Introduction to JUnit

Dr. John H Robb, PMP, SEMC
UTA Computer Science and Engineering

Please download:

Eclipse IDE for Java Developers

https://www.eclipse.org/downloads/packages/release/2018-09/r/eclipse-ide-java-develo

Performing JUnit Testing

- For this example we will create some very simple code to show you how to setup JUnit in Eclipse, test it, and some basic functions associated with JUnit

- JUnit has many nice features - it obviates the need to create drivers and harnesses in the code - it performs all this automatically. Advanced features of JUnit allow the tester to create suites of tests that can be run and results automatically checked.

- We're going to create some code that tells us what the next meal is based on the current meal. We'll also create some code that tells us what the next meal we can eat if we decide to skip successive meals.

- I recommend that you capture the code in these examples and run JUnit as shown to ensure that you understand how to use it.

Code for JUnit Example

- The following is the code for the MealNames enumeration in Java

- This code is in a class MealNames.java as follows

MealNames.java

public enum MealNames {*breakfast, lunch, supper}*

The code will be attached to slide M12 in blackboard

Code for JUnit Example (cont.)

The following code is in Meals.java

```
public class Meals {

MealNames meal = MealNames.breakfast;

 public void nextMeal() {

switch (meal) {
   case breakfast:
     meal = MealNames.lunch;
     break;
   case lunch:
     meal = MealNames.supper;
     break;
   case supper:
     meal = MealNames.breakfast;
     break;
   default:
     meal = MealNames.breakfast;
   }}

 public void skipmultiplemeals(int numberOfMeals) {
 for (int i = 0; i < numberOfMeals; i++)
   nextMeal();}}
```

Create the Eclipse Project and Source files

Create an Eclipse project containing Meals.java and MealNames.java

Package Explorer

- M09
- M10
- M11
- M12
  - src
    - (default package)
      - MealNames.java
      - Meals.java
      - MealsTest.java
  - JRE System Library [JavaSE-1.8]
  - JUnit 4
- Test

Meals.java

```
1  public class Meals {
2
3      MealNames meal = MealNames.breakfast;
4
5        public void nextMeal() {
6
7          switch (meal) {
8          case breakfast:
9            meal = MealNames.lunch;
10           break;
11         case lunch:
12           meal = MealNames.supper;
13           break;
14         case supper:
15           meal = MealNames.breakfast;
16           break;
17         default:
18           meal = MealNames.breakfast;
19         }
20       }
21
22       public void skipmultiplemeals(int numberOfMeals) {
23
24           for (int i = 0; i < numberOfMeals; i++)
25             nextMeal();
26
27       }
28 }
29
```

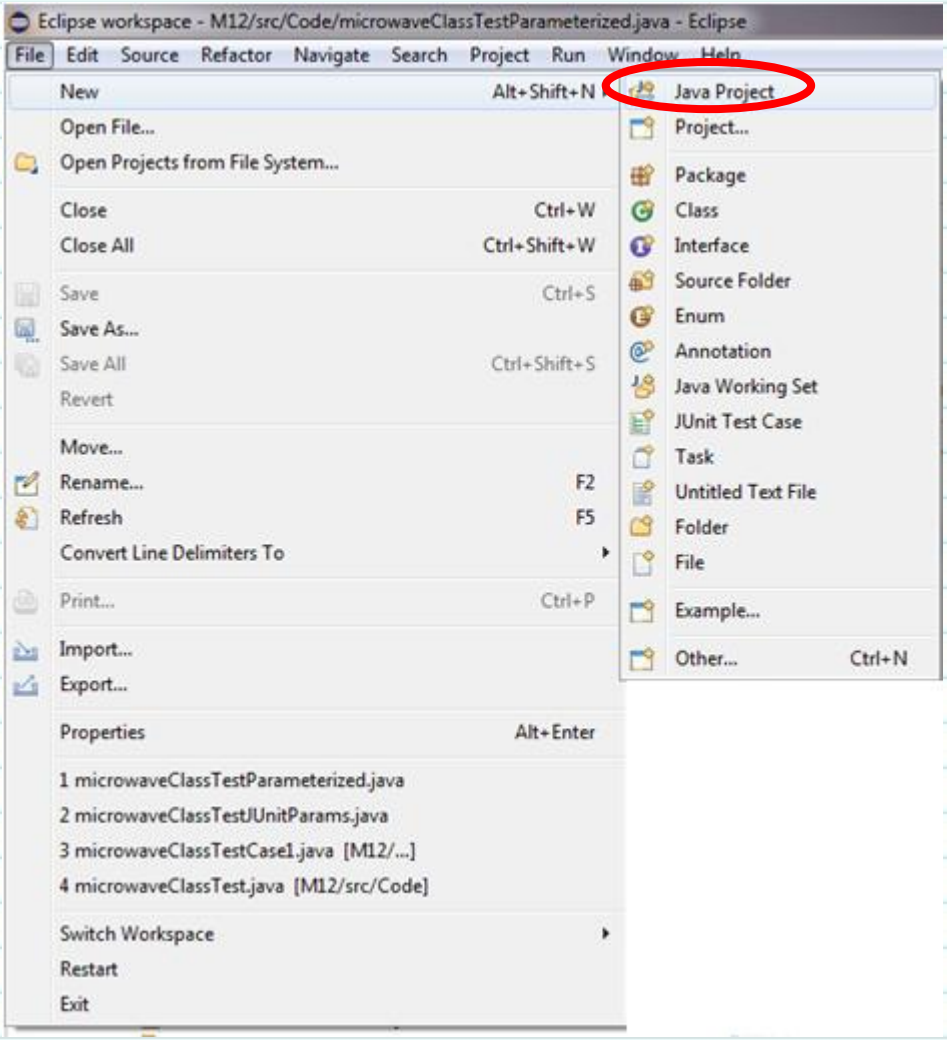
Follow the next set of slides to create this project

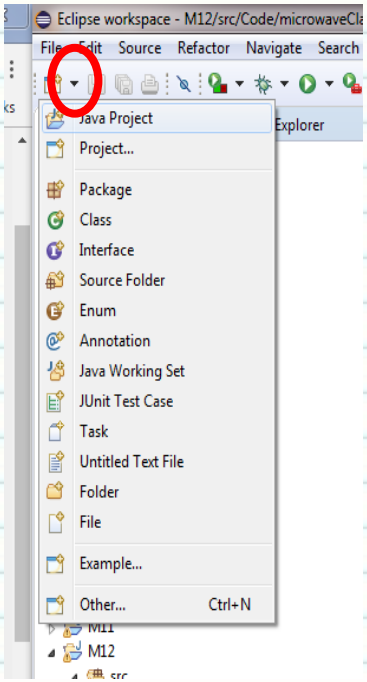
MealNames.java

```
1  public enum MealNames {breakfast, lunch, supper}
2
```

Creating an Eclipse Project

- To create a new Java Project perform the following:



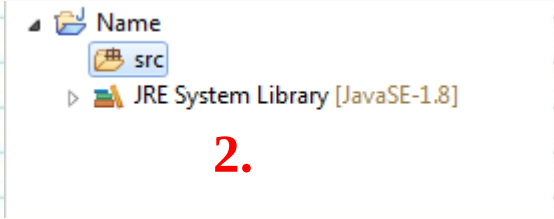


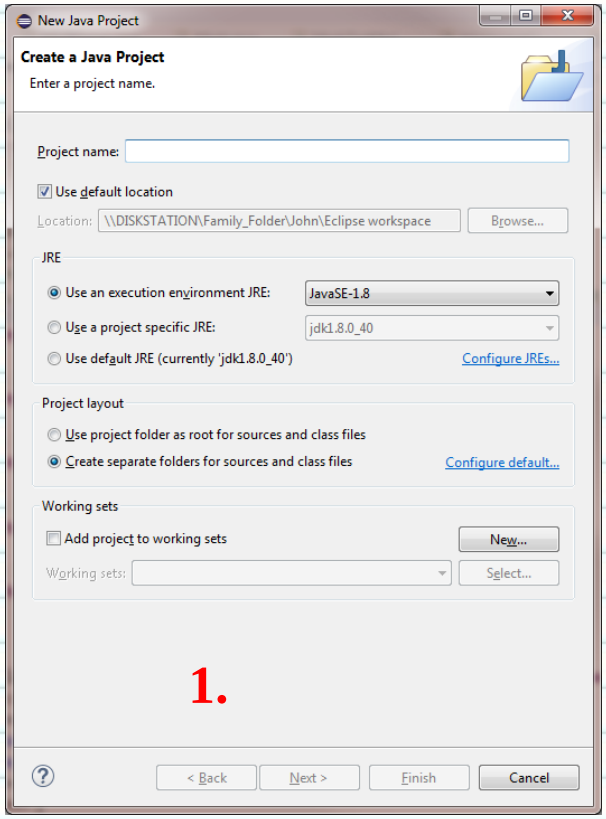
Use either approach

Creating an Eclipse Project (cont.)

1. Fill in the Project name and press Finish

2. Open the Project by clicking on the Expansion tab next to its name



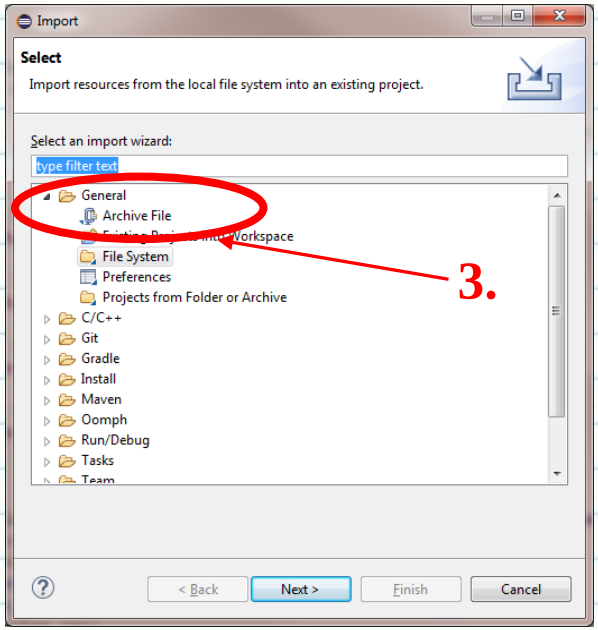


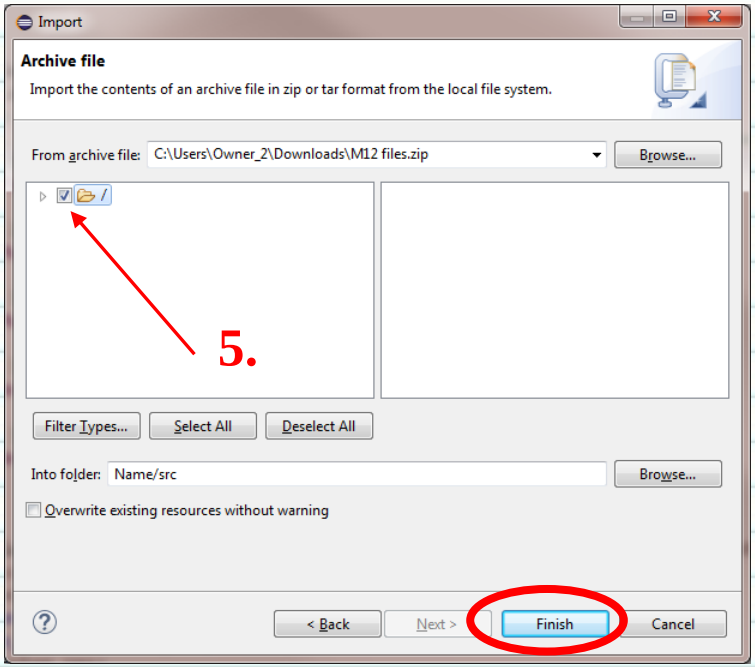
Creating an Eclipse Project (cont.)

3. Import - General - Archive File

4. Download the M12 files.zip from Blackboard - these should be in your downloads - don't unzip the folder

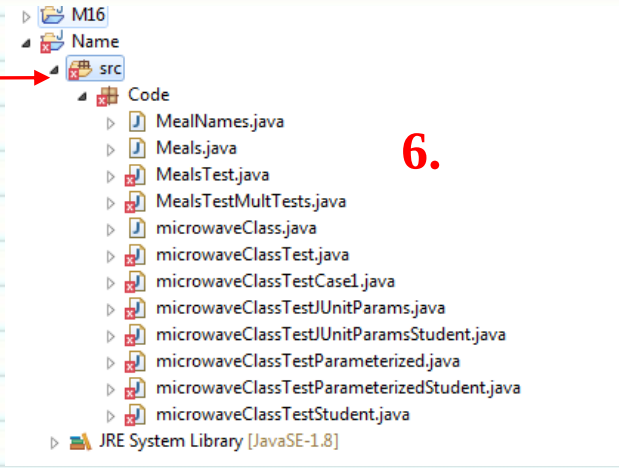
5. Click on the folder name and click Finish





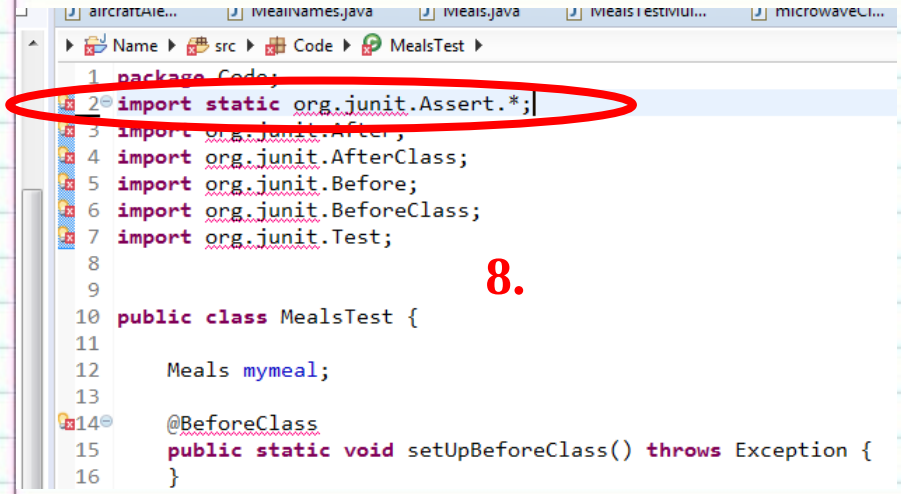
Creating an Eclipse Project (cont.)

6. Expand the src folder - you will have errors

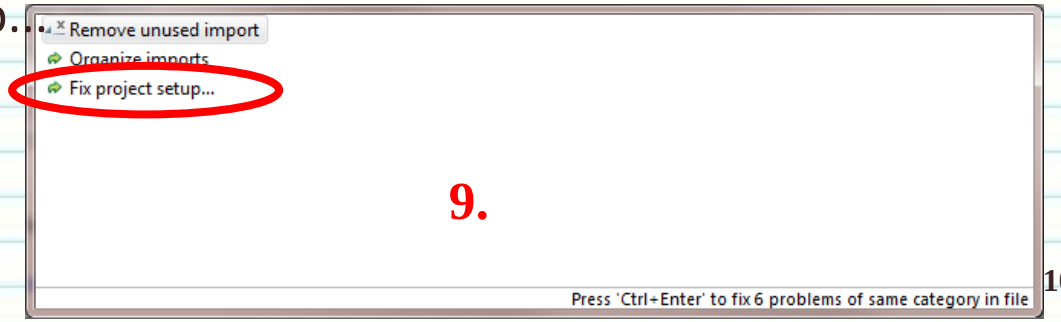


7. Double click the file MealsTest.java in the package explorer

8. Click on the first red x on statement 2

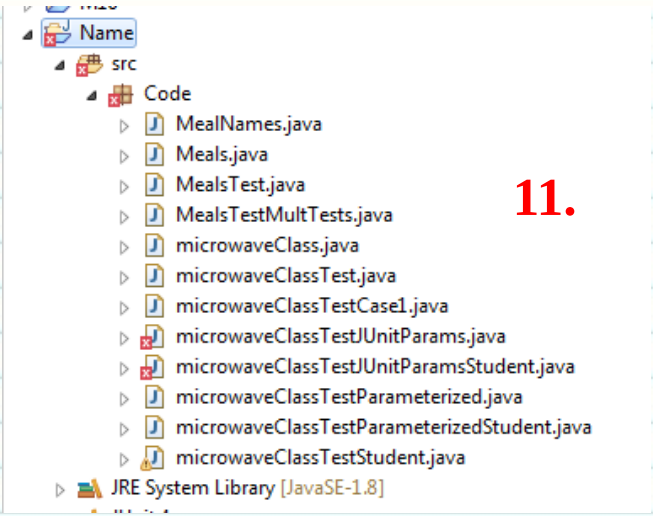


9. Select "Fix Project Setup..."

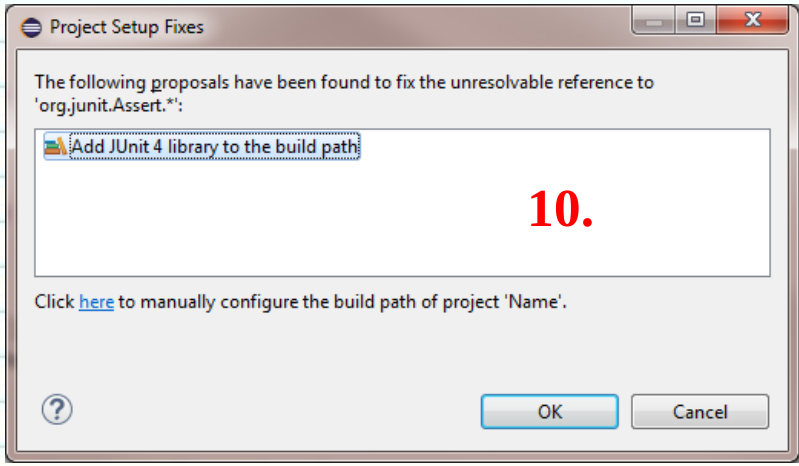


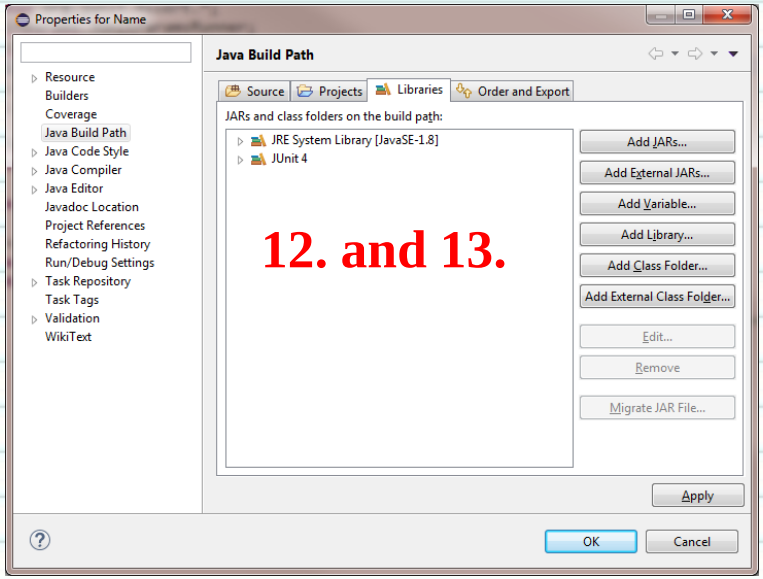
Creating an Eclipse Project (cont.)

10. Add JUnit 4 Library to the build path - click OK
11. download the JUnitParams-1.0.4.jar file from Blackboard



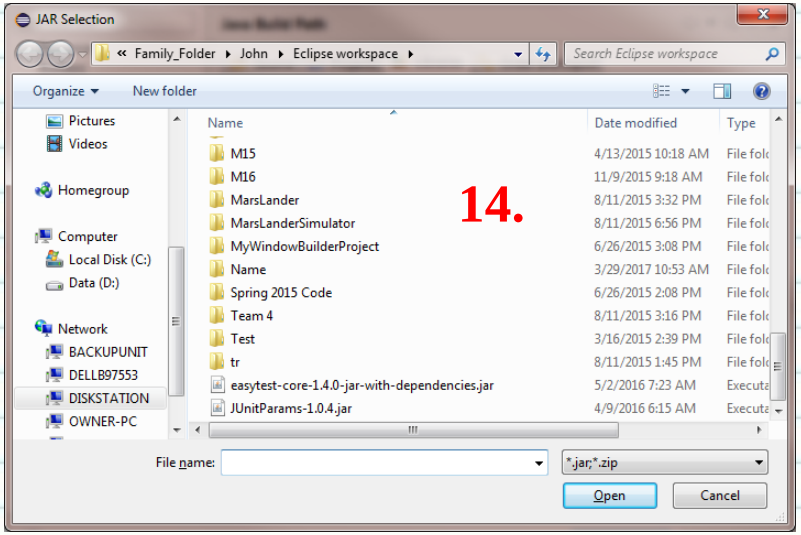
12. Right click on project - select Properties...
13. Select Java Build Path and "Add External JARs..."

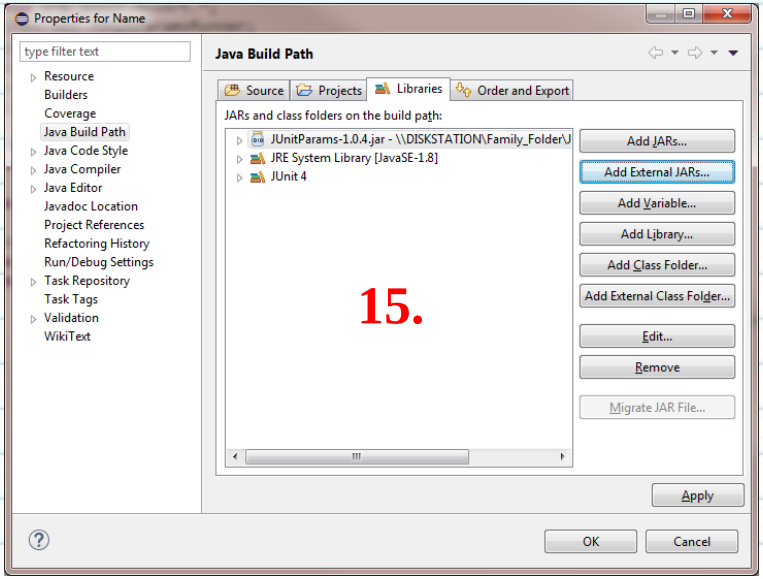




Creating an Eclipse Project (cont.)

14. Find the JUnitParams-1.0.4.jar you just downloaded
15. Double click on it





16. Click OK on the previous window
17. All red x's should disappear

Create the Test Code

- Now we're going to create the code to stimulate the inputs and check expected output using JUnit
- We will do this by taking the code JUnit automatically creates and replacing it with the code we need to setup the tests and check expected outputs.
- The following slide shows the code that JUnit creates to drive the tests
- We're going to replace the "fail" methods with the code shown next

The Correct Test Code for Our Example

```
@Test
 public void testMeals() {
        Meals mymeal = new Meals();
        assertEquals(MealNames.breakfast,mymeal.meal);
        mymeal.nextMeal();
        assertEquals(MealNames.lunch,mymeal.meal);

//      Remove the comment delimiters below to raise a wrong assertion to show the exception raised
//      assertEquals(MealNames.breakfast,mymeal.meal);
        mymeal.nextMeal();
        assertEquals(MealNames.supper,mymeal.meal);

//      Remove the comment delimiters below to raise a wrong assertion to show the exception raised
//      assertEquals(MealNames.lunch,mymeal.meal);
        mymeal.nextMeal();
        assertEquals(MealNames.breakfast,mymeal.meal);

//      Remove the comment delimiters below to raise a wrong assertion to show the exception raised
//      assertEquals(MealNames.lunch,mymeal.meal);
}

 @Test
 public void testSkipmultiplemeals () {
        Meals mymeal=new Meals();
        mymeal.skipmultiplemeals(4);
        assertEquals(MealNames.lunch,mymeal.meal);

//      Remove the comment delimiters below to raise a wrong assertion to show the exception raised
//      assertEquals(MealNames.breakfast,mymeal.meal);
```

Starting about half-way down the MealsTest.java class are two test methods (shown to the left)

What is the Test Code Doing?

- JUnit has several methods to assist with unit testing.
- Our test code is using the **assertEquals** method. This works by having JUnit automatically check that the two values being compared are equal.
- If they are equal the test passes. If not, an exception is raised.
- Here is an example of the test code that we're using to perform these checks.

expected value **actual value**

```
assertEquals(MealNames.lunch,mymeal.meal);
```

- It's checking to see if the lunch attribute in the mymeal object is equal to the enumeration value of lunch. **Make sure to get these in the correct order!**

Executing the JUnit Tests

- From the Package Explorer, right-click (in Windows) the MealsTest.java file.
- In the context menu, select Run As>JUnit Test.
- The next slide shows the successful result.

What a Successful JUnit Test Run Looks Like

Package Explorer | JUnit

Runs: 2/2 Errors: 0 Failures: 0

MealsTest [Runner: JUnit 4] (0.026 s)

MealsTest has two @test assert methods, so the Runs: 2/2 tells us that both ran successfully

Failure Trace

MealsTest.java

```java
import static org.junit.Assert.*;


public class MealsTest {

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

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testNextMeal() {

        Meals mymeal = new Meals();
        assertEquals(MealNames.breakfast,mymeal.meal);
        mymeal.nextMeal();
        assertEquals(MealNames.lunch,mymeal.meal);

        //  Remove the comment delimiters below to raise a wrong assertion to show the exception rai
        //  assertEquals(MealNames.breakfast,mymeal.meal);
        mymeal.nextMeal();
        assertEquals(MealNames.supper,mymeal.meal);

        //  Remove the comment delimiters below to raise a wrong assertion to show the exception rai
        //  assertEquals(MealNames.lunch,mymeal.meal);
        mymeal.nextMeal();
```

An Unsuccessful JUnit Test Run

- I have added some commented lines in these tests so you can remove them and inspect what happens when an assertion is not satisfied.
- The following slide shows the result of removing one of these comment delimiters.
- The test fails and the line number where the exception is raised is identified. The test results from the comparison are also identified - the expected and actual results.

An Unsuccessful JUnit Test Run (cont.)

Package Explorer | JUnit

Finished after 0.063 seconds

Runs: 2/2 Errors: 0 Failures: 1

MealsTest [Runner: JUnit 4] (0.034 s)
- testSkipmultiplemeals (0.017 s)
- testNextMeal (0.017 s)

1 Failure

Failure Trace

java.lang.AssertionError: expected:<breakfast> but was:<lunch>

at MealsTest.testNextMeal(MealsTest.java:37)

Expected vs. actual output

Line number of exception

MealsTest.java

```
10 import static org.junit.Assert.*;
8
9
10 public class MealsTest {
11
12     @
13     p
14     }
15
16     @
17     p
18     }
19
20     @
21     public void setUp() throws Exception {
22     }
23
24     @After
25     public void tearDown() throws Exception {
26     }
27
28     @Test
29     public void testNextMeal() {
30
31         Meals mymeal = new Meals();
32         assertEquals(MealNames.breakfast,mymeal.meal);
33         mymeal.nextMeal();
34         assertEquals(MealNames.lunch,mymeal.meal);
35
36         //  Remove the comment delimiters below to raise a wrong assertion to show the excep
37         assertEquals(MealNames.breakfast,mymeal.meal);
38         mymeal.nextMeal();
39         assertEquals(MealNames.supper,mymeal.meal);
40
```

When the exception is raised JUnit stops execution at that point in the failed test method

Expected output

Actual output

The order is important so that the assertion error message is correct

Assert methods

JUnit Assertions

- assertTrue()
- assertFalse()
- assertNull()
- assertNotNull()
- assertEquals()
- assertSame()
- assertNotSame()

- **We will use assertEquals** - it works with longs, doubles, Strings, Objects - for doubles it requires a 3rd parameter - the comparison threshold
- assertEquals(expected value, actual value);
- JUnit will execute the statement and compare the expected with the actual value and if not the same will raise an exception
- We will use assertEquals to check expected values with actual values

JUnit Set Up

```
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class MealsTestMultTests {

        Meals mymeal;

        @Before
        public void setUp() {
                mymeal=new Meals();
        }

        @Test
        public void testNextMealTestCase1() {
        assertEquals(MealNames.breakfast,mymeal.meal);
        mymeal.nextMeal();
        assertEquals(MealNames.lunch,mymeal.meal);
```

Put object constructors here in the @Before class in the setUp method

This means it will construct a new object before each test

We can't use the @BeforeClass because its static

Multiple Test Cases

```
...

@Test
public void testNextMealTestCase1() {
mymeal.setMeal(MealNames.breakfast);
assertEquals(MealNames.breakfast,mymeal.meal);
mymeal.nextMeal();
assertEquals(MealNames.lunch,mymeal.meal); }

@Test
public void testNextMealTestCase2() {
mymeal.setMeal(MealNames.lunch);
mymeal.nextMeal();
assertEquals(MealNames.supper,mymeal.meal); }

@Test
public void testNextMealTestCase3() {
mymeal.setMeal(MealNames.supper);
mymeal.nextMeal();
assertEquals(MealNames.breakfast,mymeal.meal); }

@Test
public void testSkipmultiplemeals() {
        mymeal.setMeal(MealNames.breakfast);
        mymeal.skipmultiplemeals(4);
        assertEquals(MealNames.lunch,mymeal.meal); }
}
```

We can make each test case a separate JUnit test method - denoted by the @Test annotation

JUnit may run these in ANY order - you can't rely on one executing before another

This approach let's you run all test cases - if they have an error then it starts the next test method

Multiple Test Cases (cont.)

```
...

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class MealsTestMultTests {

    Meals mymeal;

    @Before
    public void setUp() {
        mymeal=new Meals();
    }

...
```

We can order the test methods by using the @FixMethodOrder annotation

This will order by test method names

This must go before the test class declaration

Understanding JUnit Tests

Each test

1. **Sets up input**
2. **Invokes the method under test**
3. **Checks expected outputs with actuals**

```
…

@Test
public void testNextMealTestCase1() {
mymeal.setMeal(MealNames.breakfast);
mymeal.nextMeal();
assertEquals(MealNames.lunch,mymeal.meal); }

…
```

For the simple example we are setting the mymeal.meal attribute to breakfast

We invoke the method under test

We check that mymeal.meal equals lunch

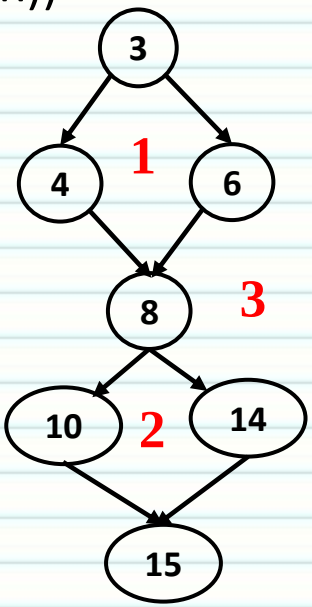
Student Exercise

JUnit Exercise

- Double click on microwaveClassTestCase1.java in Eclipse
  1. Look at test case 1 - the input and expected output values are in the comments for test case 1
  2. test case 1 also contains the JUnit script code - we're setting inputs and checking expected with actual outputs
  3. create test case 2 by copying and pasting the code from test case 1
     a) update inputs and expected outputs
  4. Execute the test! Update until it passes

Achieving Condition Coverage (Example 4)

```
1   public void operateMicrowave (boolean cooking, boolean doorOpen,
                                                boolean stopButton) {
2
3   if (cooking && (doorOpen || stopButton))
4       cookState=true;
5   else
6       cookState=false;
7
8   if (timer>0)
9       {
10      stop=false;
11      timer--;
12      }
13  else
14      stop=true;
15  }
```

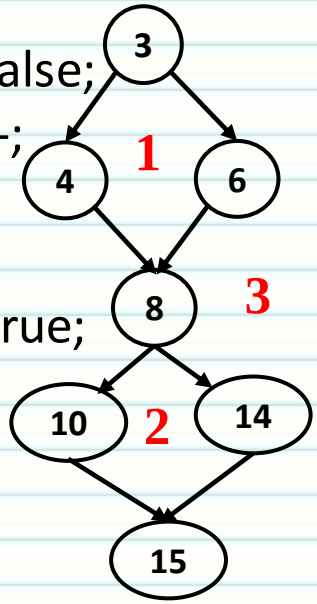


cookState, stop, and timer are private class variables

Achieving Condition Coverage (Example 4 cont.)

```
1   public void operateMicrowave (boolean cooking, boolean doorOpen,
                                  boolean stopButton) {
2
3   if (cooking && (doorOpen || stopButton))
4       cookState=true;
5   else
6       cookState=false;
7
8   if (timer>0)
9       {
10      stop=false;
11      timer--;
12      }
13  else
14      stop=true;
15  }
```

Test Case	Inputs				Exp Out			Basis Path Tested
	cooking	doorOpen	stopButton	timer	cookState	stop	timer	
1	T	F	T	1	T	F	0	3-4-8-10-15
2	T	F	F	1	F	F	0	3-6-8-10-15
3	T	F	F	0	F	T	0	3-6-8-14-15
3a	T	T	F	0	T	T	0	-
3b	F	T	F	0	F	T	0	-



1) MCDC solution for a(b + c)=TFT,TFF,TTF,FTF
2) We have combined MCDC with basis path as shown above
3) The order for basis path forces us to choose statement 3 first true then false - so we match the MCDC up in order
4) Test case 3a abd 3b test the remaining 2 MCDC terms
5) All MCDC tests toggle only the COI between tests
6) Each test changes only one input at a time

Review Student Answers

JUnit Exercise #2

- Develop the remaining 4 test cases
  - The required values are shown in the comments
  - When you get a test completed run it and see if working
  - When you complete all 4 remaining tests and are passing you are complete with this exercise

Review Student Answers

JUnit Code Can Get Quite Lengthy

- Just with the 5 tests from the previous exercise the code can get to be quite lengthy
- **Lengthy and diffuse** tests like this are not good Software Engineering practice
  1. They do NOT communicate **succinctly** and **simply**
  2. There are **copy and paste errors** lurking in the JUnit code
  3. Others will have more code to search through to determine how your test works
- We want to utilize a **data driven approach** - this IS good software engineering
  1. we want to separate test data from test commands
  2. we want data to look like how we have been developing Excel driven test cases
  3. we want to spend our JUnit lines of code mostly on test data and very little on test commands

Data Driven Tests

- There are two main approaches to data driven testing in JUnit
  - Using the Parameterized.class (this is part of JUnit 4)
  - Using the JUnitParamsRunner.class (this is a standalone JAR)

- We will examine both approaches in the next few slides and will use the microwave tests as an example

- This is the only approach we will focus on for JUnit testing - data driven is the best method for developing powerful tests

Parameterized JUnit tests

```
...
@RunWith(Parameterized.class)
public class microwaveClassTestParameterized {
...
  @Parameters
  public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
      {true,  false,      true, 1,      true,       false,      0},
      {true,  false,      false,1,      false,      false,      0}

                    purposely blanked out

    }); }

  public microwaveClassTestParameterized(boolean cooking, boolean doorOpen, boolean stopButton,
            int timer, boolean cookState_res, boolean stop_res, int timer_res) {
    this.cooking=cooking;
    this.doorOpen=doorOpen;
    this.stopButton=stopButton;
    this.cookState_res=cookState_res;
    this.stop_res=stop_res;
    this.timer=timer;
    this.timer_res=timer_res;
  }
    @Test
    public void test() {
```

Parameterized tests allow us to specify the test data as parameters (here input and expected output)

The parameters to the class constructor show how to interpret the data in the collection above - the order of the data to each row in the data() method above.

Parameterized JUnit tests (cont.)

```
@Test
public void test() {
    micOven.setTimer(timer);
    micOven.operateMicrowave(cooking, doorOpen, stopButton);
    assertEquals(cookState_res,micOven.isCookState());
    assertEquals(stop_res,micOven.isStop());
    assertEquals(timer_res,micOven.getTimer());
    }
}
```

The command script section is very short - this is the entire set of tests

JUnit will take the parameters specified by the constructor and statically create a test for each row in the data() method

Parameterized JUnit tests (cont.)

- The required imports for the JUnit Test Class are important

```
package ParameterizedClass;

import static org.junit.Assert.*;

import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class PowerTwoTest {

...
```

These are in the file microwaveClassTestParameterizedStudent.java

Parameterized JUnit tests (cont.)

- General steps for creating a Parameterized.class method
  1. Annotate test class with @RunWith(Parameterized.class).
  2. Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
  3. Create a public constructor that takes in what is equivalent to one "row" of test data.
  4. Create an instance variable for each "column" of test data.
  5. Create your test case(s) using the instance variables as the source of the test data.
- The test case will be invoked once for each row of data.

Student Exercise

- Use the file microwaveClassTestParameterizedStudent.java
  1. Complete all five test cases from the data developed in the previous student exercise (see overhead)
  2. Add the data to the Collection as the first two rows are done. Make sure to place commas in the correct places when you add rows
  3. Execute the tests

Student Exercise (cont.)

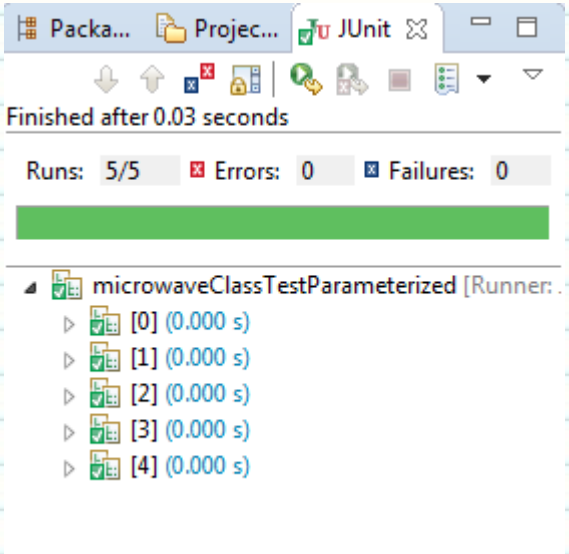
```
...
@RunWith(Parameterized.class)
public class microwaveClassTestParameterized {
...
  @Parameters
  public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
      {true,  false,  true, 1,  true,  false,  0},
      {true,  false,  false,1,  false, false,  0},
      {true,  false,  false,0,  false, true,   0},
      {true,  true,   false,0,  true,  true,   0},
      {false, true,   false,0,  false, true,   0}
    }); }

  public microwaveClassTestParameterized(boolean cooking, boolean doorOpen, boolean stopButton,
          int timer, boolean cookState_res, boolean stop_res, int timer_res) {
    this.cooking=cooking;
    this.doorOpen=doorOpen;
    this.stopButton=stopButton;
    this.cookState_res=cookState_res;
    this.stop_res=stop_res;
    this.timer=timer;
    this.timer_res=timer_res;
  }
    @Test
    public void test() {
```

Complete answer shown here

Notice that the data() method uses new Object[]

Object is the parent class to all classes in Java - this allows us to create an array of almost any mix of standard types

Student Exercise (cont.)



When you execute the 5 test cases and expand the JUnit run bar you will see each instance of the Collection row and its result.

Here we see 5 tests passed. The collector index is handy to let you know which test passed or failed.

The tests will run to completion regardless of getting failures - all tests will run.

How to Use Parameterized JUnit tests

- To create a parameterized.class test follow these steps
  1. Copy an existing parameterized.class JUnit test class
  2. Paste this in the new JUnit test class
  3. Correct name errors
  4. Correct the test inputs and expected outputs in the data() method
  5. Correct the constructor parameters
  6. Create variables needed by the constructor parameters
  7. Update the JUnit test command method

Parameterized. Class Summary

- This approach is built into JUnit 4+
  - It is one of the JUnit Runner classes
  - It can be used to test both instance and class (static) methods
  - The IDE (e.g., Eclipse) can help solve issues with its use since it is part of JUnit
  - It is much more efficient than the standard JUnit scripting language approach

JUnitParams Approach

JUnitParams

- JUnitParams is a widely used stand-alone JAR, it
  - is preferred over parameterized.class by advanced JUnit testers
  - is not part of JUnit
  - provides an even more succinct syntax than the parameterized.class
    - class variables and constructor are not needed
  - provides a more expansive feedback on the JUnit run bar
  - like parameterized.class, can be used to test both instance and class (static) methods

JUnitParams

```
...
@RunWith(JUnitParamsRunner.class)
public class microwaveClassTestJUnitParamsStudent {

    private microwaveClass micOven;

    @SuppressWarnings("unused")
    private static final Object[] parametersFormicrowaveClassTest () {
        return $(
//              Parameters are: (1,2,3,4,5)
//              1=cooking, 2=doorOpen, 3=stopButton, 4=timer, 5=cookState_res,6=stop_res,7=timer_res
//              Test case 1
                $(true,  false,  true, 1,        true,        false,        0),
//              Test case 2
                $(true,  false,  false,1,        false,       false,        0)
        );
    }

    @Before
    public void setUp () {
        micOven = new microwaveClass(false, false, 0);
    }

    @Test
    @Parameters(method="parametersFormicrowaveClassTest")
    public void test(boolean cooking, boolean doorOpen, boolean stopButton, int timer,
                                        boolean cookState_res, boolean stop_res, int timer_res) {

        micOven.setTimer(timer);
        micOven.operateMicrowave(cooking, doorOpen, stopButton);
        assertEquals(cookState_res,micOven.isCookState());
        assertEquals(stop_res,micOven.isStop());
        assertEquals(timer_res,micOven.getTimer());
    }}
```

the @parameters "method=" tells JUnit params which method to use above

this object is actually a method that returns an object with the parameters required by the test method below

JUnitParams uses the test() method parameters as the order of the test variables in the object above

JUnitParams (cont.)

- A few things you need to make JUnitParamsRunner work

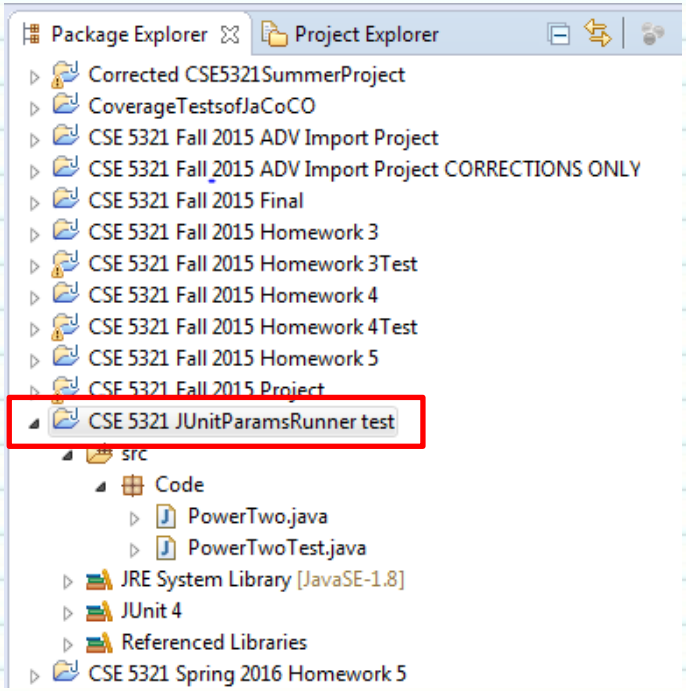
```
package Code;

import static org.junit.Assert.*;
import org.junit.Test;
import junitparams.JUnitParamsRunner;
import org.junit.runner.RunWith;
import static junitparams.JUnitParamsRunner.$;
import junitparams.Parameters;

@RunWith(JUnitParamsRunner.class)
public class PowerTwoTest {
```

Additional imports required

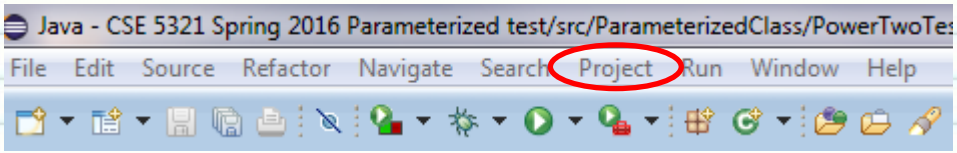
- You will also need to add Junit Params jar to your build path - this is on blackboard along with the other files.
- The next slides show you how to add this to your Java build path

JUnitParams (cont.)

- In Eclipse - right click on the Project (from Package Explorer)

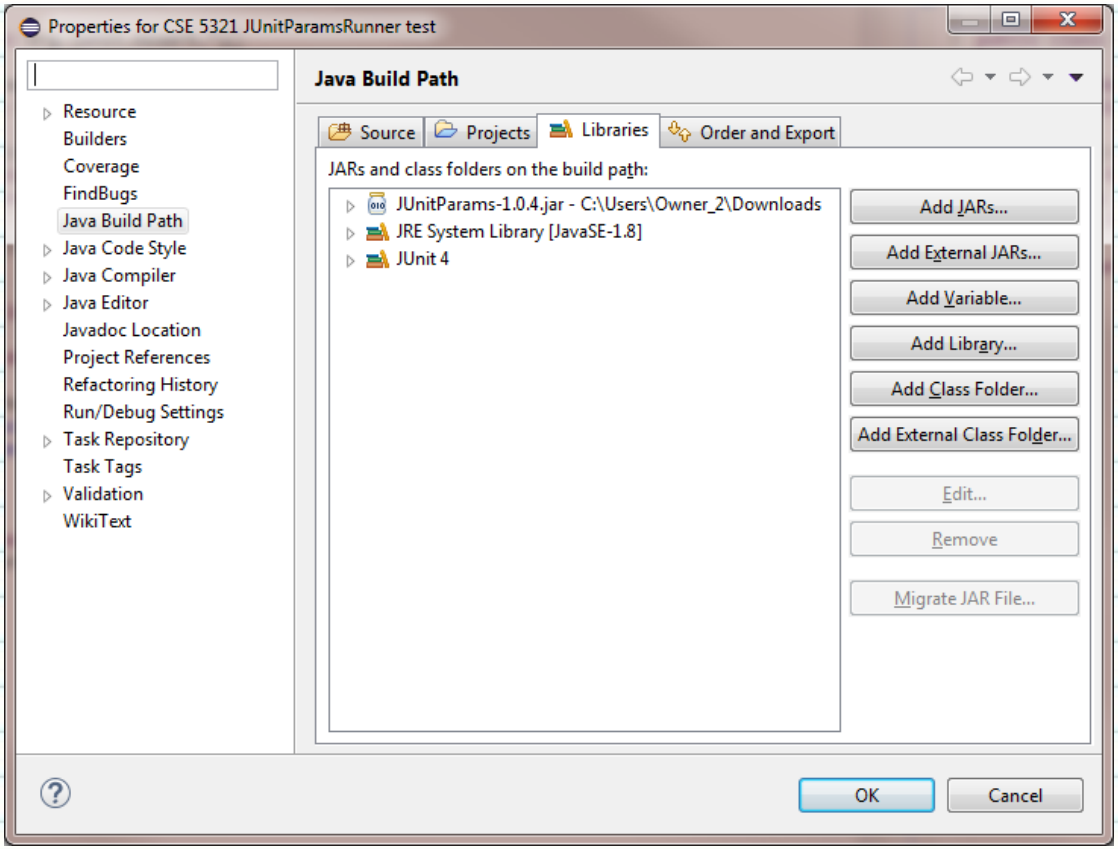


1. Select the Project tab in Eclipse
2. Select Properties under the Project tab
3. You will see the pop-up on the next slide



JUnitParams (cont.)

- Java Build Path



1. Under the Libraries tab select "Add external JARs..."
2. Find the JAR file - you need to download this first
3. Ok, etc

Student Exercise

```
...
@RunWith(JUnitParamsRunner.class)
public class microwaveClassTestJUnitParamsStudent {

    private microwaveClass micOven;

    @SuppressWarnings("unused")
    private static final Object[] parametersFormicrowaveClassTest () {
        return $(
//              Parameters are: (1,2,3,4,5)
//              1=cooking, 2=doorOpen, 3=stopButton, 4=timer, 5=cookState_res,6=stop_res,7=timer_res
//              Test case 1
                $(true,  false,  true, 1,        true,        false,        0),
//              Test case 2
                $(true,  false,  false,1,        false,       false,        0)
        );
    }

    @Before
    public void setUp () {
        micOven = new microwaveClass(false, false, 0);
    }

    @Test
    @Parameters(method="parametersFormicrowaveClassTest")
    public void test(boolean cooking, boolean doorOpen, boolean stopButton, int timer,
                                        boolean cookState_res, boolean stop_res, int timer_res) {

        micOven.setTimer(timer);
        micOven.operateMicrowave(cooking, doorOpen, stopButton);
        assertEquals(cookState_res,micOven.isCookState());
        assertEquals(stop_res,micOven.isStop());
        assertEquals(timer_res,micOven.getTimer());
    }}
```

this file is microwaveClassTestJUnitParamsStudent.java

Student Exercise

- Use the file microwaveClassTestJUnitParamsStudent.java
  1. Complete all five test cases from the data developed in the previous student exercise (see overhead)
  2. Add the data to the object as the first two rows are done. Make sure to place commas in the correct places when you add rows
  3. Execute the tests

Student Exercise (cont.)

```
@RunWith(JUnitParamsRunner.class)
public class microwaveClassTestJUnitParams {

    private microwaveClass micOven;

    @SuppressWarnings("unused")
    private static final Object[] parametersFormicrowaveClassTest () {
        return $(
            //Parameters are: (1,2,3,4,5)
            //1=cooking, 2=doorOpen, 3=stopButton, 4=timer, 5=cookState_res,6=stop_res,7=timer_res
            //Test case 1
            $(true,  false,true, 1,true,false,0),
            //Test case 2
            $(true,  false,false,1,false,false,0),
            //Test case 3
            $(true,  false,false,0,false,true,0),
            //Test case 3a
            $(true,  true,false,0,true,true,0),
            //Test case 3b
            $(false,  true,false,0,false,true,0)
        );
    }

@Before
public void setUp () {
micOven = new microwaveClass(false, false, 0);
}
```

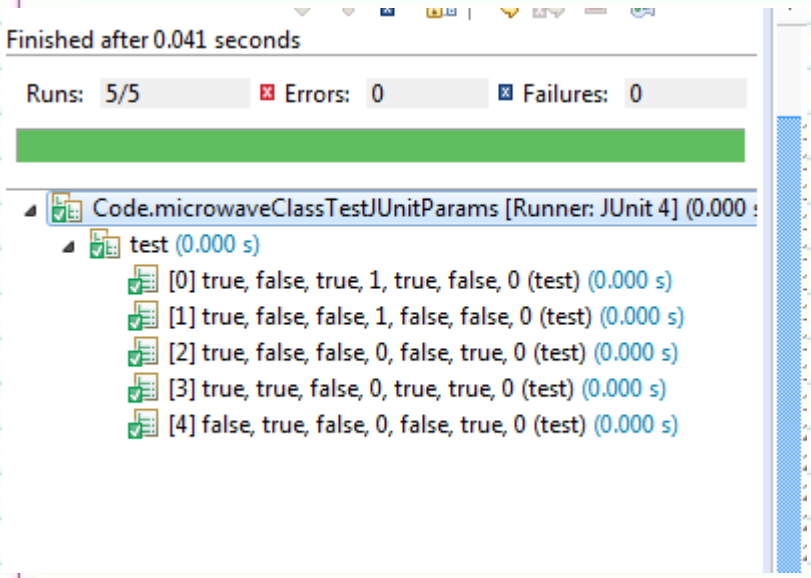
Complete answer shown here

Notice that the collection uses $ - this is an abbreviation for new Object[]

Student Exercise (cont.)

```
    @Test
    @Parameters(method="parametersFormicrowaveClassTest")
    public void test(boolean cooking, boolean doorOpen, boolean stopButton, int timer,
                                                    boolean cookState_res, boolean stop_res, int timer_res) {

        micOven.setTimer(timer);
        micOven.operateMicrowave(cooking, doorOpen, stopButton);
        assertEquals(cookState_res,micOven.isCookState());
        assertEquals(stop_res,micOven.isStop());
        assertEquals(timer_res,micOven.getTimer());
    }

}
```

Student Exercise (cont.)



When you execute the 5 test cases and expand the JUnit run bar not only will you see each instance of the Collection row and its result but you will see each value in the test data for that test

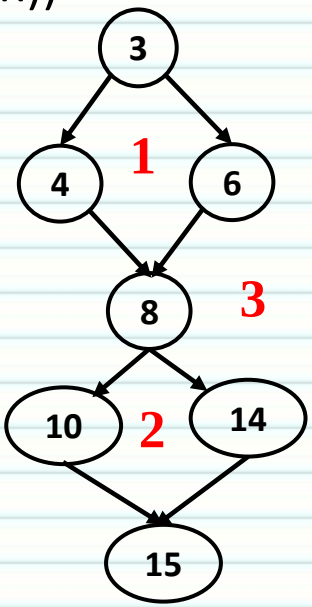
Here we see 5 tests passed. The collector index is handy to let you know which test passed or failed.

The tests will run to completion regardless of getting failures - all tests will run.

Reading Test Data From CSV files

Achieving Condition Coverage (Example 4)

```
1   public void operateMicrowave (boolean cooking, boolean doorOpen,
                                                boolean stopButton) {
2
3   if (cooking && (doorOpen || stopButton))
4       cookState=true;
5   else
6       cookState=false;
7
8   if (timer>0)
9       {
10      stop=false;
11      timer--;
12      }
13  else
14      stop=true;
15  }
```

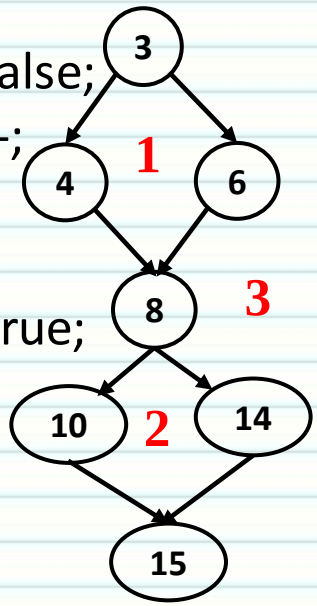


cookState, stop, and timer are private class variables

Example - Microwave Test

```
1   public void operateMicrowave (boolean cooking, boolean doorOpen,
                                  boolean stopButton) {
2
3   if (cooking && (doorOpen || stopButton))
4       cookState=true;
5   else
6       cookState=false;
7
8   if (timer>0)
9       {
10      stop=false;
11      timer--;
12      }
13  else
14      stop=true;
15  }
```

Test Case	Inputs				Expected Outputs			Basis Path tested
	cooking	doorOpen	stopButton	timer	cookState	stop	timer	
1	TRUE	FALSE	TRUE	1	TRUE	FALSE	0	3-4-8-10-15
2	TRUE	FALSE	FALSE	1	FALSE	FALSE	0	3-6-8-10-15
3	TRUE	FALSE	FALSE	0	FALSE	TRUE	0	3-6-8-14-15
4	TRUE	TRUE	FALSE	0	TRUE	TRUE	0	-
5	FALSE	TRUE	FALSE	0	FALSE	TRUE	0	-



1) MCDC solution for a(b + c)=TFT,TFF,TTF,FTF
2) We have combined MCDC with basis path as shown above
3) The order for basis path forces us to choose statement 3 first true then false - so we match the MCDC up in order
4) Test case 3a abd 3b test the remaining 2 MCDC terms
5) All MCDC tests toggle only the COI between tests
6) Each test changes only one input at a time

Example - Microwave Test (cont.)

- Steps to convert a JUnitParamsRunner test into a FileParameters test
- Modify JUnit test file
  1. Add "int testcaseNumber" as first parameter in the test method
  2. Add "String bpNumber" as the last parameter in the test method
  3. Change @Parameters to @FileParameters as the following
     a. @FileParameters("src/Code/microwave.csv")
     b. Make sure the path is correct - here the file is in the same directory as the JUnit test
     c. If you paste in a path make sure to change the "\" to "/"
- In Excel
  1. Save as CSV
  2. Delete table header rows
  3. Remove thousands separators (commas) from numerical data
  4. Remove currency symbols from currency data
  5. Save again

Results

Package Explorer | Project Explorer | JUnit

Finished after 0.058 seconds

Runs: 5/5 Errors: 0 Failures: 0

- Code.microwaveClassTest [Runner: JUnit 4] (0.007 s)
  - test (0.007 s)
    - [0] 1,TRUE,FALSE,TRUE,1,TRUE,FALSE,0,3-4-8-10-15 (test) (0.002 s)
    - [1] 2,TRUE,FALSE,FALSE,1,FALSE,FALSE,0,3-6-8-10-15 (test) (0.000 s)
    - [2] 3,TRUE,FALSE,FALSE,0,FALSE,TRUE,0,3-6-8-14-15 (test) (0.002 s)
    - [3] 4,TRUE,TRUE,FALSE,0,TRUE,TRUE,0,- (test) (0.000 s)
    - [4] 5,FALSE,TRUE,FALSE,0,FALSE,TRUE,0,- (test) (0.003 s)

Failure Trace

microwaveClass.java

M12a ▸ src ▸ Code ▸ microwaveClass ▸ operateMicrowave(boolean, boolean, boolean) : void

```
1  package Code;
2
3
4  public class microwaveClass {
5
6      boolean cookState,stop;
7      int timer;
8
9      public microwaveClass (boolean cookState, boolean stop, int timer)
10         this.cookState=cookState;
11         this.stop=stop;
12         this.timer=timer;
13     }
14
15     public void operateMicrowave (boolean cooking, boolean doorOpen, bo
16
17     if (cooking && (doorOpen || stopButton))
18         cookState=true;
19     else
20         cookState=false;
21
22     if (timer>0)
23         {
24         stop=false;
25         timer--;
26         }
27     else
28         stop=true;
29     }
30
31     public boolean isCookState() {
32         return cookState;
33     }
34
35     public boolean isStop() {
36         return stop;
37     }
38
39     public int getTimer() {
```

Demo