

Unit Testing

Lecture Question

- This is Lecture Objective 4 from the Pale Blue Dot project -
 - In the `pbd.PaleBlueDot` object, write a method named "bigCities" which:
 - Takes three Strings and an Int as parameters representing:
 - The name of a file containing city data. ex. "data/cities.csv"
 - A two character country code. You may assume that the input is in all lowercase letters
 - A region code in the same format as they appear in the cities file
 - The minimum population of a city
 - Returns a List of all the city names in the given country/region with at least the minimum population.
 - In `tests.LectureObjective4`, complete a test suite to test the functionality of this method
 - Note: All the Strings will be in the same format as they appear in the cities file. You don't not have to do anything with upper/lower case for this objective

Last Time

Lecture Question

Function: In a package named "lecture" create an object named "FirstObject" with a method named "computeShippingCost" that takes a Double representing the weight of a package as a parameter and returns a Double representing the shipping cost of the package

The shipping cost is (\$) $5 + 0.25$ for each pound over 30

Unit Testing: In a package named "tests" create a class/file named "UnitTesting" as a test suite that tests the computeShippingCost method

rounding_weight

```
def computeShippingCost(weight: Double): Double = {  
  if (weight < 3.0) {  
    5.0  
  } else {  
    5.0 + (Math.round(weight) - 30.0) * 0.25  
  }  
}
```

always_over_thirty

```
def computeShippingCost(weight: Double): Double = {  
  5.0 + (weight - 30.0) * 0.25  
}
```

boundary_overweight

```
def computeShippingCost(weight: Double): Double = {  
  if (weight < 33.0) {  
    5.0  
  } else {  
    5.0 + (weight - 33.0) * 0.25  
  }  
}
```

light_employee_discount

```
def computeShippingCost(weight: Double): Double = {  
  if (weight < 30.0) {  
    4.0  
  } else {  
    5.0 + (weight - 30.0) * 0.25  
  }  
}
```

always_under_thirty

```
def computeShippingCost(weight: Double): Double = {  
  5.0  
}
```

Live Demo

Solving this lecture question

Recap of Lecture Question

- Comparing Doubles
 - Never use ==
- Using a Map to store test cases can keep your testing code organized
- How much testing is enough?
 - At a minimum, complete the AutoLab portion of the assignment
 - More testing will often be required to ensure correctness
- Your testing should be thorough enough that you are confident that your code is correct if it passes all your testing
 - Convince yourself that your testing is complete

Example

- **Task:** Write a method that takes a String and returns a List of all the anagrams of the input
- **Testing:** Write a test suite to test this functionality
- Live Walkthrough

Recap of Anagrams

- Comparing Lists
 - Can use ==
 - Elements and order must match
 - Can sort the Lists if the order is not important
- It will not always be easy to know that a method is correct
 - My method should be very difficult for you to read at this point in your career
- How will you be confident that my code is correct on all inputs?
 - Thorough unit testing!
- How will you be confident that code you write is correct on all inputs?
 - Thorough unit testing!

Lecture Question

Method: In a package named "lecture" create an object named "Algebra" with a method named "factor" that takes an Int as a parameter and returns the prime factorization of that parameter as a List of Ints.

The following apply to this method:

- If the input is negative, 0, or 1, return an empty list
- Do not include 1 in the output for any inputs
- The order of the factors in the output List is undefined

Example: `lecture.Algebra.factor(12)` can return `List(2,2,3)` -or- `List(2,3,2)` -or- `List(3,2,2)`

Unit Testing: In a package named "tests" create a class/file named "TestFactoring" as a test suite that thoroughly tests the factor method