# Unit Testing Strategies	start time:

This activity will explore strategies to use JUnit to test Java code.

Before you start, complete the form below to assign a role to each member.
If you have 3 people, combine Speaker & Reflector.

Team	**Date**
Team Roles	**Team Member**
Recorder: records all answers & questions, and provides copies to team & facilitator.	
Speaker: talks to facilitator and other teams.	
Manager: keeps track of time and makes sure everyone contributes appropriately.	
Reflector: considers how the team could work and learn more effectively.	

Notes

1. When a question asks you to **explain** something, always answer in **complete sentences**.
2. **Recorder**: Write legibly & neatly so that everyone can read & understand your responses.
    a. Note the time whenever your team starts a new section or question.
3. You will need a computer with BlueJ open and ready to use below.
    a. In BlueJ, go to Tools > Preferences > Editor to adjust the font size.
    b. Consider using an external monitor or projector so everyone can see BlueJ easily.

As examples, we will use classes `EmailAddress` and `EmailAddressTest`.
(To save space, javadoc comments have been collapsed to one line.)
Create both classes in BlueJ with the code below.
As you work through the activity, add each piece of sample code into its class,
so that at the end of the activity your classes contain all of the sample code.
You will then use this code in a homework assignment.

```
// add to class EmailAddress
import java.util.*;

/** class for email addresses. */
public class EmailAddress {

} // end class
```

```
// add to class EmailAddressTest
import static org.junit.Assert.*;
import        org.junit.*;

/** test class for EmailAddress */
public class EmailAddressTest {
  /** Sets up test fixture before every test case method. */
  @Before public void setUp() { }

  /** Tears down test fixture after every test case method. */
  @After  public void tearDown() { }

} // end class
```

(6 min) I. Tests for Self-Contained Methods	start time:

```
// add to class EmailAddress
  /** Gets user name from email address. */
  public static String getUserPart(String address) {
    return address.substring( 0, address.indexOf("@") );
  }
  /** Gets domain from email address. */
  public static String getDomainPart(String address) {
    return address.substring( 1 + address.indexOf("@") );
  }
```

```
// add to class EmailAddressTest
  @Test public void getUserPartNormal() {
    assertEquals( "root",
                  EmailAddress.getUserPart("root@example.com") );
  }
  @Test public void getDomainPartNormal() {
    assertEquals( "example.com",
                  EmailAddress.getDomainPart("root@example.com") );
  }
```

1. (2 min) Add this code to your code and make sure it compiles and the tests pass.
Explain why each method in `class EmailAddress` needs more than one test method in `EmailAddressTest` to guarantee that it works correctly.

2. (1 min) These methods use their input parameters, but do not refer to any variables or fields. Explain why such methods are easy to test.

3. (1 min) **Class methods** are defined with the keyword `static`, and called using a class name, while **instance methods** are called using a specific instance (object) of the class. Which type of method (C=class or I=instance) is:

a.	`Math.abs(-4)`		c.	`getUserPart()`	
b.	`"abc".toUpperCase()`		d.	`getDomainPart()`	

```
// add to class EmailAddressTest
  @Test public void getUserPartNoUser() {
    assertEquals( "INVALID",
                  EmailAddress.getUserPart("@example.com") );
  }
  @Test public void getUserPartNoDomain() {
    assertEquals( "INVALID",
                  EmailAddress.getUserPart("root@") );
  }
  @Test public void getUserPartNoAt() {
    assertEquals( "INVALID",
                  EmailAddress.getUserPart("root") );
  }
```

4. (2 min) Add this code to your code and make sure it compiles and these tests **fail**.
Explain why these tests are useful.

For homework (after class, not during):

a. Modify `getUser()` to pass these tests.
b. Define at leat 3 good unit tests in `EmailAddressTest` for `getDomain()`.
c. Modify `EmailAddress` as needed to pass all tests.

(If failing tests bother you, you may comment them out during activity,
but for the homework you should uncomment them and make sure all tests pass.)

(5 min) II. Tests for Input & Output (IO) Methods	start time:

```
// add to class EmailAddress
  /** Prints email address in standard format. */
  public static void printAddress(String user, String domain) {
    System.out.println( user + "@" + domain );
  }
  /** Prints mailto link in standard format. */
  public static void printMailto(String user, String domain) {
    System.out.println( "mailto:" + user + "@" + domain );
  }
  /** Asks user for email address. */
  public static String askForAddress() {
    System.out.print( "Please enter an email address: " );
    Scanner scanner = new Scanner(System.in);
    String  email   = scanner.nextLine();
    int     index   = email.indexOf("@");
    if (index < 1 || index > email.length() - 5) {
      return "INVALID";
    }
    return email;
  }
```

1. (2 min) Add this code to your code and make sure it compiles.
Explain why it might be hard to write unit tests for these methods.

```
// add to class EmailAddress
  /** Gets email address in standard format. */
  public static String formatAddress(String user, String domain) {
    return ( user + "@" + domain );
  }
  /** Prints email address in standard format. */
  public static void printAddress(String user, String domain) {
    System.out.println( formatAddress(user, domain) );
  }
```

2. (3 min) Add this code to your code (replace the original `printAddress` with 2 methods) and make sure it compiles. One method is easy to test, and the other doesn't need to be tested. Explain which method is which, and why.

For homework (after class, not during):

a. Define `getMailTo()` and use it in `printMailto()`.
b. Refactor `askForAddress()` in similar ways.
c. In `EmailAddressTest`, define at least 3 good unit tests each for `getAddress()` and `getMailTo()`.
d. Modify `EmailAddress` as needed to pass all tests.

(12 min) III. Tests for Get & Set Methods	start time:

```
// add to class EmailAddress
  // instance fields
  private String  domain  = "unknown",
                  user    = "unknown";

  /** Gets domain name. */
  public String getDomain() { return this.domain; }
  /** Gets user name. */
  public String getUser  () { return this.user  ; }
  /** Sets domain name. */
  public void setDomain(String domain) {
    this.domain = domain;
  }
  /** Sets user name. */
  public void setUser(String user) {
    if ( null != user ) {
      user = user.trim();
      if ( ! "".equals(user) ) {
        this.user = user;
      }
    }
  }
```

1. (3 min) Add this code to your code and make sure it compiles.
This is a common pattern - a class contains private fields for useful data values, and public methods to get and set field values.
Explain why `setUser()` will likely cause fewer problems than `setDomain()`.

2. (2 min) Explain why it might be hard to write unit tests for **set** methods.

3. (1 min) Why does the code use `==` to check for `null` and `.equals()` to check for “”?

```
// add to class EmailAddressTest
  @Test public void setUserNormal() {
    EmailAddress ea = new EmailAddress();
    assertEquals("unknown"  , ea.getUser() );
    ea.setUser  ("webmaster");
    assertEquals("webmaster", ea.getUser() );
  }
  @Test public void setUserNull() {
    EmailAddress ea = new EmailAddress();
    ea.setUser  (null);
    assertEquals("unknown"  , ea.getUser() );
  }
```

4. (3 min) Add this code to your code and make sure it compiles.
Explain why we should usually test get and set methods together.

5. (3 min) Add unit tests to show how `setUser()` handles strings that are:
a. **empty** ("") b. **blank** (" ") c. **padded** (" webmaster ").

(16 min) IV. Setup & Teardown	start time:

1. (1 min) Notice that many of the unit tests contain identical lines of code.
Explain why many test classes are likely to contain test methods with identical code.

2. (2 min) To reduce duplicate code, we could refactor the test class
to declare an instance field with a test object that is shared by the test methods.
Study `setUserNormal()` and `setUserNull()` above,
and describe problems that could occur if test methods share a test object.

```
// add to class EmailAddressTest
  private EmailAddress ea;

  /** Sets up test fixture before every test case method. */
  @Before public void setUp   () { this.ea = new EmailAddress(); }

  /** Tears down test fixture after every test case method. */
  @After  public void tearDown() { this.ea = null; }
```

3. (3 min) Add this code to your code to replace the original versions of
`setUp()` and `tearDown()`. Next, remove the duplicate code from individual tests.
Make sure your code compiles and runs.

4. (2 min) Explain why the instance field combined with `setUp()` and `tearDown()`
allows us to reduce duplicate code and simplify tests, without introducing added risks.

```
  /** Sets user name. */
  public String setUser(String user) {
    if ( null != user ) {
      user = user.trim();
      if ( ! "".equals(user) ) {
        this.user = user;
      }
    }
    return this.user;
  }
```

5. (2 min) **Do not** add this code to your code. How does this differ from setUser() above?

6. (1 min) Why might this approach make it easier to write unit tests?

7. (1 min) Why might this approach be more confusing for developers?

```
// add to class EmailAddress
  /** Prints email address in standard format. */
  public String getAddress() {
    return formatAddress(this.user, this.domain);
  }
  public void printAddress() {
    System.out.println( getAddress() );
  }
```

13. (3 min) Add this code to your code and make sure it compiles.
Note that there are now two methods named `printAddress()`.
Describe the differences between them and explain how Java decides which to use when.
(When two or more methods have the same name, we call it **overloading**.)

For homework (after class, not during):

a. Modify `setDomain()` to contain similar error checking.
b. Define at least 4 good unit tests in `EmailAddressTest` for `setDomain()` .
   Note that domains must be of the form:
   `example.com` `www.example.com` `www.cs.example.com`
c. Modify `EmailAddress` as needed to pass these tests.
d. Define good unit tests for `getAddress()`.
e. Modify `EmailAddress` as needed to pass these tests.
f. Define new instance methods `getMailto()` and `printMailto()`,
   using `getAddress()` and `printAddress()` as a model.
g. Define good unit tests for `getMailto()`.
h. Modify `EmailAddress` as needed to pass these tests.

(8 min) V. Tests for Constructors	start time:

```
// add to class EmailAddress
  /** Creates new email address without data. **/
  // default constructor only works if no constructor provided
  public EmailAddress() { }

  /** Creates new email address */
  public EmailAddress(String user, String domain) {
    setUser  (user);
    setDomain(domain);
  }
```

```
// add to class EmailAddressTest
  @Test public void newDefault() {
    EmailAddress ea = new EmailAddress();
    assertEquals( "UNKNOWN" , ea.getUser  () );
    assertEquals( "UNKNOWN" , ea.getDomain() );
  }
  @Test public void newNormal() {
    EmailAddress ea = new EmailAddress("web","example.org");
    assertEquals( "web"         , ea.getUser  () );
    assertEquals( "example.org" , ea.getDomain() );
  }
```

1. (3 min) Add this code to your code and make sure it compiles. Study the code and explain why the first test above expects the user and domain to be “UNKNOWN”.

2. (2 min) Recall that a **constructor** is a special method that constructs a new object. Unlike a regular method, a constructor does not have a return type and method name. Explain how constructors in Java are defined and called.

```
// DO NOT add to your code
  // if constructors were methods, they might be defined like this
  public static EmailAddress new(String user, String domain) {
    setUser(user); setDomain(domain);
  }
  // and called like this
    EmailAddress ea = EmailAddress.new("web", "example.org");
```

Note: This code shows what Java code might look like if constructors were static methods.

3. (2 min) Often (not always), a constructor calls a set method for each input parameter. Explain why we do not need unit tests for many different input arguments.