



## Lecture 8: Accessing the Web and Unit Tests

# Interacting with the Web

# How to Access Web Content

---

1. Give your app permission to access the web
2. Open a connection to a URL
3. Read data from the URL and store it somewhere
4. Display the data from the URL on your app

# How to Access Web Content

---

1. Give your app permission to access the web
2. Open a connection to a URL
3. Read data from the URL and store it somewhere
4. Display the data from the URL on your app

# Give your app permission to access the web

---

- Find AndroidManifest.xml file
- Navigate to the Permissions tab
- Select “Add → Uses Permission”
- Select `android.permission.INTERNET` from the drop-down menu

# How to Access Web Content

---

1. Give your app permission to access the web
2. Open a connection to a URL
3. Read data from the URL and store it somewhere
4. Display the data from the URL on your app

# Open a connection to a URL

---

- A URL is a type of URI
- Sample code:

```
URL myURL;  
myURL = new URL("http://myWebsite.com");  
URLConnection conn =  
    (URLConnection) url.openConnection();
```

# How to Access Web Content

---

1. Give your app permission to access the web
2. Open a connection to a URL
3. Read data from the URL and store it somewhere
4. Display the data from the URL on your app



# Read data from the URL and store it somewhere

---

- Get data from the URL

```
InputStream in = conn.getInputStream();
```

# Read data from the URL and store it somewhere

---

- Get data from the URL

```
InputStream in = conn.getInputStream();
```

- Use a reader to convert the data into the format you want. Useful Java classes.
  - `InputStreamReader`
  - `BufferedReader`
  - `StringBuffer`
  - `CharBuffer`

# Read data from the URL and store it somewhere

---

- Sample code  
(printing data out instead of storing)

```
BufferedReader in =  
    new BufferedReader(  
        new InputStreamReader(conn.getInputStream()));  
String inputLine;  
  
while ((inputLine = in.readLine()) != null) {  
    System.out.println(inputLine);  
}
```

# Read data from the URL and store it somewhere

---

- May encounter methods that throw exceptions, such as:
  - `MalformedURLException`  
(`new URL()` throws when the string isn't a URL)
  - `IOException`  
(`getInputStream()` throws on bad connection)
- Handle them gracefully
  - How should the app work without Internet?

# How to Access Web Content

---

1. Give your app permission to access the web
2. Open a connection to a URL
3. Read data from the URL and store it somewhere
4. Display the data from the URL on your app

# Display the data from the URL on your app

---

Access your stored data and display it using  
whatever combination of layouts and widgets  
that you choose!

# Unit Tests and JUnit

# What are unit tests?

---

- Small pieces of code that test your code
  - Test the smallest testable piece (unit)
  - Tests interact with your main code



# Why unit test?

---

- Guarantee your code does what you say
- Uncover corner cases early on
  - Ensure graceful degradation (GPS unavailable?)
- Debug before you release
- Can help guide development
  - Test-driven development (write tests first)

# What goes into a unit test?

---

- *Assertion* – A test of a single property or value (e.g. assert that “1+1” gives “2”)
- *Test Case* – A set of assertions that test a single function or use case
- *Test Suite* – A collection of related *Test Cases* to run together
- *Test Runner* – Code that runs the *Test Suites*
- *Mock Object* – An object substituting for another (when the object itself is not being tested)

# Some Types of Assertions

---

- All can take an extra first argument `String` message to print out when the assertion fails
- `org.junit.Assert`.
  - `assertEquals(expected, actual)`
    - Test that *expected* and *actual* are equal (`.equals()`)
  - `assertTrue(condition)/assertFalse(condition)`
    - Test that *condition* is `true/false`
  - `fail()`
    - Always fail

See also: <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

# More Types of Assertions

---

- `android.test.MoreAsserts`.
  - `assertMatchesRegex(expectedRegex, actual)`
    - Test that *actual* matches regular expression *expectedRegex*
  - `assertEmpty(iterable)`
    - Test that *iterable* contains no objects
  - `assertContentsInOrder(iterable, expected...)`  
`assertContentsInAnyOrder(iterable, expected...)`
    - Test that *iterable* contains exactly all of the remaining arguments in exact/any order and nothing else

See also: <http://developer.android.com/reference/android/test/MoreAsserts.html>

# Mock Objects

---

- Objects that *implement an interface* (i.e. they look like the interface)
- But results of functions may be pre-defined (i.e. behavior is deterministic)

# Mock Objects: Example

---

```
public interface ProxySettings {  
    public abstract String fetchWithProxy(URL url);  
}
```

```
public class Weather {  
    public static String fetchCurrentWeather(  
        String place, ProxySettings proxy) {  
        /* ... */  
        return proxy.fetchWithProxy(url);  
    }  
}
```

```
ProxySettings proxy = new DeviceProxySettings();  
String s = Weather.fetchCurrentWeather("Mumbai", proxy);  
Assert.assertEquals(s, "Rain");
```

# Mock Objects: Example

---

*Problem: DeviceProxySettings is device-specific!*

*Also, if I don't use a proxy, I can't test!*

*Solution: Make a mock object (class: MockProxySettings)!*

# Mock Objects: Example

---

```
public class MockProxySettings implements ProxySettings {  
    public String fetchWithProxy(URL url) {  
        /* Fetch without a proxy! */  
    }  
}
```

```
ProxySettings proxy = new MockProxySettings();  
String s = Weather.fetchCurrentWeather("Mumbai", proxy);  
Assert.assertEquals(s, "Rain");
```

**NOTE:** *What MockProxySettings does isn't important. What we care about is that fetchCurrentWeather works with a class that behaves like a ProxySettings interface.*



# Unit Testing with JUnit

# Making a TestCase

---

```
public class Email {
    private String mSubject; // And so on...

    public Email(String from, String subject, String body) {
        mSubject = subject;
        // And so on...
    }

    public String getSubject() {
        return mSubject;
    }
}
```

# Making a TestCase

---

```
public class EmailTestCase extends TestCase {
    public Email myEmail;

    protected void setUp() throws Exception {
        super.setUp();
        /* Set up the objects which will be tested. */
        myEmail = new Email("From", "Subject", "Body");
    }

    protected void tearDown() throws Exception {
        super.tearDown();
        /* Destroy the objects that were tested. */
    }
}
```

# Adding Tests

---

```
public class EmailTestCase extends TestCase {  
    public void testSubject() {  
        /* Testing that getSubject() returns what we expect. */  
        assertEquals(myEmail.getSubject(), "Subject");  
    }  
}
```

***All tests start with test!***

***NOTE: TestCase extends [is a child class of] Assert, so assertEquals may be called without referring to Assert.***

# Collecting Tests in a TestSuite

---

```
public class MyTestSuiteBuilder {  
    public static Test suite() {  
        /* Create a TestSuite and add each test. */  
        TestSuite suite = new TestSuite();  
        suite.addTest(new EmailTest("testSubject"));  
        return suite;  
    }  
}
```

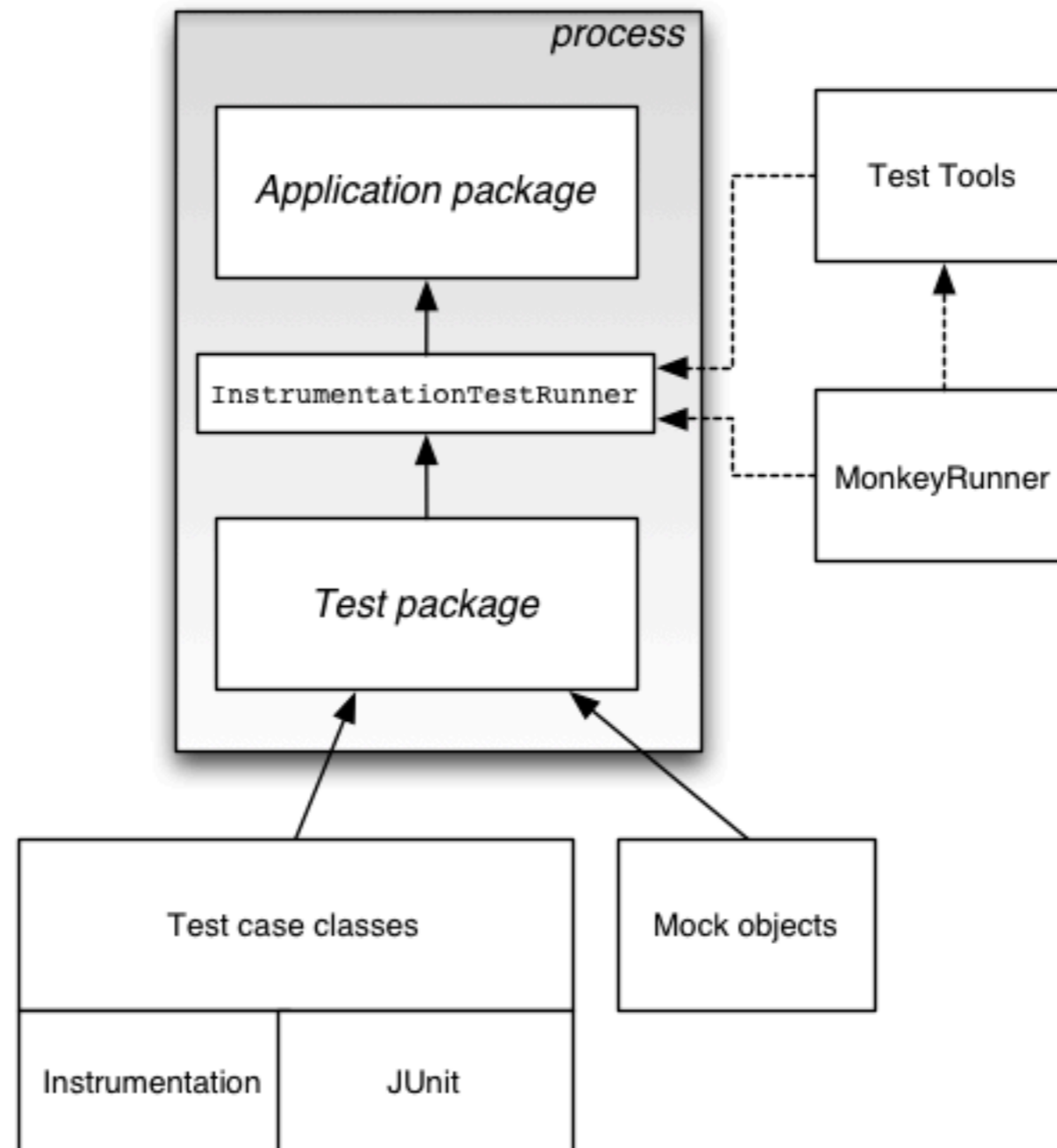
*or*

```
public static Test suite() {  
    /* Create a TestSuite and add each test. */  
    TestSuite suite = new TestSuite(EmailTest.class);  
    return suite;  
}
```

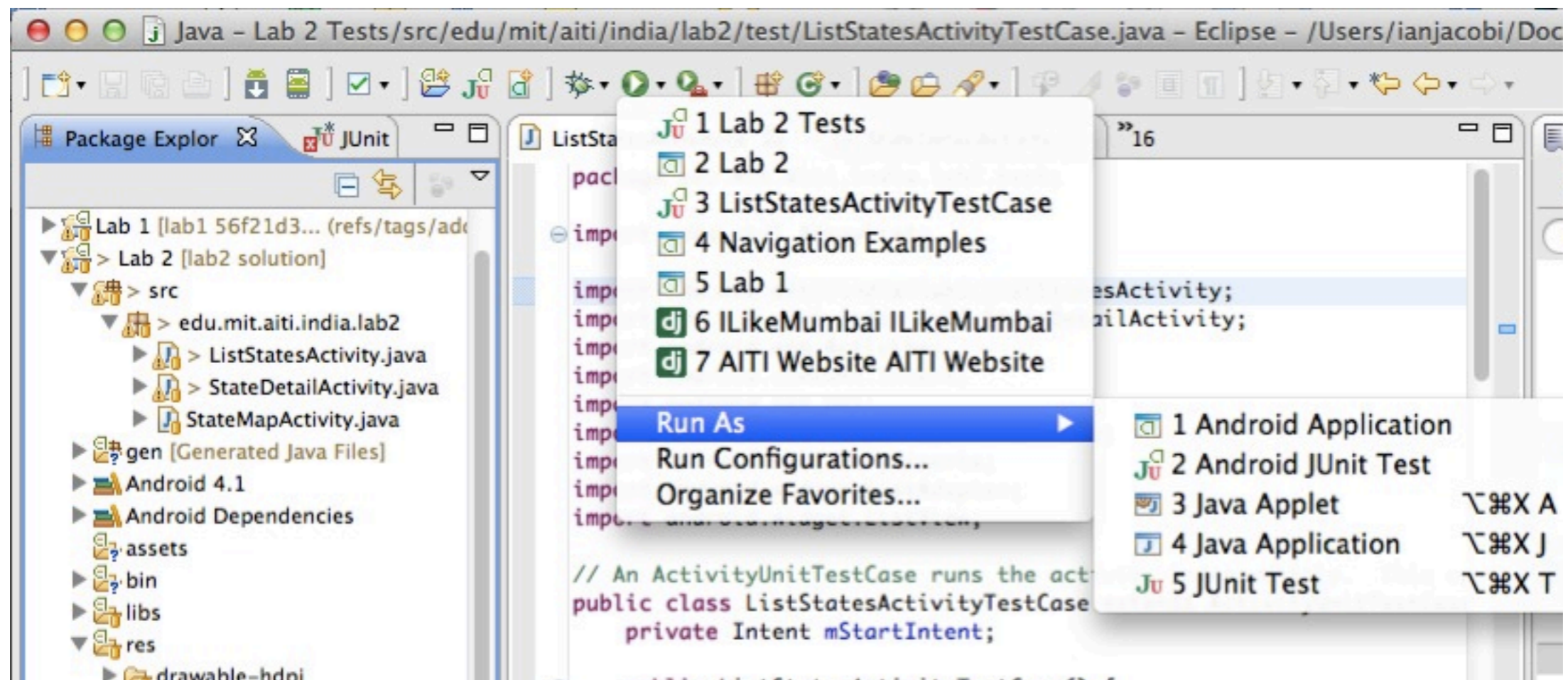
# Unit Testing on Android

# Unit Testing on Android

---



# Unit Testing in Eclipse

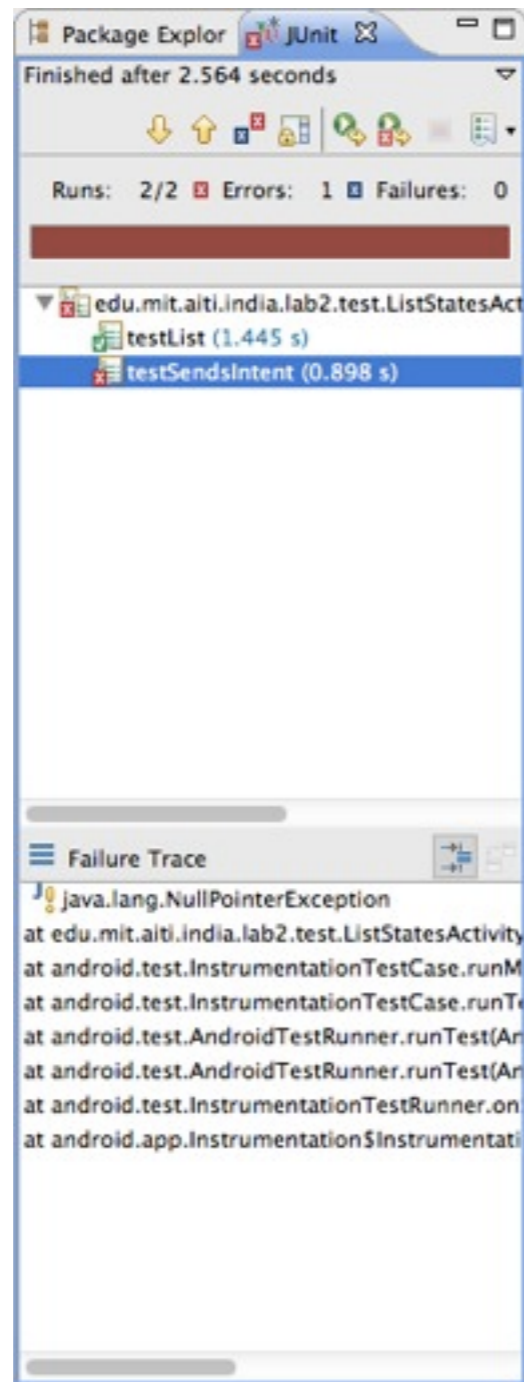


*Run like any other app*



# Unit Testing in Eclipse

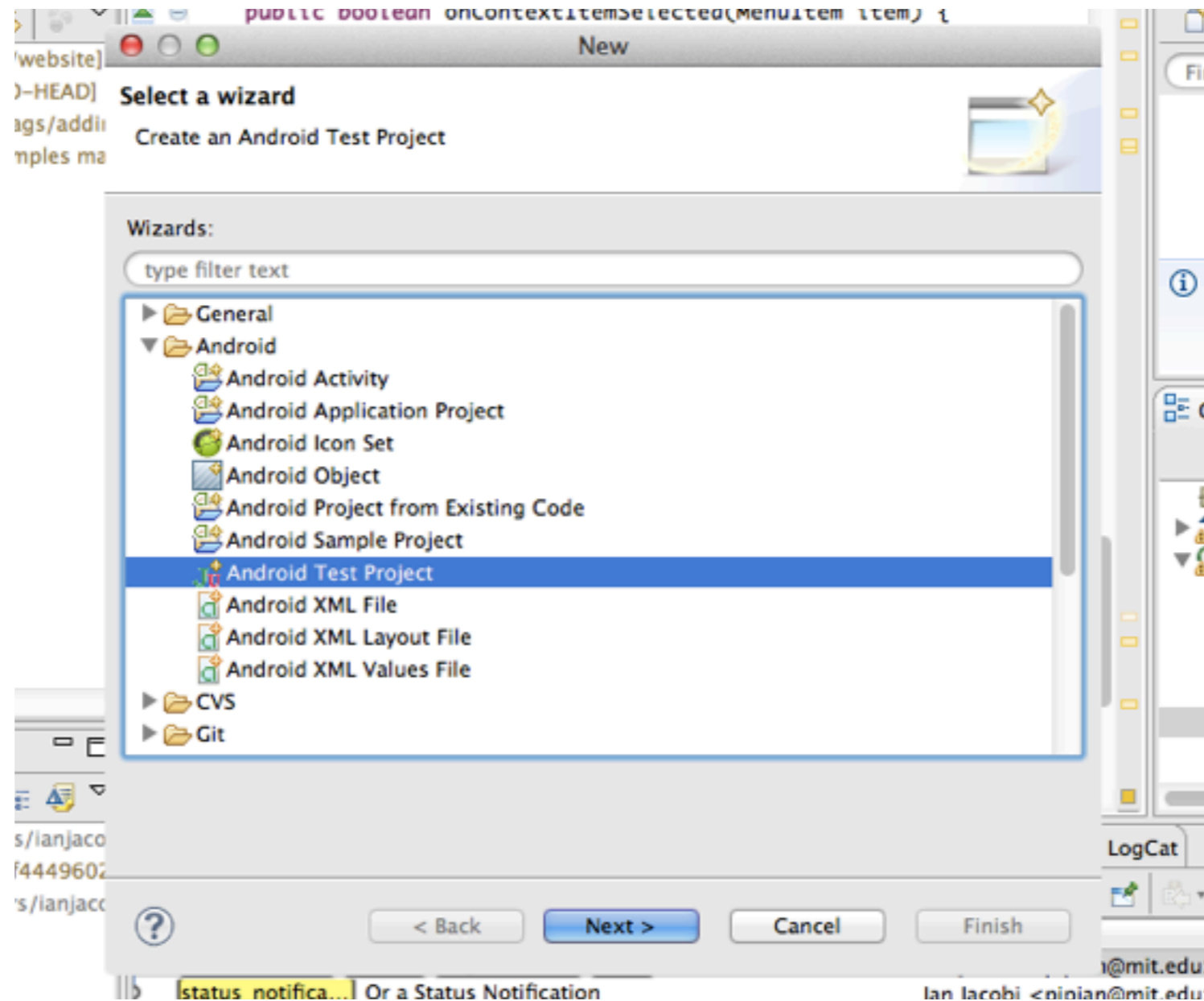
---



*View tests which fail*

*And why*

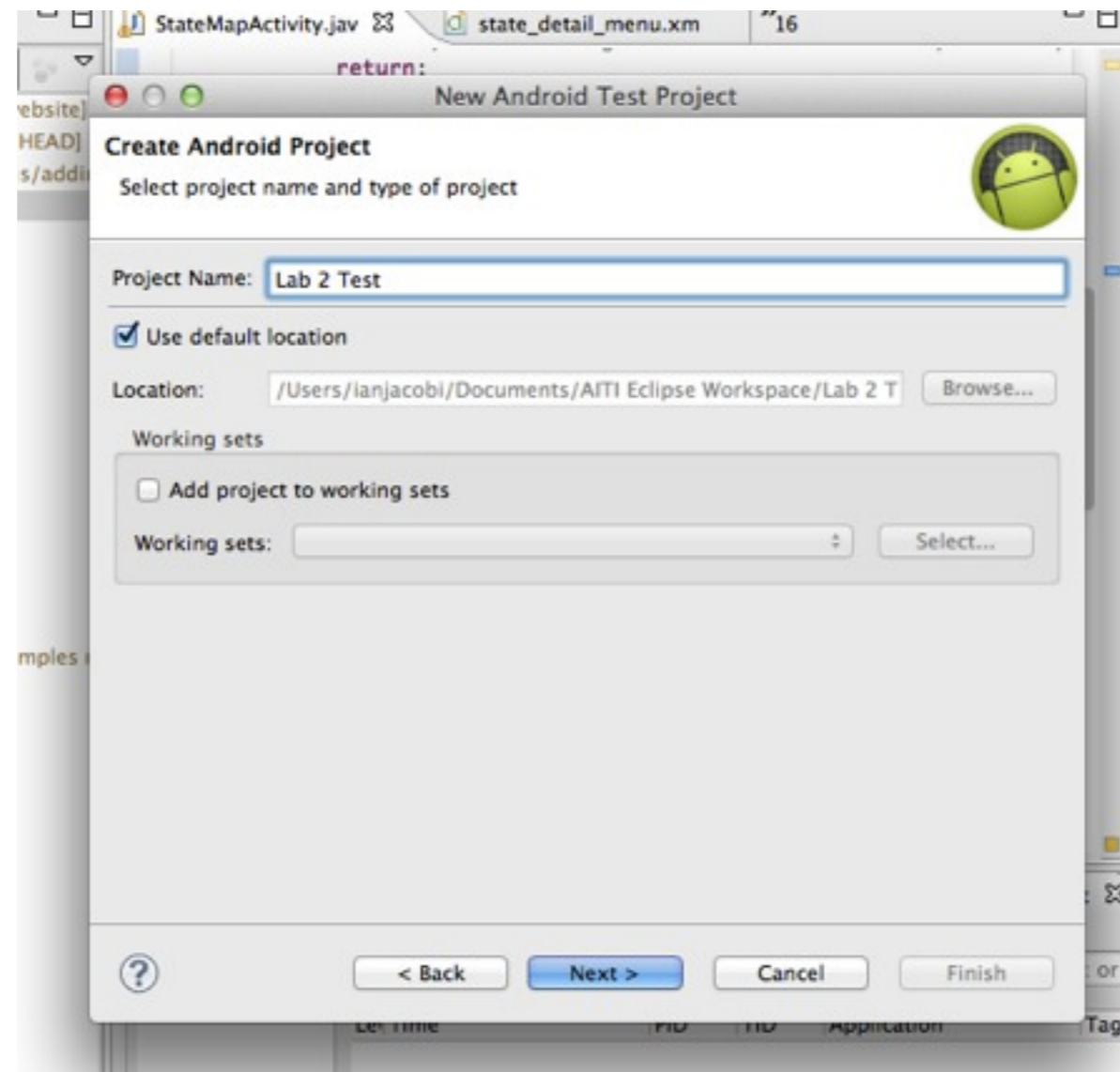
# Creating Unit Tests in Eclipse



*Make an Android Test Project*

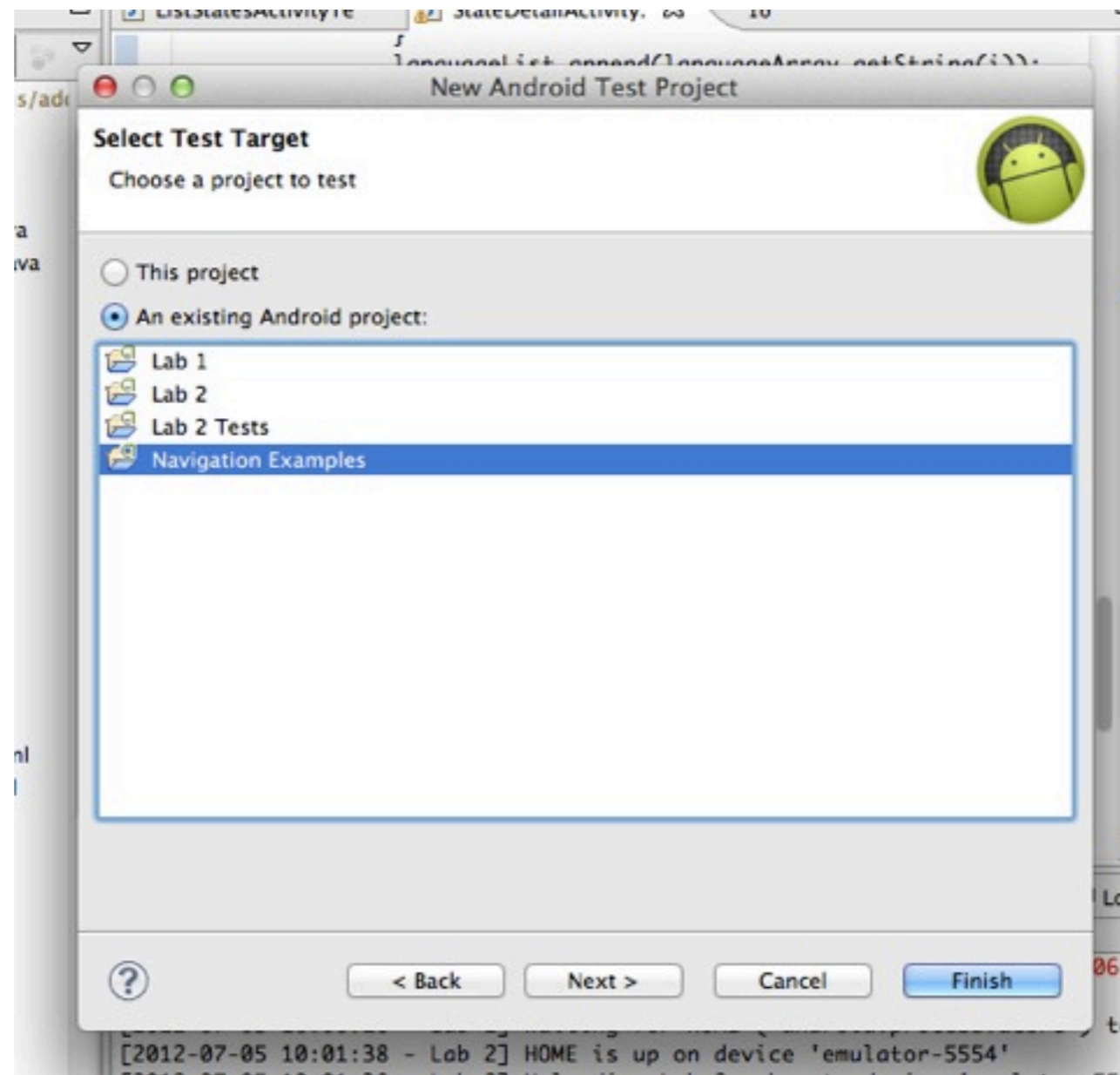
# Creating Unit Tests in Eclipse

---



*Name it after your project  
(Yes, this means you need 2 git repositories)*

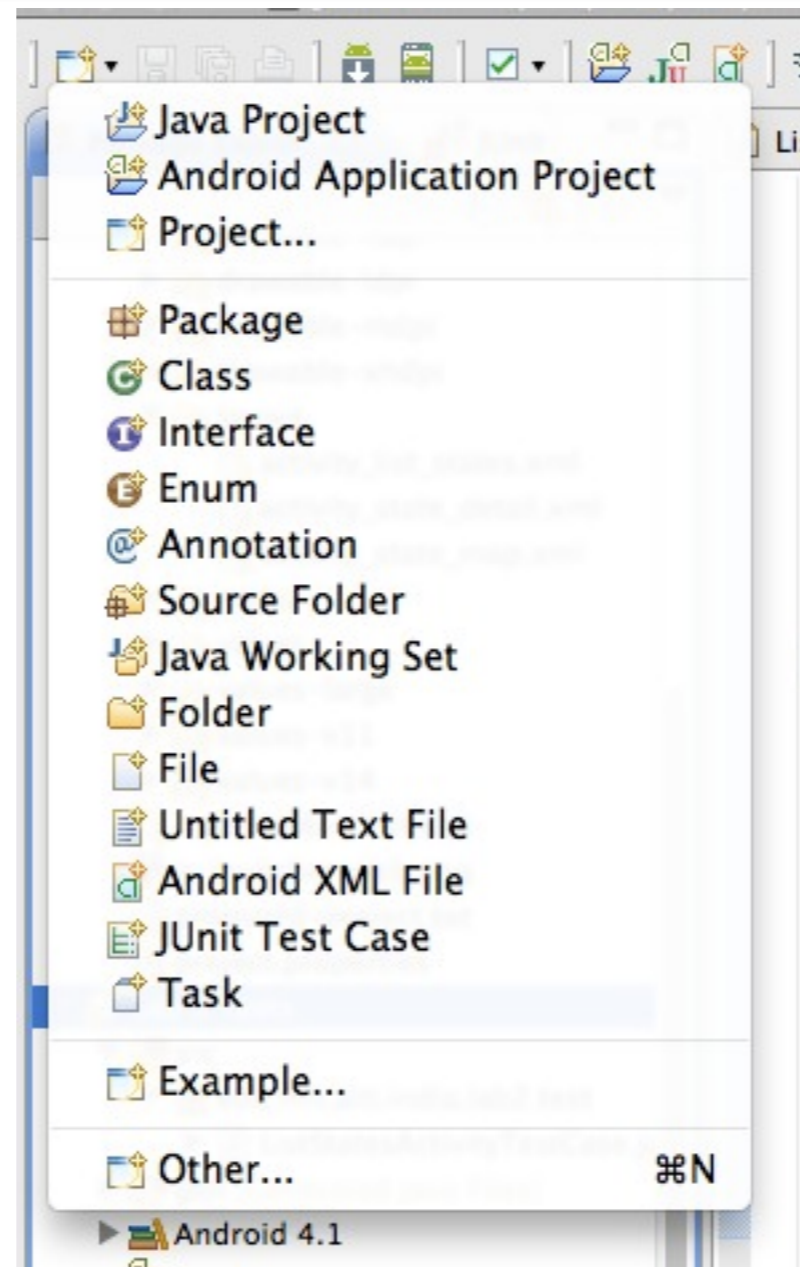
# Creating Unit Tests in Eclipse



*Choose the project to test*

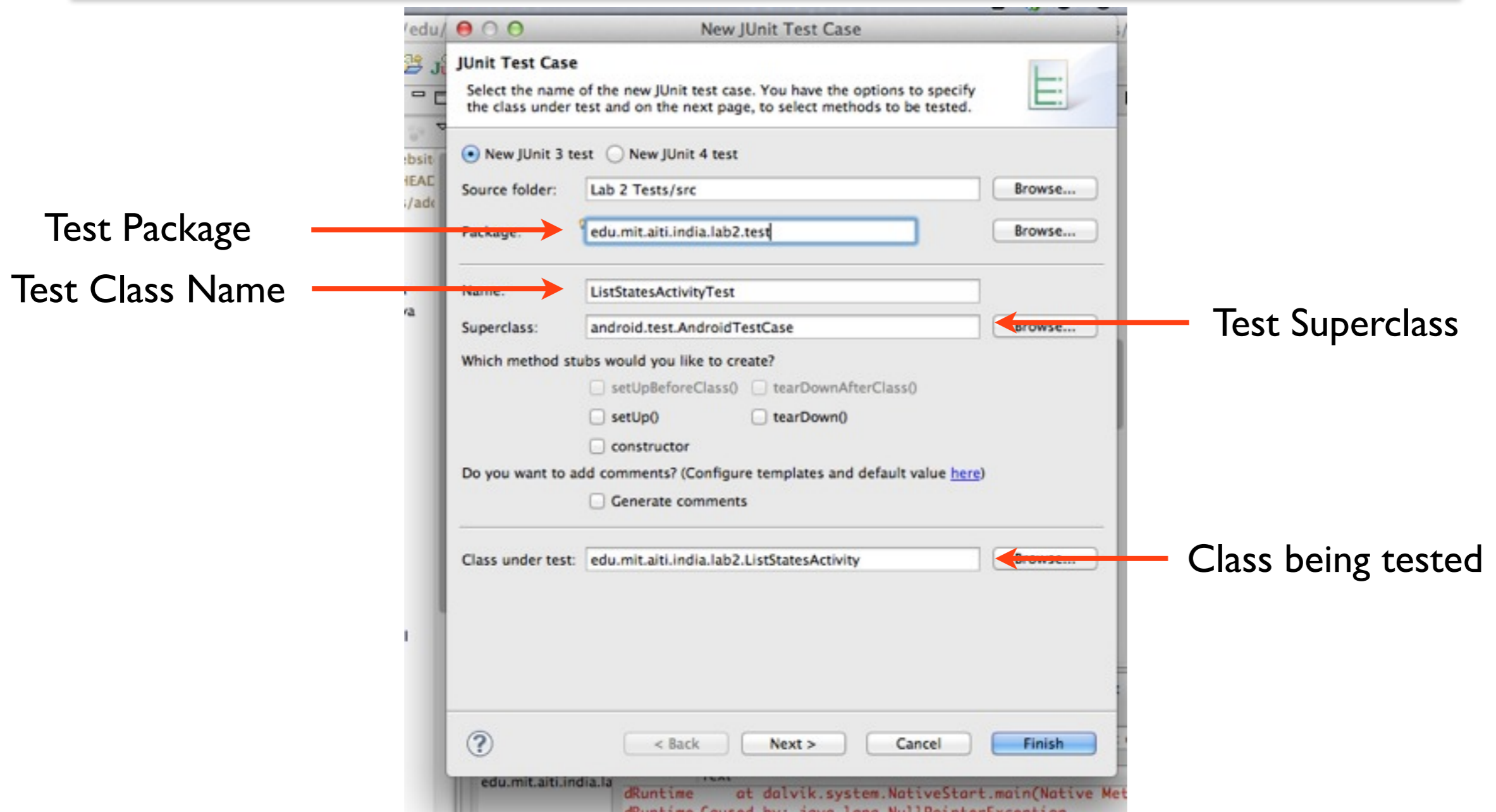
# Creating Unit Tests in Eclipse

---



*Add a new JUnit Test Case*

# Creating Unit Tests in Eclipse



*Set the properties of the Test Case*

# Aside: Test Case Superclasses

---

- `TestCase` – JUnit default (not for Android classes)
- `AndroidTestCase` – Android default
- `ActivityInstrumentationTestCase2<T>` – Test Activities in Android environment
- `ActivityUnitTestCase<T>` – Test Activities in standalone environment (e.g. to test Intents)
- `ServiceTestCase` – Test Services
- `ProviderTestCase2` – Test Content Providers

# What's with the `<T>`?

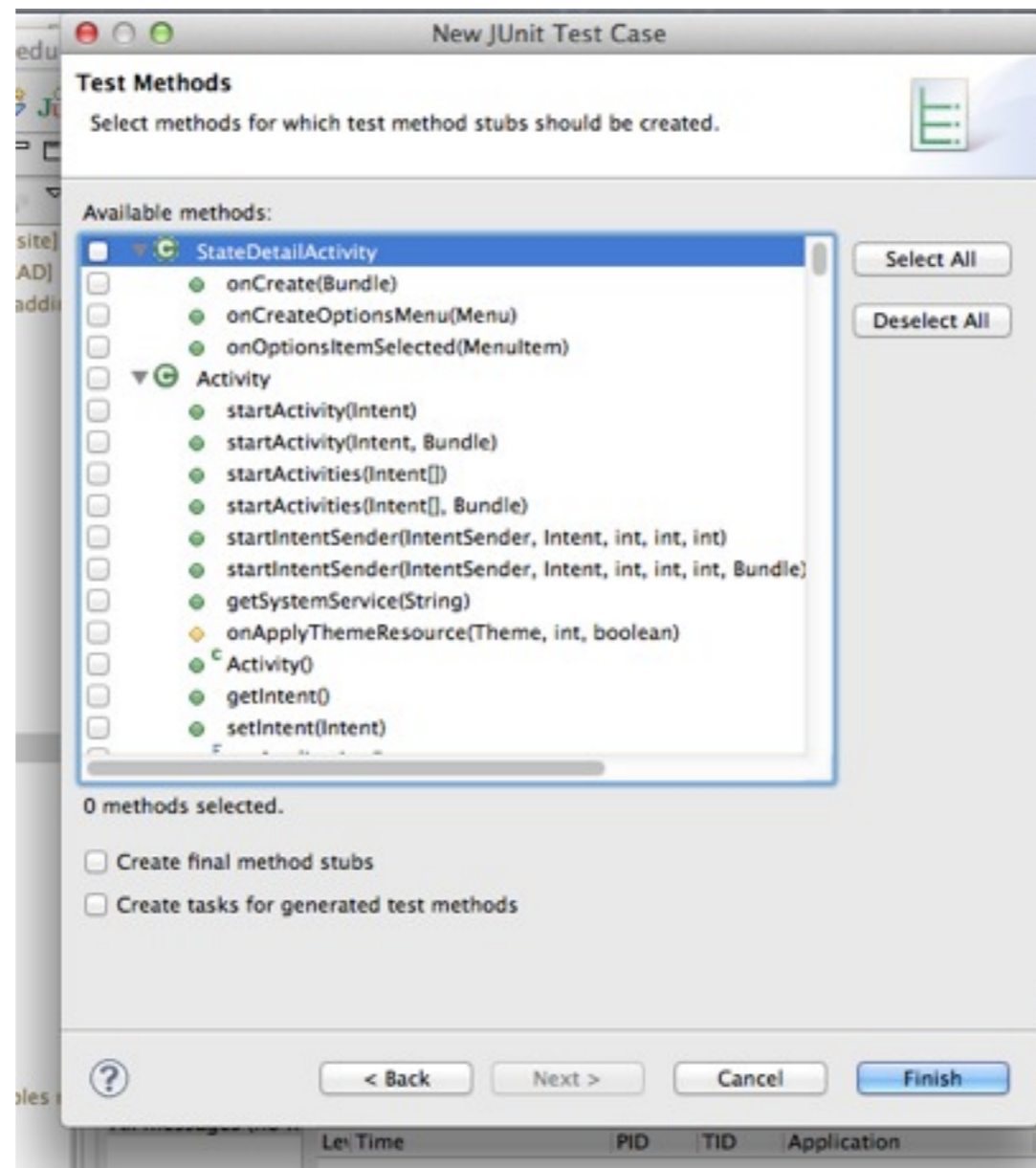
---

- `<T>` is used in “generic templates”
- `T` substituted with a class (e.g. `<MyActivity>`)
- Used with classes to specify the type of class being operated on
  - (e.g. `ArrayList<String>` is an `ArrayList` of `String` objects)



# Creating Unit Tests in Eclipse

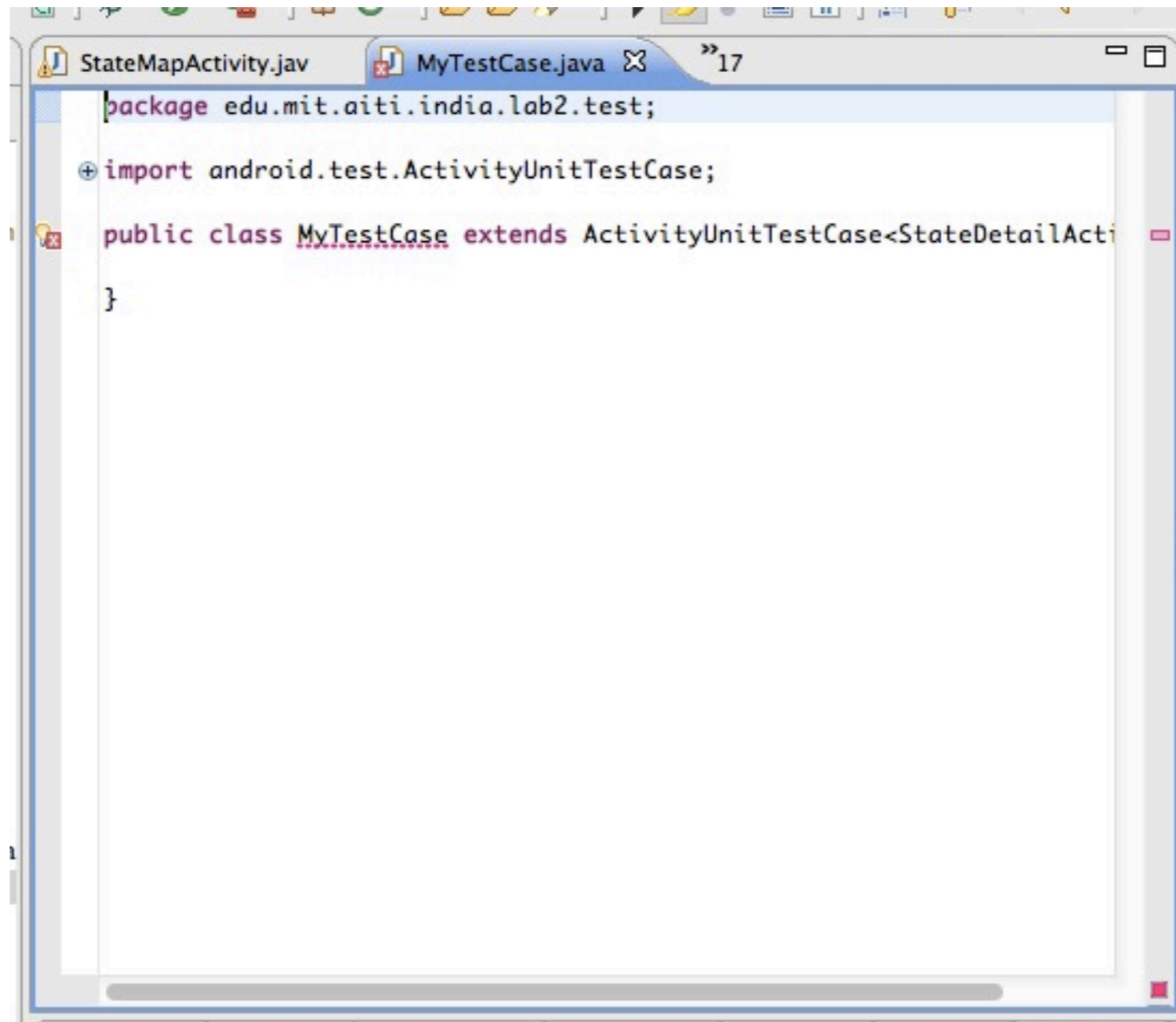
---



*Select what functions you want to test (if any)*

# Creating Unit Tests in Eclipse

---



```
StateMapActivity.jav  MyTestCase.java  »17
package edu.mit.aiti.india.lab2.test;

import android.test.ActivityUnitTestCase;

public class MyTestCase extends ActivityUnitTestCase<StateDetailActi
}
}
```

*Basic Unit Test created!*

**A Live Activity Unit Test!**

# References

---

- HttpURLConnection (Android APIs)  
<<http://developer.android.com/reference/java/net/HttpURLConnection.html>>
- Simple HttpURLConnection example  
<<http://digiassn.blogspot.in/2008/10/java-simple-httpurlconnection-example.html>>
- How do you GET/POST? See Tim Bray's "HttpURLConnection's Dark Secrets"  
<<http://www.tbray.org/ongoing/When/201x/2012/01/17/HttpURLConnection>>
- JUnit Cookbook  
<<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>>
- Android Testing  
<<http://developer.android.com/tools/testing/index.html>>
- Unit Testing Best Practices  
<<http://www.bobmccune.com/2006/12/09/unit-testing-best-practices/>>