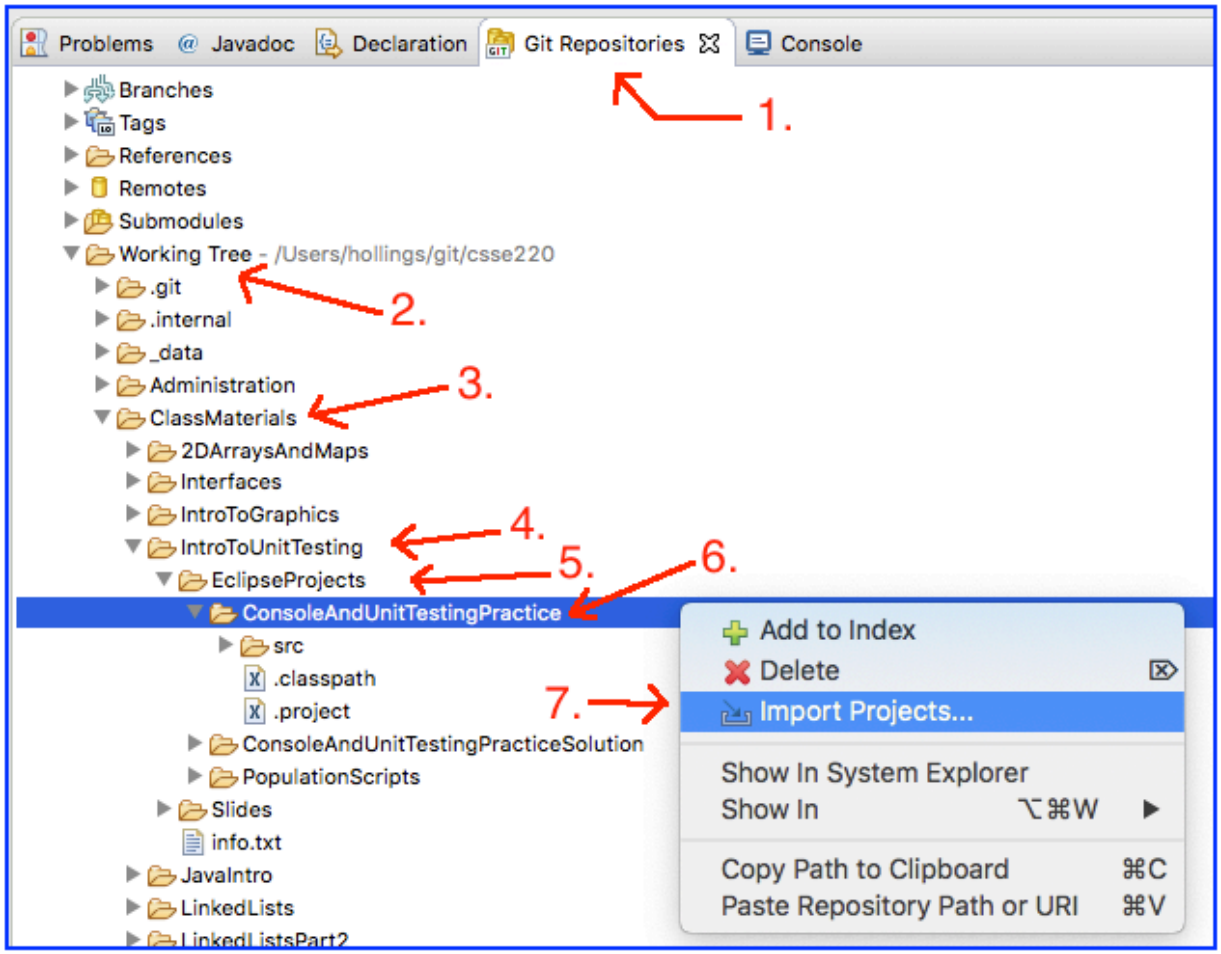
CSSE 220

Console Input

Import out ConsoleAndUnitTestingPractice.

Import Today's Eclipse Project



Outline

- Console Input
- Unit Testing
- Work time

Reading keyboard input from the console

CONSOLE INPUT WITH JAVA.UTIL.SCANNER

Console input with Scanner

- Creating a Scanner object:

```
import java.util.Scanner;
Scanner inputScanner = new Scanner(System.in);
```

- Scanner defines methods to read from keyboard
  - inputScanner.nextInt();
  - inputScanner.nextDouble();
  - inputScanner.nextLine();
  - inputScanner.next();

Q1

Console input with Scanner

- Exercise: Look at UnitTesting/src/ConsoleWorker.java
  Add missing methods to read from console
- Missing Setter Methods:
  - setName()
  - setBirthMonth()
  - setBirthDay()
  - setBirthYear()
  - setAnualSalary()

Console input with Scanner

- Example:

```
public void setName(){
   // Provide a prompt to user with System.out.print
   System.out.print("What is your name? ");
   this.name = this.inputScanner.nextLine().trim();
} // setName
```

Use:

inputScanner.nextInt(); // to get an integer

inputScanner.nextDouble(); // to get a double

Scanner Gotcha

```
Double nonNegDouble;
if (scanner.nextDouble() > 0) {
    nonNegDouble = scanner.nextDouble();
}
```

Scanner Gotcha

```
Double nonNegDouble;
if (scanner.nextDouble() > 0) {
    nonNegDouble = scanner.nextDouble();
}
```

- Functions normally don't change the objects that they work on
- But the stream of input characters is coming in from outside world, so scanner behaves differently than operations working internal data objects, e.g., String operations on string objects

Work on In-Class Activity

- In Eclipse
- Import *ConsoleAndUnitTestingPractice*
- Show what to do

Unit Testing

- Idea: Test “small pieces” of larger program
- A “small piece” of code is referred to as a “unit”
- For each unit, you must determine:
  - Does the unit produce the expected values, i.e., does its result match what you expected it to give
  - This means you have to know ahead of time what it the expected value

Q2

Unit Testing

How to test a software system in “small pieces”?
Two approaches:

1. Driver program – Make a *main* method that calls all the operations under test
   - *handleTestOp1*() calls *Op1* to test it
   - *handleTestOp2*() calls *Op2* to test it
   - Etc.
2. JUnit Testing
   - Create a Tester JUnit class
   - In JUnit class create an operation that is a client of the operation you want to test

Q2

How to do Unit Testing

- Create a “testing harness”
  1. Driver program – Make a *main* method that calls all the methods with various test cases
     - Often test cases are created in an ad hoc method
     - For example, a user typing in commands to the driver program, i.e., just testing what ever comes to mind
  2. JUnit Testing
     - By creating a Tester JUnit class
     - This approach can lead to more systematic and methodical testing
     - Can reuse these JUnit test classes later to re-test

Q2

Why Unit Testing?

- There are several goals of unit testing:
  - The #1 goal of testing is: to reveal defects
  - Make sure your code works (as specified!)
  - Keep it working
  - Confirm understanding of the specification
  - Confirm pieces of code in isolation
  - Provide Documentation

Q3

Unit Tests – the *printout method*

1. Construct one or more objects of the class that is being tested
2. Invoke one of the class's methods
3. Print out results from invoked method
4. Also print the expected results
5. Do 3 and 4 match?

Why JUnit?

- Provides a Framework
  - Framework: Collection of classes to be used by another program
- Provides easy-to-read output in Eclipse
- The *printout method* requires you to visually analyze all lines
  - What if it scrolls off the page?
  - What if it's only 1 character different?

Unit Testing

- Use “assert” to make sure results match
- Switch to Eclipse
- Select BadFac.java, right click, select “New | JUnit Test Case”
- Let’s look at BadFrac.java and BadFracTest.java
  - Let’s make some unit tests and figure out why this project has been yielding some strange results

What are good unit tests?

- Unit tests should:
  1. First test: The most common cases of legal input
  2. Then working on: Edge cases of legal input – e.g., minimum, maximum, switching from positive to negative, etc.
  3. All specific/special cases (e.g., when 0 or null the behavior is different than for any other value)
  4. If you find and fix a defect w/o the benefit of a test case, then create a unit test that would have exposed this defect so that later if that defect is accidentally reintroduced, your test case will find it
  5. Any overly complex code that 1-4 above don't cover

Q4

Unit Testing On Upcoming Computer Part of Exam

- Worth X points
- You will be given an operation and asked to write a multiple unit tests for it
- Each test case should test a different aspect of the operation
- It's recommend that you add a comment saying, e.g., "edge case test"
- The operation may have a defect, or it may not. Evaluation of the quality of your test cases will not include if your test case actually exposes a defect
- Your test case's expected value must be correct based on what the provided operation's detailed behavioral description

Back to In-Class Activity

Now come up with good test cases for:

`isReduced` operation in BadFrac.java

Use these assert statements:

```
assertTrue(Boolean producing expression);
assertFalse(Boolean producing expression);
assertEquals(expression1, expression2);
```

You might have to import:

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

- About 5 to 10 minutes
- Then give me some of your test cases, and I'll write them on the board