COMP311: Unit Testing with JUnit

Due
This assignment must be performed in class on Lab class.

Marking
It is marked out of 10 and worth up to 3% of your final mark. Your mark is based on attendance and the JAR file that you export from Eclipse and upload to the dropbox on eCentennial. The JAR cannot be marked unless it includes source.

Submitting your solution
1. When you are finished testing, export a JAR file containing all packages in your Java project, following instructions at the end of this document.
2. Rename your jar so that the filename contains your name, as in ***MaryWorth*****JUnit.jar**.
3. Upload your JAR to the dropbox for this assignment.

Introduction
JUnit is possibly the most popular unit-testing tool for Java code. In this exercise you follow a script that walks you through testing a sample slice of an application. Some Java coding knowledge and experience in an Eclipse-based IDE is assumed.

While completing this assignment, you learn how JUnit works. At the end you should be able to use JUnit without a script and answer questions on JUnit in an exam. You learn:
- How to code a run a test class
- How to combine test classes into test suite
- How to use the JUnit perspective in Eclipse

Notes
Pages 2 to 4 describe the sample code to be tested. The code is identical to the air ticketing sample you used in the Debugging and CheckStyle assignments.
Be sure to use the startup version code. If you made any changes to the code, download the file again from eCentennial.

Recall that the purpose of testing is to identify defects, not to fix them. For some of the tests, a failure is the correct result: a test failure indicates a bug.

Instructions for testing with JUnit start on page 5.

Specifications for the air ticketing prototype

The requirement is for a prototype that implements a system for issuing tickets for seats on an air flight. The prototype does not include the user interface. For testing purposes do not use the driver and UI stub classes **Manifest** or **UserPrompter**. Instead, test cases can call methods on the classes listed on the next page directly, passing constant values instead of user input as arguments.

Simplifying factors for this prototype:

- Preconditions state that the passenger has already selected a specific flight on a specific day.
- Invoicing and processing payment are handled by a separate subsystem. The application calculates the price of a ticket, but does no billing.
- The prototype airplane has very limited seating of only 14 seats. Row 1 is business class and has one seat on either side of the aisle: 1A and 1B. Rows 2, 3 and 4 are economy class and have two seats on either side of the aisle: 2A, 2B, 2C,2D, 3A ..3D, 4A ..4D.
- The standard price for a business class seat is $750 and for an economy seat is $500.
- The seating plan – number of rows in each class, seats in each row, and seat price -- is specified by the instances of the enumeration `SeatingClass`.
- No persistent data (databases) are used.
- All system output goes to the console (`System.out.stdout`).

Other requirements

- **FrequentFlyers** are passengers who are members of the airline's frequent flyer club. They are identified by a membership number and have privileges that are outside the scope of the prototype.
- Employees of the airline cannot also be frequent flyers. They are **StaffPassengers** and are identified by employee number.
- Members of airline staff get a 50% deduction on the price of seats **in Economy class only**. A **StaffPassenger** who wants to travel in business class is treated like a regular passenger and pays the full fare.
- Only one ticket can be sold for each seat. Overbooking is not allowed.

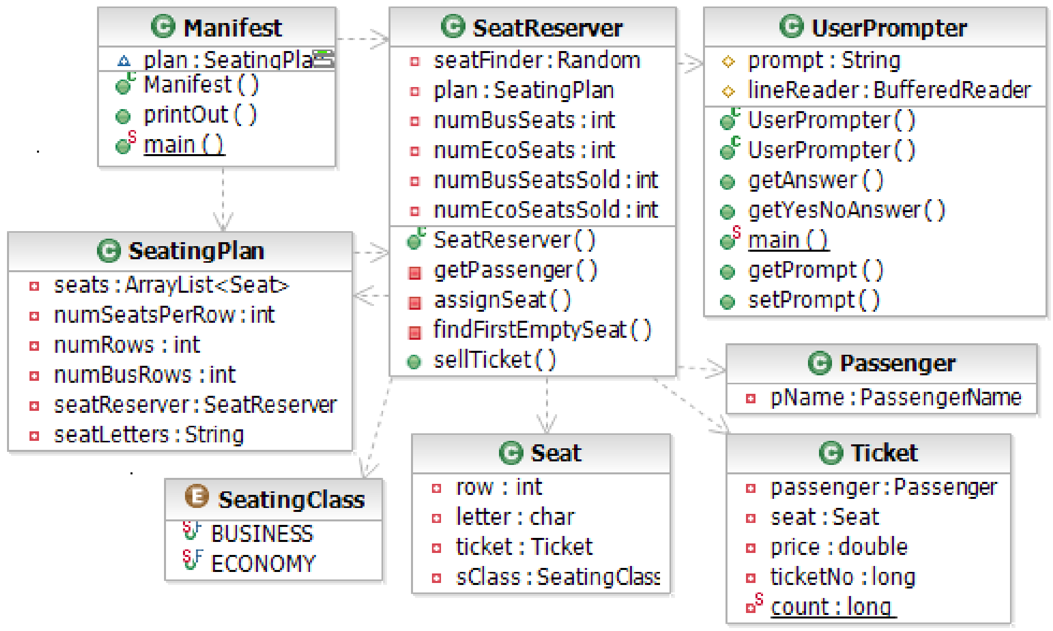
Design Documentation

The next page has two UML class diagrams showing the main types and associations between them. These diagrams are for your reference only. Just to fit the page, one diagram is split into two and, as a result, some classes appear in both diagrams.

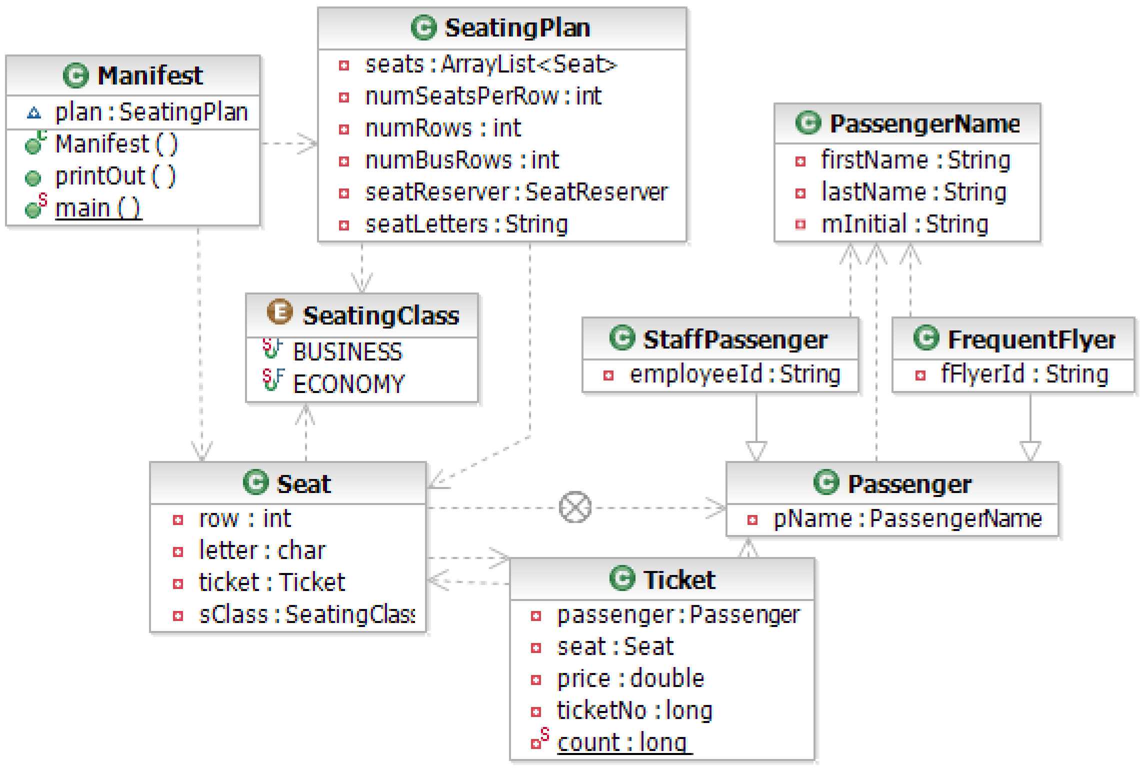
Dependencies (method calls and has-a relationships) are shown by dashed-line arrows.

COMP311: Unit Testing with JUnit

UML class diagram 1 of air ticketing example:



UML class diagram 2 of air ticketing example:



Not shown in these diagrams is the interface **Discountable**, implemented by the class **StaffPassenger**.

Summary of the Types involved in the prototype

Name	**Type -- description**	**Fields and/or comments**
Seat	Class – instance represents a single seat in the airplane	Fields are row, letter, seating class, and either a `Ticket` if sold or `null` if not sold.
SeatingPlan	Class – instance represents the seating plan of a plane	Fields include an `ArrayList` of `Seat` objects, a `SeatReserver` object, and variables such as the total number of rows in the seating plan.
SeatingClass	Enumeration – Two instances are BUSINESS and ECONOMY	Treat this type as a set of constants that specify the number of rows, seats per row, and standard price of a seat in each class. *There is no need to test it for this exercise.*
SeatReserver	Class that contains business logic to find available seats, assign seats to passengers, and sell tickets	Other fields are counters such as total number of seats and number of seats sold.
Ticket	Class – instance represents a ticket	Fields are a `Passenger` object, a `Seat` object, ticket price, and ticket number. (A static field counts tickets sold.)
Passenger	Class – instance represents a passenger	Field is a `PassengerName` object.
PassengerName	Class – instance is a name	Fields are `Strings`: firstName, lastName, mInitial. Middle initial must be supplied.
FrequentFlyer	Subclass of `Passenger` – instance is a passenger who is a frequent flyer	Cannot also be a `StaffPassenger` Field is a `String`: fFlyerID.
Staff Passenger	Subclass of `Passenger` – instance is a passenger who is also an airline employee	Cannot also be a `FrequentFlyer` Field is a `String` employeeId.
Discountable	Interface implemented by `StaffPassenger`	Used by `SeatReserver` calculate discount on ticket price.

You cannot extensively test all of these classes in this exercise. The instructions walk you through some sample tests. While running these tests you should realize that many more test cases are required to thoroughly test even this very small prototype.

Instructions

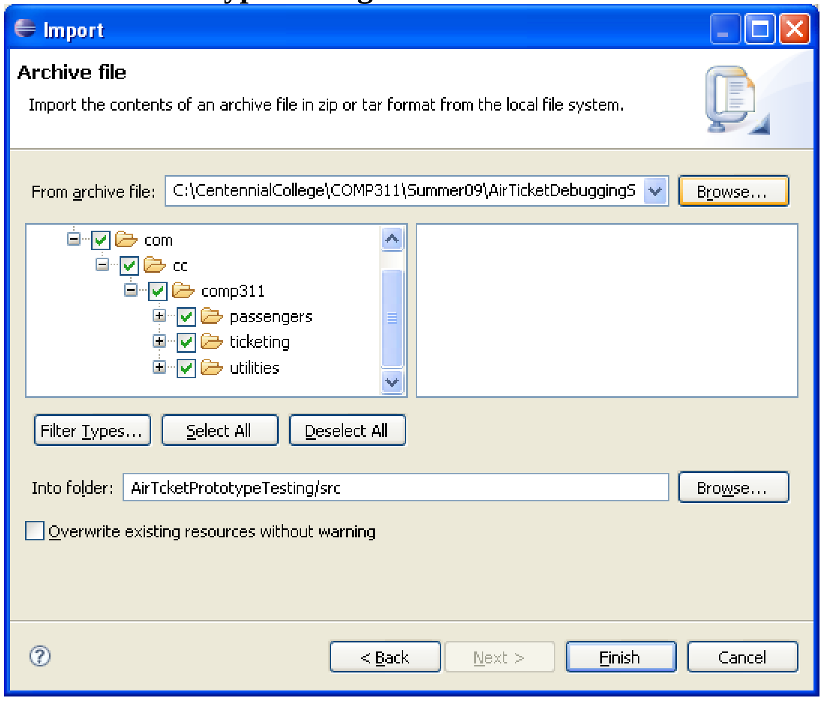
Wherever you see the name *Mary Worth* in these instructions, substitute your own name. Do this so that your name appears in the solution code.

1. Download file **AirTicketDebuggingStart.jar** from eCentennial. You can find it under **Unit 2: Assignments** as the document with title *Air Ticket Example: assignment startup code*.

2. Start **Eclipse**
   a. You can use a new workspace or the same workspace used in previous assignments.
   b. Make sure work in the **Java** perspective.

3. Create a new Java project and call it **AirTicketPrototypeTesting**.

4. Import the JAR file **AirTicketDebuggingStart.jar** into the new project:
   a. Right-click the project name in the Project Explorer view and select **Import → General → Archive File**.
   b. **Browse** to the downloaded JAR file and select **Open**.
   c. Make sure the entire contents of the jar are selected, that the **Into folder** is **AirTicketPrototypeTesting/src** and click **Finish**.



Import the contents of an archive file in zip or tar format from the local file system.

From archive file: C:\CentennialCollege\COMP311\Summer09\AirTicketDebuggingS

com
cc
comp311
passengers
ticketing
utilities

Into folder: AirTcketPrototypeTesting/src

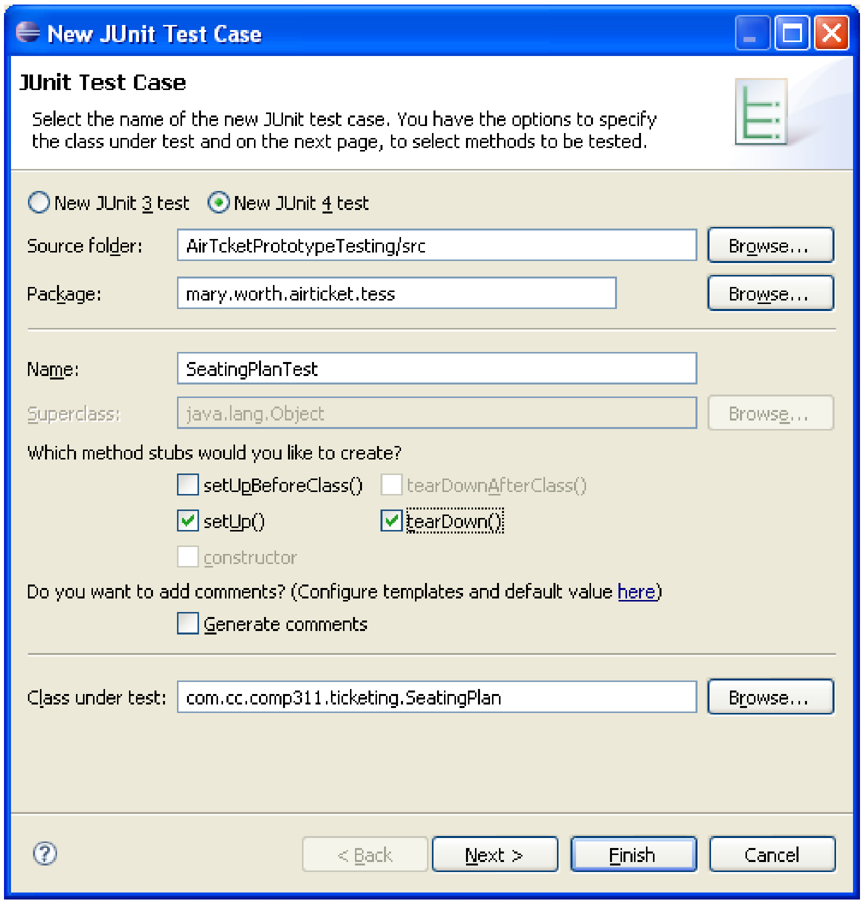
Overwrite existing resources without warning

d. In the Package Explorer, verify that the `src` folder contains three packages:
   i. `com.cc.airline.passengers`
   ii. `com.cc.airline.ticketing`
   iii. `com.cc.airline.utilities`

5. Create a JUnit test case to verify that the constructor of the `SeatingPlan` class generates the required seating plan, by following the steps below.

For this assignment, put all test classes into one package and include your name in the package name. For example, **mary.worth.airticket.tests**.

a. Right-click on the **AirTicketPrototypeTesting** project, select **New →JUnit Test Case**.
   i. Be sure to select the **New JUnit 4 test** radio button.
   ii. Enter the package name ***your.name*.airticket.tests**.
   iii. Enter the class name **SeatingPlanTest**.
   iv. Check the boxes to create stubs for **setUp()** and **tearDown().**
   v. Specify that the class under test is **SeatingPlanTest**.
   vi. Click **Next**.



JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder: AirTcketPrototypeTesting/src

Package: mary.worth.airticket.tess

Name: SeatingPlanTest

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☑ setUp() ☑ tearDown()

☐ constructor

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: com.cc.comp311.ticketing.SeatingPlan

vii. When you see a list of test methods, select to generate tasks for the constructor **SeatingPlan()** only. Click **Finish**.

viii. A dialog asks if you want to add Junit 4 to the project build path. Click **OK**. Because JUnit is built into Eclipse this is all the setup required to build and run test cases.

b. The tool generates a skeleton test class with annotated methods for you. Complete the methods:

i. Insert code into the `setUp()` method to write the message “Starting test of the SeatingPlan default constructor” to the console.

ii. Insert code into the `tearDown()` method to write the message “Test of the SeatingPlan default constructor complete” to the console.

iii. Modify the to the `testSeatingPlan()` method to create an instance of `SeatingPlan` and verify that the new object is constructed as expected. Consider how to express the test using methods of the `org.junit.Assert` class. For example, the `SeatingPlan` instance must not be null. Its `getSeats()` method should return an `Array List` of size 14, and its `getSeatReserver()` method should return a non-null object reference.
Just for this first test, solution code is provided here.

```
package mary.worth.airticket.tests;
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import com.cc.airline.ticketing.SeatingPlan;

public class SeatingPlanTest {
      @Before
      public void setUp() throws Exception {
            System.out.println(
                  "Starting test of the SeatingPlan default constructor");
      }
      @After
      public void tearDown() throws Exception {
            System.out.println(
                  "Test of the SeatingPlan default constructor complete");
      }
      @Test
      public void testSeatingPlan() {
            SeatingPlan sp = new SeatingPlan();
            Assert.assertNotNull(sp);
            Assert.assertEquals(sp.getSeats().size(), 14);
            Assert.assertNotNull(sp.getSeatReserver());

      }
}
```

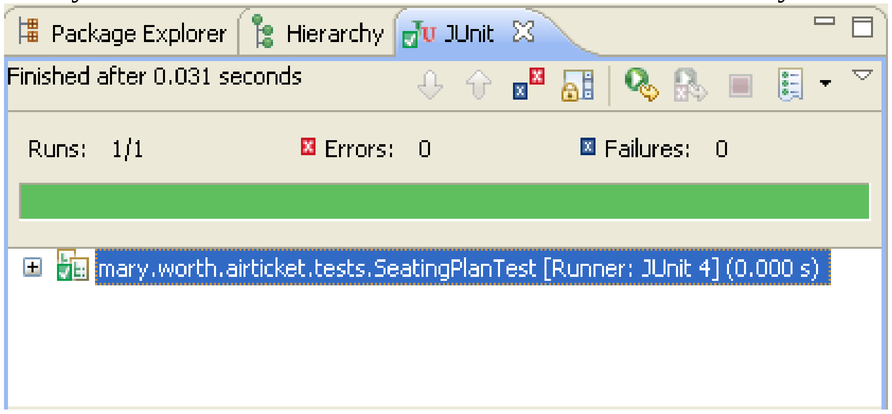
c. Run the test and verify that it is successful.

i. In the Package Explorer, right-click the **SeatingPlanTest** class and select **RunAs → JUnit test**.

ii. The Console view opens and displays the following messages:

```
Starting test of the SeatingPlan default constructor
Seating plan has 14 seats.
Test of the SeatingPlan default constructor complete
```

iii. Verify that the JUnit view shows that the test ran successfully.



Finished after 0.031 seconds

Runs: 1/1 Errors: 0 Failures: 0

mary.worth.airticket.tests.SeatingPlanTest [Runner: JUnit 4] (0.000 s)

6. Create a second test class called `TicketTest` to test that a `Ticket` object is constructed correctly when provided reasonable input.

a. Write a test method that calls the following constructor with valid arguments: `Ticket(Passenger p, Seat s, double price)`
Before you create the Ticket object:

i. Use the wizard to generate the `TicketTest` class

ii. Add a test method to the class.

iii. In the test method, create a `Passenger` object. The default constructor of the `Passenger` class generates a valid name "T. B. A.", so *either* of the following statements instantiate a `Passenger`:

```
Passenger passenger = new Passenger();
Passenger passenger = new Passenger(
      new PassengerName("Mary", "I", "Worth");
```

iv. Create a `Seat` object.
The `Seat` class has no explicit constructor so you must use the implicit default constructor and then call `seat.setRow(int i)` and `seat.setLetter(char c)` to specify a specific seat.

v. Create the ticket object, providing the passenger object, seat object and the price as a number as the three arguments of the constructor.

```
Ticket ticket = new Ticket(passenger, seat, price);
```

b. For a condition that verifies that a ticket is issued, use the fact that tickets are numbered, starting at 1,000,000. So lines in the test method could look like:

```
Assert.assertTrue(ticket.getTicketNo() > 1000000 );
```

c. Write a second test in the same class that generates a ticket when called with arguments that are not valid. In real life, you should create several tests to isolate different combinations of valid and invalid input values, but for purposes of this exercise combine several tests, as in:

```
@Test( expected=Exception.class)
public void testTicketBad() {
      Ticket ticket = new Ticket(null, new Seat(), -100);
      Assert.assertTrue(ticket.getTicketNo() > 1000000 );
}
```

d. Run the test class as a JUnit test and take note of the results.

Hint: The test fails because the second test method does not throw an exception where clearly one should be thrown. This defect should be reported. It would helpful for the test report suggest more tests that give better coverage invalid input.

7. You want a test class to test the logic of selling a ticket. Review the `SeatReserver.sellTicket()` method and note that it contains a mixture of user interface code and business logic. Your test manager decides that it is appropriate to postpone testing class `SeatReserver` until the actual user interface is ready. However, risk analysis indicates that ensuring that the price of a ticket is calculated correctly is very important. Create a new test class:

   a. Create a new JUnit test case as before:
      i. Name the new class **SellTicketTest**.
      ii. Ask for method stubs for **setUpBeforeClass()** and **tearDownAfterClass()**.
      iii. Leave the box **Class under test** empty and click **Finish**.

   b. When the skeleton class is generated, add bodies to the `setUpBeforeClass()` and `tearDownAfterClass()` methods to print messages to the console as you did for the previous `setUp()` and `tearDown()` methods.

If necessary, search on Google or refer to JUnit documentation to recall the difference between JUnit annotations @Before and @BeforeClass and between @After and @AfterClass.

Create a `sellTicket1` test method as shown below. The logic of the code is similar to the `SeatReserver.sellTicket()` method but this test replaces user input with specific values:

```
@Test
public void sellTicket1() {
      SeatingClass sClass = SeatingClass.ECONOMY;
      Passenger passenger = new Passenger(
            new PassengerName("Mary", "I", "Worth"));
      Seat seat = new Seat();
      seat.setRow(3); seat.setLetter('D');
      double price = sClass.getPrice();
      Ticket ticket = new Ticket(passenger, seat, price);
      System.out.println("Ticket issued: " + ticket);
      Assert.assertEquals(ticket.getPrice(), 500.0, 0.005 );
}
```

c. Add a similar method `sellTicket2` to test the case of a selling a ticket to an airline employee who expects to pay half price, $250, for an economy class ticket. For example, instantiate the passenger with:

```
PassengerName pName = new
      PassengerName("Mary", "I", "Worth");
StaffPassenger passenger = new StaffPassenger(pName, "EMP123");
```

d. Add a method `sellTicket3` to see what happens when a StaffPassenger tries to buy a business class ticket and expects to pay full fare. Specify business class using an instance of Seating class created with the line:

```
 SeatingClass sClass = SeatingClass.BUSINESS;
```

e. Run all three test cases in the `SellTicketTest` class. Does a `StaffPassenger` get the employee discount for economy class and is he or she correctly charged the full price for business class? Analyze the test results and test code to try to deduce what class and method has a problem calculating ticket prices for `StaffPassengers`.

8. Create a test suite that combines all the tests and run the test suite.

   a. Close the JUnit view and all open editor windows.

   b. Create a new Java class called **AirTicketTestSuite** in the **your.name.airticket.tests** package.

Note: you can always create a JUnit test class or suite as though it was an ordinary Java class. For test cases, using the JUnit tools in Eclipse save you time by generating method stubs, import statements, and annotation for you. The tools also detect whether the JUnit jars are on the classpath (called project build path in Eclipse) and can add them to the classpath for you.

c. Before the class declaration, add the annotations:

```
@RunWith(Suite.class)
@SuiteClasses({SeatingPlanTest.class,
   TicketTest.class,
               SellTicketTest.class})
```

d. To correct errors use the **Source → Organize Imports** feature of the editor.

e. Run the test suite class as a JUnit test. Review the results of this test suite and the seven test cases it contains.

At this point the tests required for this assignment are complete. Your test suite as a whole should fail because at least one test failed. In other words, the testing exercise was productive because it found defects.

9. Export your package of test classes to a JAR: As you follow this step, refer to the image on the next page.

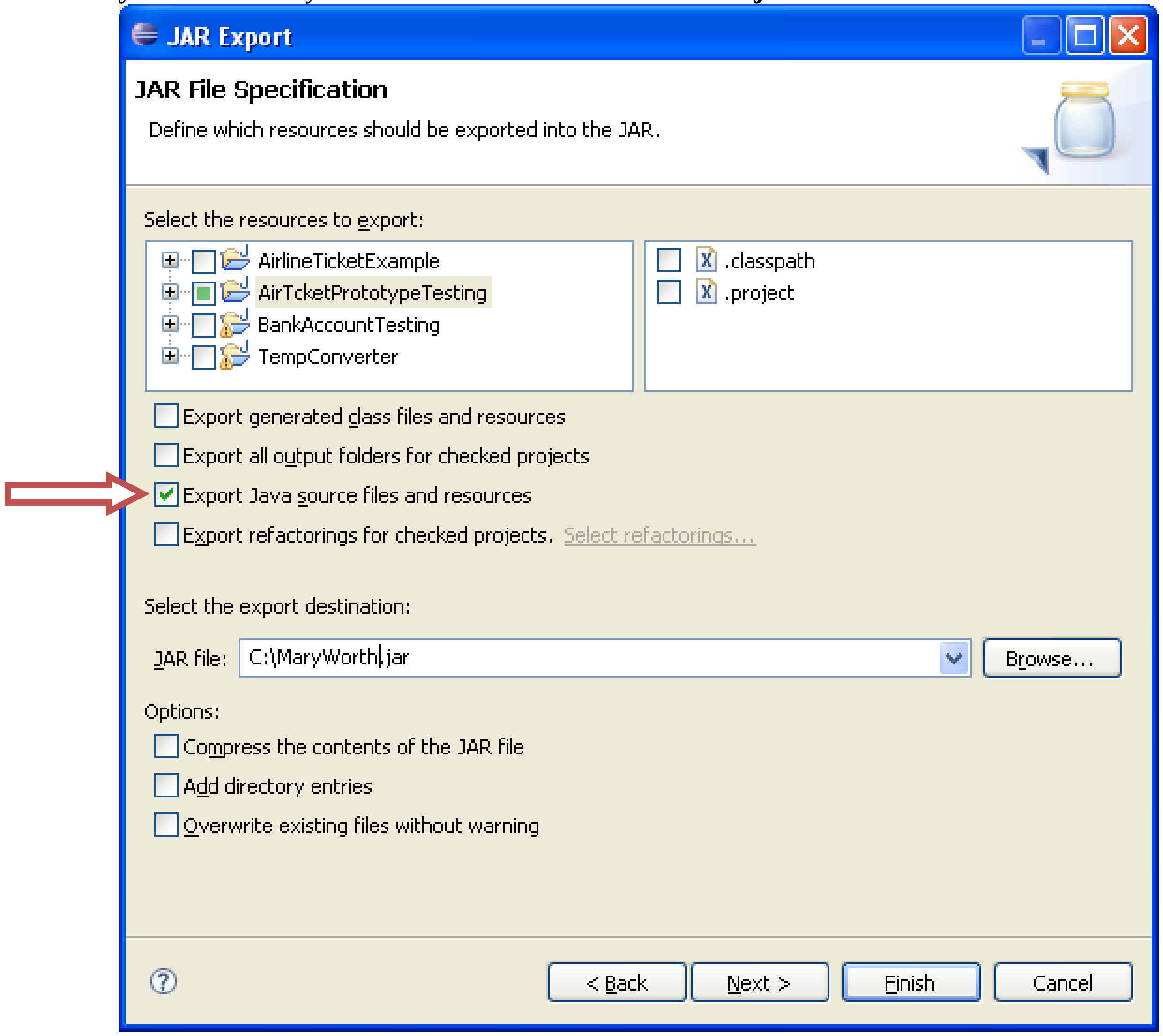
a. In the package explorer, right-click on package **your.name.airticket.tests** and select **Export**.

b. Under **Java**, select to export a **JAR** file. Click **Next**.

c. In the Select the resources to export area, expand the project **AirTicketPrototypeTesting** and make sure only your test package is selected (marked with a green square).

⇨ d. Make sure the box **Export Java source files and resources** is checked. No other boxes below the list of exportable resources should be checked.

e. In the JAR file entry area, enter the full path to where you want to put the jar on your local file system. Call the file ***YourName*JUnit.jar**. Click **Finish**.


JAR Export

JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

- [ ] AirlineTicketExample
- [■] AirTcketPrototypeTesting
- [ ] BankAccountTesting
- [ ] TempConverter

- [ ] .classpath
- [ ] .project

- [ ] Export generated class files and resources
- [ ] Export all output folders for checked projects
- [✓] Export Java source files and resources
- [ ] Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: C:\MaryWorth.jar

Options:

- [ ] Compress the contents of the JAR file
- [ ] Add directory entries
- [ ] Overwrite existing files without warning

10. **Optional:** to make sure that you have exported source, create another Java project and import your jar into to, following the same steps as importing the startup code. Make sure you can see the source code for your classes.

11. Hand in your test code:
    a. Demonstrate the test suite running to the instructor. Be prepared to explain the resulting results view.
    b. Upload the exported JAR to the dropbox on eCentennial.

End of exercise