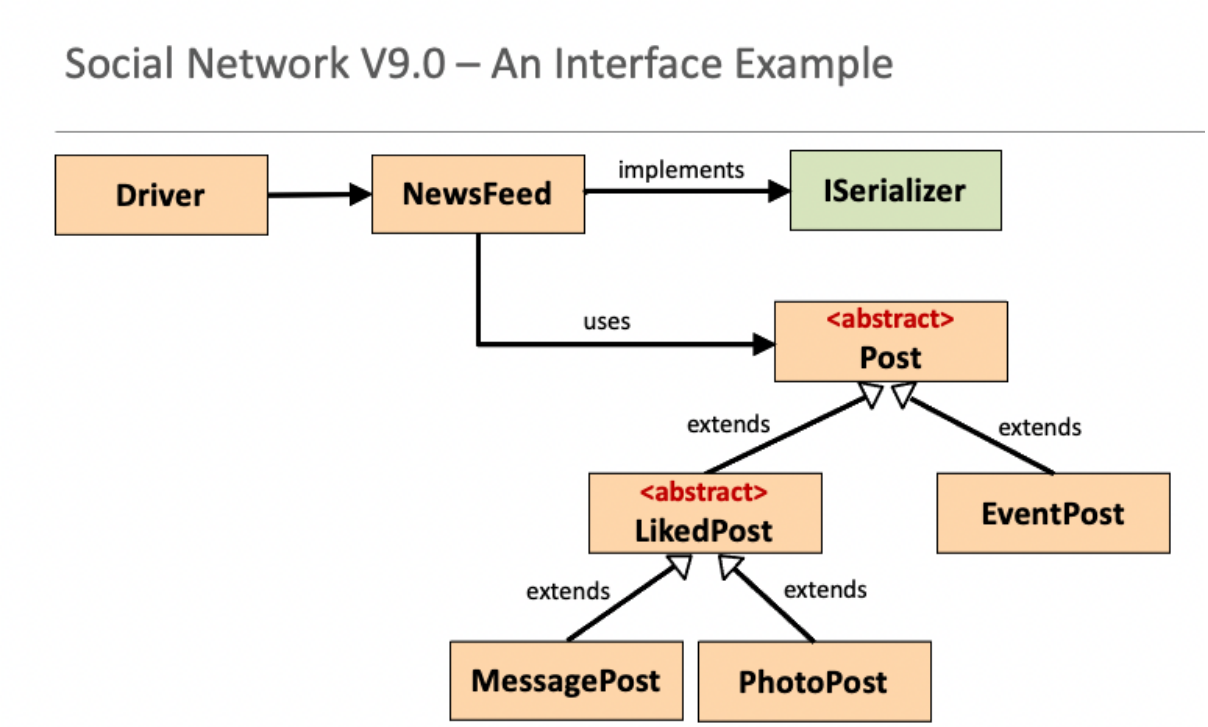
Add in the LinkedList import and remove the ArrayList one:

```
   import java.util.LinkedList;
```

Run your app again…it should work as expected, but instead of using an ArrayList, we are now using a LinkedList (with a one line code change!). If we weren't defining our collections at interface level (List), we would have many more code changes to make to move to LinkedList.

Change back to an ArrayList and move onto the next Step.

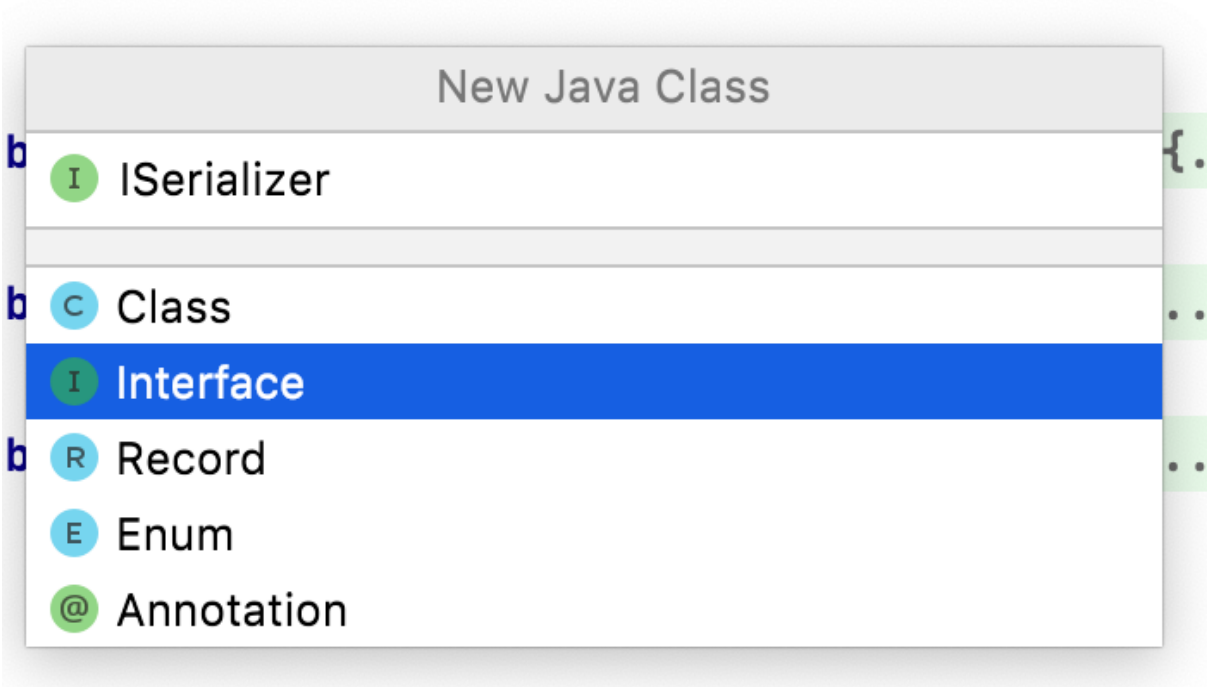
Implementing ISerializer



In this step, you will create the *ISerializer* interface and implement it in NewsFeed.

Creating ISerializer.java

Within the **src/utils** folder, create a new Interface called **ISerializer.java**. If you don't see the option to create an interface, create a new *Java Class* and when the dialog box appears, choose *Interface* as the *kind* from the drop-down box.



Add the following abstract methods into the Interface:

```
void save() throws Exception;
void load() throws Exception;
String fileName();
```

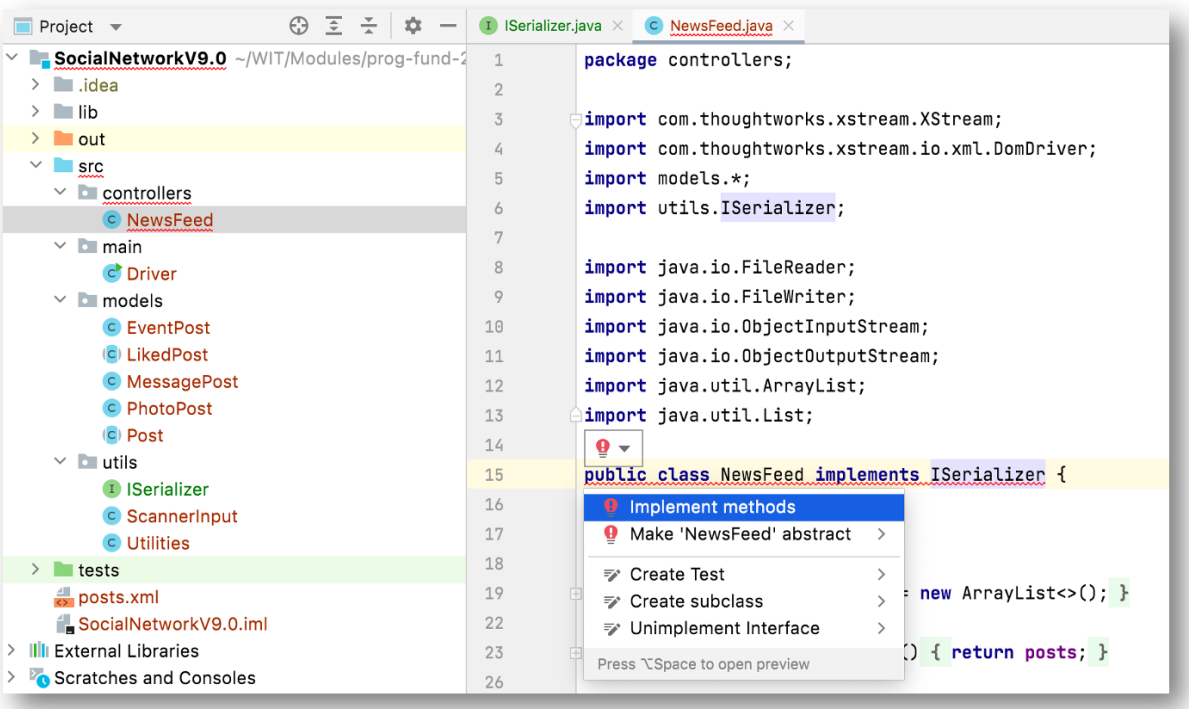
Any class implementing this new interface will have to provide concrete implementations fo reach of these abstract methods, if they haven't provided them already. This is how we can force a class to adhere to a design we want i.e. code to contract.

Updating NewsFeed.java

Implement the ISerializer interface in the NewsFeed class i.e.

```
public class NewsFeed implements ISerializer
```

You will notice immediately that IntelliJ starts complaining…we either need to make the Newsfeed class abstract, or provide a concrete implemention for each method listed in the interface:



Previously, our code contained these concrete methods:

- void save() throws Exception;
- void load() throws Exception;

So IntelliJ isn't complaining about these. However, there is no concrete implementation for this method:

- String fileName();

We will now provide an implementation for this method (note you can select the *implement method* option displayed when you hover over the red light bulb - this will create a method stub for you):

```
    @Override
    public String fileName() {

}
```

Add the code to return the fileName "posts.xml" i.e.:

```
    public String fileName(){
        return "posts.xml";
    }
```

Updating Driver.java

We can use this new method when saving and loading in the Driver; we can tell the user what the filename is.

To do this, update the Driver, loadPosts() method to:

```
    //load all the posts into the newsFeed from a file on the hard
disk
    private void loadPosts() {
        try {
            System.out.println("Loading from file: " +
newsFeed.fileName());
            newsFeed.load();
        } catch (Exception e) {
            System.err.println("Error reading from file: " + e);
        }
    }
```

And the savePosts() method to:

```
    //save all the posts in the newsFeed to a file on the hard disk
    private void savePosts() {
        try {
            System.out.println("Saving to file: " +
newsFeed.fileName());
            newsFeed.save();
        } catch (Exception e) {
            System.err.println("Error writing to file: " + e);
        }
  }
```

Run your app again, and try saving and loading…the filename should now be displayed.
Save your work.

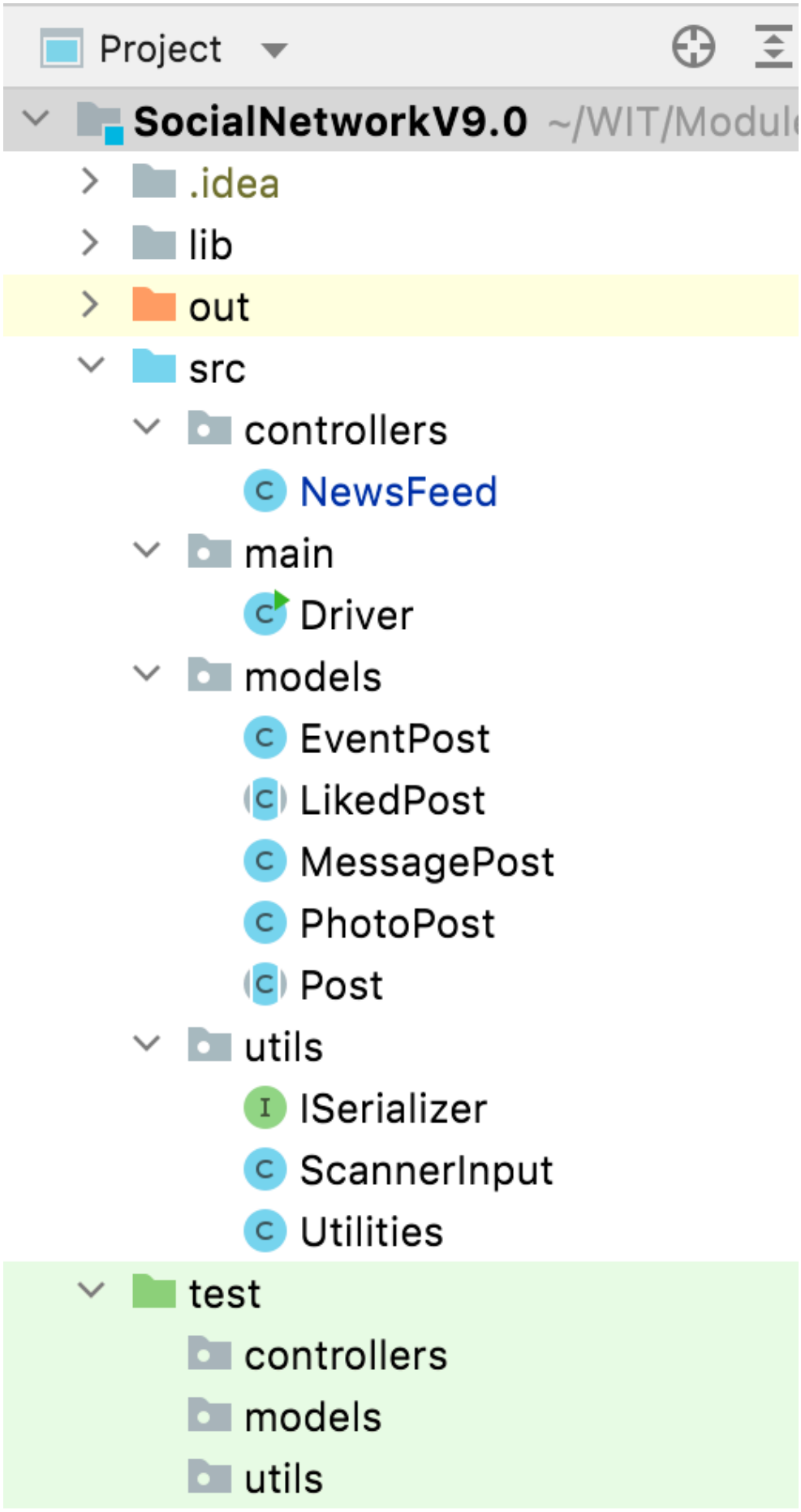
JUnit - MessagePost

We will now turn our attention to JUnit and testing Hierarchies with Abstraction.

If you still have a Test Harness class in your V9 project, you can just delete it. We will be using JUnit for this testing.

Test directory

If you haven't got a **test** folder in your project, create one now and mark the directory as **Test Sources Root**.
In your **test** folder, replicate your package structure that is in **src**.
Your folder structure should now look like this:

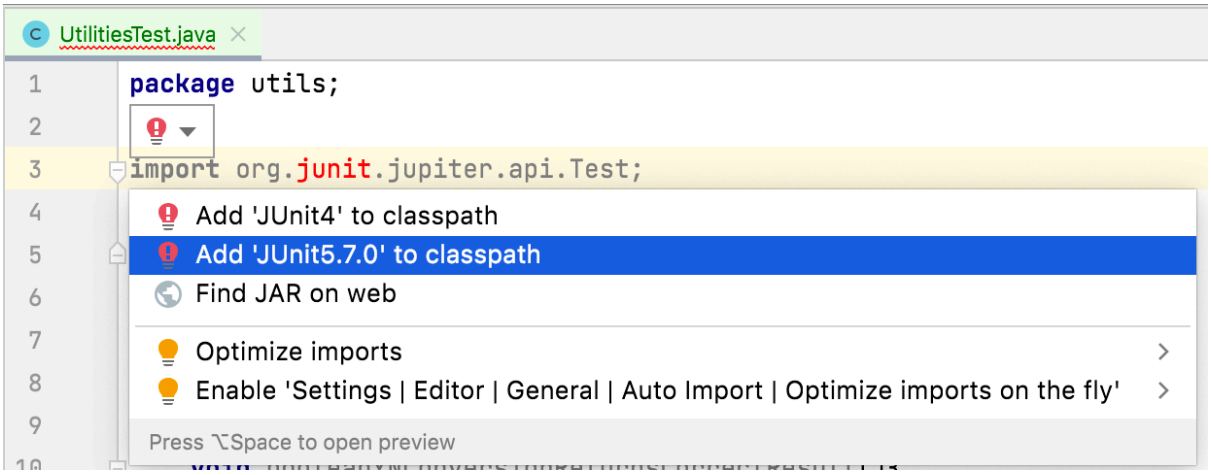

Project
SocialNetworkV9.0 ~/WIT/Modul
.idea
lib
out
src
controllers
NewsFeed
main
Driver
models
EventPost
LikedPost
MessagePost
PhotoPost
Post
utils
ISerializer
ScannerInput
Utilities
test
controllers
models
utils

UtilitiesTest

We previously wrote a UtilitiesTest class (for Shop). We can now reuse that test class here. In **test/utils**, create a new Java Class and copy this code into it:
package utils;

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class UtilitiesTest {

    @Test
    void validRangeWorksWithPositiveTestData(){
        assertTrue(Utilities.validRange(1, 1, 1));
        assertTrue(Utilities.validRange(1, 1, 2));
        assertTrue(Utilities.validRange(1, 0, 1));
        assertTrue(Utilities.validRange(1, 0, 2)) ;
        assertTrue(Utilities.validRange(-1, -2, -1)) ;
    }

    @Test
    void validRangeWorksWithNegativeTestData(){
        assertFalse(Utilities.validRange(1,0,0));
        assertFalse(Utilities.validRange(1,1,0));
        assertFalse(Utilities.validRange(1,2,1));
        assertFalse(Utilities.validRange(-1, -1, -2)) ;
    }

    @Test
    void truncateStringMethodTrucatesCorrectly(){
        assertEquals("123456789",
Utilities.truncateString("1234567890", 9));
        assertEquals("1234567890",
Utilities.truncateString("1234567890", 10));
        assertEquals("1234567890",
Utilities.truncateString("1234567890", 11));
        assertEquals("", Utilities.truncateString("1234567890", 0));
        assertEquals("", Utilities.truncateString("", 0));
        assertEquals("", Utilities.truncateString("", 10));
    }
}
```

Immediately, we will have syntax errors on our asserts, etc. To resolve this, we need to add JUnit5 to our build path. You can do this by clicking on the red light bulb and selecting the JUnit5 option:



Run these tests to make sure that you get all green ticks (i.e. no tests failed).

Testing the Hierarchy

In the **test/models** package, we will create test classes for the concrete classes in our hierarchy e.g.:

- EventPostTest
- MessagePostTest
- PhotoPostTest

It is through these classes that we will test the absract classes, LikedPost and Post. Remember, we cannot create objects of abstract classes and this is why we will do the testing via the concrete classes that comeplete them.

MessagePostTest – Getters

In the MessagePostTest class, add the following test fixture data:

```
    private MessagePost messagePostBelow, messagePostExact,
messagePostAbove, messagePostZero;

    @BeforeEach
    void setUp() {
        //author 9 chars, message 39 chars
        messagePostBelow = new MessagePost("Mairead M", "Programming
fundamentals assignment due");
        //author 10 chars, message 40 chars
        messagePostExact = new MessagePost("SiobhanDro", "Objects
and Classes Lecture starting now");
        //author 11 chars, message 41 chars
        messagePostAbove = new MessagePost("SiobhanRoch", "Computing
and Maths Centre open from 9am.");
        //author 0 chars, message 0 chars
        messagePostZero = new MessagePost("", "");
    }
```

```
    @AfterEach
    void tearDown() {
        messagePostBelow = messagePostExact = messagePostAbove =
messagePostZero = null;
    }
```

This data will test at the validation boundaries for the author and message. It will also test that the class can handle the “existance of nothing” i.e. empty Strings being passed for author and message.

Now add a nested class called Getters that will have three methods (only one is completed):

```
  @Nested
    class Getters {

        @Test
        void getMessage() {
            assertEquals("Programming fundamentals assignment due",
messagePostBelow.getMessage());
            assertEquals("Objects and Classes Lecture starting now",
messagePostExact.getMessage());
            assertEquals("Computing and Maths Centre open from 9am",
messagePostAbove.getMessage());
            assertEquals("", messagePostZero.getMessage());
        }

        @Test
        void getAuthor() {
        }

        @Test
        void getLikes() {
        }

    }
```

Note how we are testing the getMessage() method for each of our objects in the text fixture. Run this test method and it should pass.

Now complete the getAuthor() code, based on the approach used for getMessage().

Then complete the getLikes() code, making sure each of the four objects has a default likes value of 0.

Run these tests; they should all pass.

MessagePostTest - Setters

Continuing in the MessagePostTest class, add the folllowing nested class called Setters that will have three methods (again, only one is completed):

```
    @Nested
    class Setters {

        @Test
        void setMessage() {
            assertEquals("Programming fundamentals assignment due",
messagePostBelow.getMessage());

            messagePostBelow.setMessage("Programming fundamentals
results -- out");   //39 chars - update performed
            assertEquals("Programming fundamentals results -- out",
messagePostBelow.getMessage());

            messagePostBelow.setMessage("Programming fundamentals
results are out");   //40 chars - update performed
            assertEquals("Programming fundamentals results are out",
messagePostBelow.getMessage());

            messagePostBelow.setMessage("Programming fundamentals
module now over!");   //41 chars - update ignored
            assertEquals("Programming fundamentals results are out",
messagePostBelow.getMessage());
        }

        @Test
        void setAuthor() {
        }

        @Test
        void setLikes() {
        }

    }
```

Read the above test and make sure you understand what's happening. Run the test; it should be successful.

Then write similar test for both the setAuthor() and setLikes(). Run the tests and if they aren't passing, return to the code and troubleshoot the issue.

Did you notice, when testing setLikes(), that you could set your likes to anything you wanted, even preposterous values like -23143. We've uncovered a design issue in our code based. Really we should have a check in our setLikes() to make sure that

we only update the likes field if the value is 0 or more and maybe capped at 10,000. You can add that code in and update your tests accordingly.

MessagePostTest - Display

Now let's turn our attention to the two display methods. Add in the following to test the display() method:

```
    @Nested
    class DisplayMethods {

        @Test
        void testDisplay() {
            //testing the display when a post has no likes
            String toStringContents = messagePostExact.display();

assertTrue(toStringContents.contains(messagePostExact.getAuthor()));

assertTrue(toStringContents.contains(messagePostExact.getMessage()))
;
            assertTrue(toStringContents.contains("0 likes"));

            //testing the display when a post has likes
            messagePostExact.setLikes(1);
            assertTrue(messagePostExact.display().contains("1 people
like this"));
        }

        @Test
        void testDisplayCondensed(){

        }
    }
```

Run the testDisplay() method and your test should pass. Note how we are testing two paths in it…one when the MessagePost has zero likes and one when it has 1 like.

Now try write the testDisplayCondensed() method (i.e. for testing displayCondensed()) and run it.

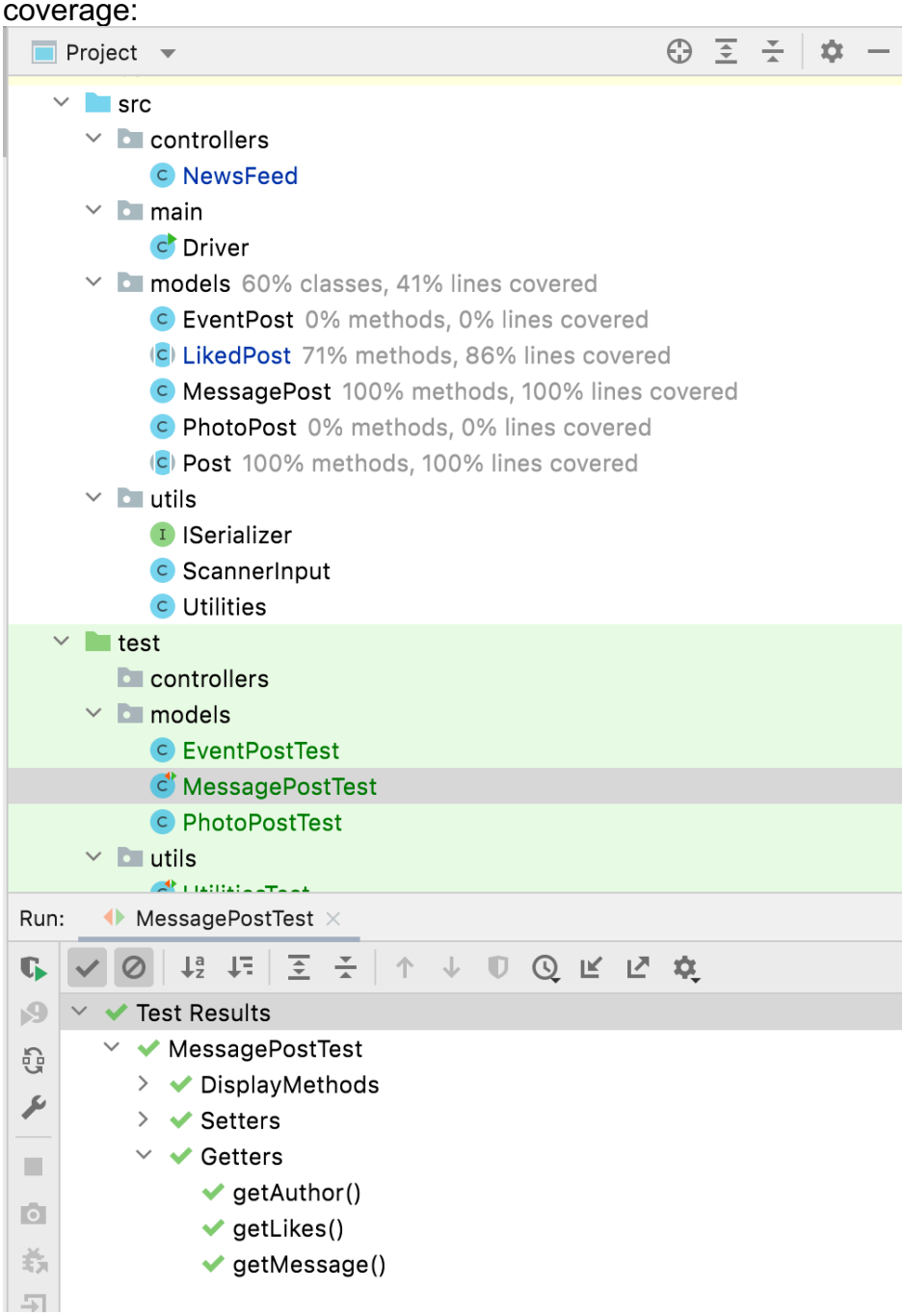
JUnit Coverage

Now that we have written a good few tests, we want to see how much of our code is tested by our JUnit tests. This is called code coverage.

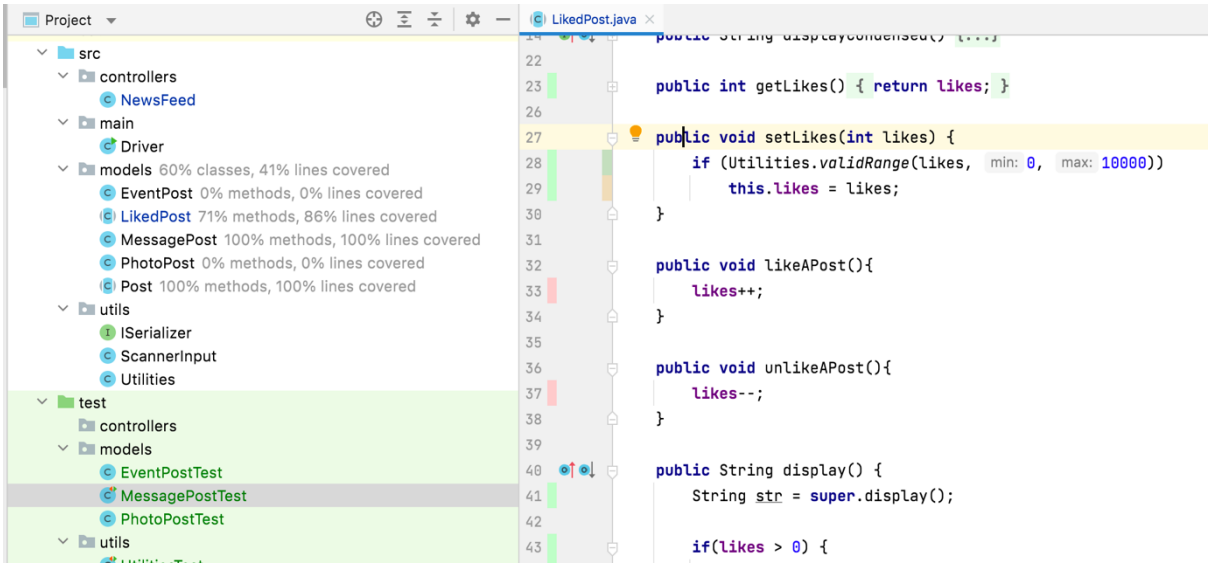
MessagePostTest - Coverage Report

Assuming your tests run successfully, run ALL the tests in the MessagePostTest class, this time with Coverage.

You should now see that the Post class is 100% tested, as is the MessagePost class. However, we have missed some methods in LikedPost as we only have 71% coverage:



When we look at the LikedPost code, we can see the methods we have missed (they are marked with a pink line):



MessagePostTest - Like and Unlike

Currently, these two methods haven't been tested:

```
public void likeAPost(){
     likes++;
 }

 public void unlikeAPost(){
     likes--;
 }
```

Let's create a new nested class category called LikesOnPosts with the following two method stubs in it:

```
@Nested
class LikesOnPosts {

    @Test
    void testingLikingOfPosts() {

    }

    @Test
    void testingUnLikingOfPosts() {

    }
}
```

In the method, testingLikingOfPosts() add code that will test this approach:

- verify that the likes for messagePostExact is 0
- call likeAPost() over messagePostExact
- verify that the likes for messagePostExact is now 1
- call likeAPost() over messagePostExact
- verify that the likes for messagePostExact is now 2

In the method, testingUnLikingOfPosts() add code that will test this approach:

- verify that the likes for messagePostExact is 0
- call unLikeAPost() over messagePostExact
- verify that the likes for messagePostExact is still 0
- call setLikes() over messagePostExact with a value of 2
- verify that the likes for messagePostExact is now 2
- call unLikeAPost() over messagePostExact
- verify that the likes for messagePostExact is now 1

Run your new tests.

You will notice that the first one is succsssful, however the second one fails. We have uncovered a bug…we can have minus likes. However, we should not let the likes drop below 0.

Return to your LikedPost class and make the change so that unLikeAPost() will not reduce below zero.

Now run your test again; this time it should pass.

Now run all the MessagePost tests with coverage to verify that you have 100% coverage in MessagePost, LikedPost and Post.

More models tests

Now that you have full coverage on your MessagePost hierarchy, you could try apply this knowledge to ***EITHER***:

- the EventPostTest and PhotoPostTest classes ***OR***
- the model classes in your current assignment.

Note: there is no soloution to the EventPostTest and PhotoPostTest classes as they are very similar in approach to MessagePost.

JUnit - NewsFeed

We will now turn our attention to JUnit and testing the API, NewsFeed.

NewsFeedTest

In the **test/controllers** package, we will create a test class for NewsFeed i.e.:

- NewsFeedTest

and add the following test fixture data to it:

```
    private MessagePost messagePostBelow, messagePostExact,
messagePostAbove, messagePostZero;
    private EventPost eventPostBelow, eventPostExact,
eventPostAbove, eventPostZero;
    private PhotoPost photoPostBelow, photoPostExact,
photoPostAbove, photoPostZero;
    private NewsFeed newsFeed = new NewsFeed();
    private NewsFeed emptyNewsFeed = new NewsFeed();

    @BeforeEach
    void setUp() {
        //author 9 chars, message 39 chars
        messagePostBelow = new MessagePost("Mairead M", "Programming
fundamentals assignment due");
        //author 10 chars, message 40 chars
        messagePostExact = new MessagePost("SiobhanDro", "Objects
and Classes Lecture starting now");
        //author 11 chars, message 41 chars
        messagePostAbove = new MessagePost("SiobhanRoch", "Computing
and Maths Centre open from 9am.");
        //author 0 chars, message 0 chars
        messagePostZero = new MessagePost("", "");

        //Event - author (max 10), eventname (max 35) event cost (0
- 99999)
        //author 9 chars, event 34 chars
        eventPostBelow = new EventPost("Mairead M", "Programming
Hackathon : Room IT101", -1);
        //author 10 chars, event 35 chars
        eventPostExact = new EventPost("SiobhanDro", "Programming
Fundamentals Quiz Night", 99999);
        //author 11 chars, event 36 chars
        eventPostAbove = new EventPost("SiobhanRoch", "Programming
Fundamentals Study Group", 100000);
        //author 0 chars, event 0 chars, cost 0
        eventPostZero = new EventPost("", "", 0);

        //Photo - author (max 10), caption (max 100) filename (40)
        //author 9 chars, caption 99 chars, filename 39
        photoPostBelow = new PhotoPost("Mairead M",
                "Programming Hackathon : Room IT101 : group photo
with all participants from BSc (Hons) Applied Yr 1",
                "Hackathon IT101-BSc (Hons) Applied Yr 1");
        //author 10 chars, caption 100 chars, filename 40
        photoPostExact = new PhotoPost("SiobhanDro",
```

```
                "Programming Fundamentals Quiz Night 2021- Podium
photo of the winning team BSc (Hons) Applied Year 1",
                "Prog Fund Quiz 2021-Applied Winning Team");
        //author 11 chars, caption 101 chars, filename 41
        photoPostAbove = new PhotoPost("SiobhanRoch",
                "Programming Fundamentals Study Group - multiple
groups hard at work on the morning of day 1, Sep 2021",
                "Programming Fundamentals Study Group 2021");
        //author 0 chars, caption 0 chars, filename 0 chars
        photoPostZero = new PhotoPost("", "", "");

        //9 - add all objects above except messagePostAbove,
photoPostBelow, eventPostExact
        newsFeed.addPost(messagePostBelow);
        newsFeed.addPost(messagePostExact);
        newsFeed.addPost(eventPostBelow);
        newsFeed.addPost(photoPostExact);
        newsFeed.addPost(messagePostZero);
        newsFeed.addPost(photoPostAbove);
        newsFeed.addPost(eventPostAbove);
        newsFeed.addPost(eventPostZero);
        newsFeed.addPost(photoPostZero);
    }

    @AfterEach
    void tearDown(){
        messagePostBelow = messagePostExact = messagePostAbove =
messagePostZero = null;
        eventPostBelow = eventPostExact = eventPostAbove =
eventPostZero = null;
        photoPostBelow = photoPostExact = photoPostAbove =
photoPostZero = null;
        newsFeed = emptyNewsFeed = null;
    }
```

Also include the following Nested classes:

- GettersSetters
- ArrayListCRUD
- CountingMethods
- LikingMethods
- ListingMethods
- FindingMethods
- HelperMethods

In the following sections, we will write the tests methods for these Nested classes.

NewsFeedTest - GettersSetters

Read and understand the approach in these test methods. Then add the code to the GetterSetters nested class:

```
        @Test
        void gettingPostListReturnsList() {
            List<Post> testPosts = new ArrayList<>();
            testPosts.add(messagePostBelow);
            testPosts.add(messagePostExact);
            testPosts.add(eventPostBelow);
            testPosts.add(photoPostExact);
            testPosts.add(messagePostZero);
            testPosts.add(photoPostAbove);
            testPosts.add(eventPostAbove);
            testPosts.add(eventPostZero);
            testPosts.add(photoPostZero);
            assertEquals(testPosts, newsFeed.getPosts());

            assertEquals(new
ArrayList<Post>(),emptyNewsFeed.getPosts());
        }

        @Test
        void settingPostListReplacesList() {
            List<Post> testPosts = new ArrayList<>();
            testPosts.add(messagePostBelow);
            testPosts.add(messagePostExact);
            testPosts.add(eventPostBelow);

            assertEquals(9, newsFeed.numberOfPosts());
            newsFeed.setPosts(testPosts);
            assertEquals(testPosts, newsFeed.getPosts());
            assertEquals(3, newsFeed.numberOfPosts());

            emptyNewsFeed.setPosts(new ArrayList<Post>());
            assertEquals(0,emptyNewsFeed.numberOfPosts());
        }
```

Run these tests. Assuming they run successfully, move onto the next Nested class.

NewsFeedTest - ArrayListCRUD

Read and understand the approach in these test methods. Then add the code to the ArrayListCRUD nested class:

```
    @Test
        void addingAPostAddsToArrayList() {
            assertEquals(9, newsFeed.numberOfPosts());
```

```
            //testing MessagePost object
            assertTrue(newsFeed.addPost(messagePostAbove));
            assertEquals(messagePostAbove,
newsFeed.findPost(newsFeed.numberOfPosts() - 1));

            //testing PhotoPost object
            assertTrue(newsFeed.addPost(photoPostBelow));
            assertEquals(photoPostBelow,
newsFeed.findPost(newsFeed.numberOfPosts() - 1));

            //testing EventPost object
            assertTrue(newsFeed.addPost(eventPostExact));
            assertEquals(eventPostExact,
newsFeed.findPost(newsFeed.numberOfPosts() - 1));

            //testing empty arraylist
            assertEquals(0, emptyNewsFeed.numberOfPosts());
            assertTrue(emptyNewsFeed.addPost(eventPostExact));
            assertEquals(eventPostExact,
emptyNewsFeed.findPost(emptyNewsFeed.numberOfPosts() - 1));
        }

        @Test
        void updatingAPostThatDoesNotExistReturnsFalse() {
            //testing arraylist with items
            assertFalse(newsFeed.updateMessagePost(9, "Updating
Message", "Work"));
            assertFalse(newsFeed.updateEventPost(-1, "Updating
Event", "No update", 1));
            assertFalse(newsFeed.updatePhotoPost(10, "Updating
Photo", "No update", "No file"));

            //testing empty arraylist
            assertFalse(emptyNewsFeed.updateMessagePost(0, "Updating
Note", "Work"));
        }

        @Test
        void updatingAMessagePostThatExistsReturnsTrueAndUpdates() {
            //check messagePost, index 1 exists and verify object
            MessagePost foundPost = (MessagePost)
newsFeed.findPost(1);
            assertEquals(messagePostExact, foundPost);

            //update messagePost, index 1 exists and check the
contents
            assertTrue(newsFeed.updateMessagePost(1, "NewAuthor",
"Updated Message"));
```

```
            MessagePost updatedPost = (MessagePost)
newsFeed.findPost(1);
            assertEquals("NewAuthor", updatedPost.getAuthor());
            assertEquals("Updated Message",
updatedPost.getMessage());
        }


        @Test
        void deletingAPostThatDoesNotExistReturnsNull(){
            assertNull(emptyNewsFeed.deletePost(0));
            assertNull(newsFeed.deletePost(-1));

assertNull(newsFeed.deletePost(newsFeed.numberOfPosts()));
        }

        @Test
        void
deletingAPostThatExistsDeletesAndReturnsDeletedObject(){
            //deleting a post at the start of the arraylist
            assertEquals(9, newsFeed.numberOfPosts());
            assertEquals(messagePostBelow, newsFeed.deletePost(0));
            assertEquals(8, newsFeed.numberOfPosts());

            //deleting a post at the end of the arraylist
            assertEquals(8, newsFeed.numberOfPosts());
            assertEquals(photoPostZero, newsFeed.deletePost(7));
            assertEquals(7, newsFeed.numberOfPosts());
```

Run these tests. Assuming they run successfully, try write the test methods for:

- void updatingAnEventPostThatExistsReturnsTrueAndUpdates()
- void updatingAPhotoPostThatExistsReturnsTrueAndUpdates()

You can base the approach on updatingAMessagePostThatExistsReturnsTrueAndUpdates().

Run these tests. Assuming they run successfully, move onto the next Nested class.

NewsFeedTest – CountingMethods

Read and understand the approach in this test methods. Then add the code to the CountingMethods nested class:

```
        @Test
        void numberOfPostsCalculatedCorrectly() {
            assertEquals(9, newsFeed.numberOfPosts());
            assertEquals(0, emptyNewsFeed.numberOfPosts());
```

```
    }
```

Run these tests. Assuming they run successfully, try write the test methods for:

- void numberOfEventPostsCalculatedCorrectly()
- void numberOfMessagePostsCalculatedCorrectly()
- numberOfPhotoostsCalculatedCorrectly()

Run these tests. Assuming they run successfully, move onto the next Nested class.

NewsFeedTest – LikingMethods

Read and understand the approach in these test methods. Then add the code to the LikingMethods nested class:

```
    @Test
    void likingAnExistingPostIncreasesTheLikesByOne() {
        //check post, index 3 exists and verify the likes
        PhotoPost foundPost = (PhotoPost) newsFeed.findPost(3);
        assertEquals(0, foundPost.getLikes());
        newsFeed.likeAPost(3);
        assertEquals(1, foundPost.getLikes());

        //liking a post that doesn't exist doesn't cause an
error
        assertEquals(null,
newsFeed.findPost(newsFeed.numberOfPosts()));
        newsFeed.likeAPost(newsFeed.numberOfPosts());
    }

    @Test
    void unLikingAnExistingPostDecreasesTheLikesByOne() {
        //check post, index 3 exists and update likes to 2.
        PhotoPost foundPost = (PhotoPost) newsFeed.findPost(3);
        assertEquals(0, foundPost.getLikes());
        newsFeed.likeAPost(3);
        newsFeed.likeAPost(3);
        assertEquals(2, foundPost.getLikes());

        //unlike 3 times and verify likes (should not drop below
zero)
        newsFeed.unLikeAPost(3);
        assertEquals(1, foundPost.getLikes());
        newsFeed.unLikeAPost(3);
        assertEquals(0, foundPost.getLikes());
        newsFeed.unLikeAPost(3);
        assertEquals(0, foundPost.getLikes());
```

```
            //unliking a post that doesn't exist doesn't cause an
error
            assertEquals(null,
newsFeed.findPost(newsFeed.numberOfPosts()));
            newsFeed.unLikeAPost(newsFeed.numberOfPosts());
        }
```

Run these tests. Assuming they run successfully, move onto the next Nested class.

NewsFeedTest - ListingMethods

Read and understand the approach in these test methods. Then add the code to the ListingMethods nested class:

```
    @Test
        void showReturnsPostsWhenArrayListHasPostsStored() {
            assertEquals(9, newsFeed.numberOfPosts());
            String posts = newsFeed.show();
            //checks for some objects in the string
            assertTrue(posts.contains("Programming fundamentals
assignment due"));
            assertTrue(posts.contains("Objects and Classes Lecture
starting now"));
            assertTrue(posts.contains("Programming Hackathon : Room
IT101"));
            assertTrue(posts.contains("Programming Fundamentals Quiz
Night"));
            assertTrue(posts.contains("Programming Fundamentals
Study Group"));
            assertTrue(posts.contains("Prog Fund Quiz 2021-Applied
Winning Team"));
        }

        @Test
        void showMessagesReturnsNoPostsStoredWhenArrayListIsEmpty()
{
            assertEquals(0, emptyNewsFeed.numberOfMessagePosts());

assertTrue(emptyNewsFeed.showMessagePosts().toLowerCase().contains("
no message posts"));
        }
```

Run these tests. Assuming they run successfully, try write the test methods for:

- void showMessagesReturnsNoPostsStoredWhenArrayListIsEmpty()
- void showMessagesReturnsPostsWhenArrayListHasPostStored()
- void showPhotosReturnsNoPostsStoredWhenArrayListIsEmpty()
- void showPhotosReturnsPostsWhenArrayListHasPostStored()
- void showEventReturnsNoPostsStoredWhenArrayListIsEmpty()

- void showEventReturnsPostsWhenArrayListHasPostStored()

Run these tests. Assuming they run successfully, move onto the next Nested class.

NewsFeedTest – FindingMethods

Read and understand the approach in these test methods. Then add the code to the FindingMethods nested class:

```
    @Test
        void findPostReturnsPostWhenIndexIsValid() {
            assertEquals(9, newsFeed.numberOfPosts());
            assertEquals(messagePostBelow, newsFeed.findPost(0));
            assertEquals(photoPostZero, newsFeed.findPost(8));
        }

        @Test
        void findPostReturnsNullWhenIndexIsInValid() {
            assertEquals(0, emptyNewsFeed.numberOfPosts());
            assertNull(emptyNewsFeed.findPost(0));

            assertEquals(9, newsFeed.numberOfPosts());
            assertNull(newsFeed.findPost(-1));
            assertNull(newsFeed.findPost(9));
        }
```

Run these tests. Assuming they run successfully, move onto the next Nested class.

NewsFeedTest – HelperMethods

Read and understand the approach in these test methods. Then add the code to the HelperMethods nested class:

```
    @Test
        void isValidIndexReturnsTrueForValidIndex() {
            assertTrue(newsFeed.isValidIndex(0));

assertTrue(newsFeed.isValidIndex(newsFeed.numberOfPosts() -1));
        }

        @Test
        void isValidIndexReturnsFalseForInValidIndex() {
            assertFalse(emptyNewsFeed.isValidIndex(0));
            assertFalse(emptyNewsFeed.isValidIndex(1));

            assertFalse(newsFeed.isValidIndex(-1));

assertFalse(newsFeed.isValidIndex(newsFeed.numberOfPosts()));
        }
```

Run these tests. Assuming they run successfully, try write the test methods for:

- void isValidMessageIndexReturnsTrueForValidIndex()
- void isValidPhotoIndexReturnsTrueForValidIndex()
- void isValidEventIndexReturnsTrueForValidIndex()
- void isValidMessageIndexReturnsFalseForInValidIndex()
- void isValidPhotoIndexReturnsFalseForInValidIndex()
- void isValidEventIndexReturnsFalseForInValidIndex()

Run these tests. Assuming they run successfully, move onto the next Nested class.

Test Coverage

Finally run a test coverage over the NewsFeedTest class.
You will notice that we haven't tested persistence (we won't cover it this semester as it involved many more steps).
Save your work.