Unit Testing

Testing

- How do you know if your code is correct?
- Submit to AutoLab?
  - Does not exist outside of class
  - Does not exist for your project

Recall

```scala
package example

object Conditional {

  def computeSize(input: Double): String = {
    val large: Double = 60.0
    val medium: Double = 30.0
    if (input >= large) {
      "large"
    } else if (input >= medium) {
      "medium"
    } else {
      "small"
    }
  }

}
```

This method should return "large" if the input is greater than or equal to 60.0, "medium" if the input is less than 60.0, but greater than or equal to 30.0, and "small" if the input is less than 30.0

- **How do we test this function to verify that it's correct?**

Recall

```scala
package example

object Conditional {

  def computeSize(input: Double): String = {
    val large: Double = 60.0
    val medium: Double = 30.0
    if (input >= large) {
      "large"
    } else if (input >= medium) {
      "medium"
    } else {
      "small"
    }
  }

  def main(args: Array[String]): Unit = {
    println(computeSize(70.0))
    println(computeSize(50.0))
    println(computeSize(10.0))
  }

}
```

- **Call the method from main**
- **Print the results**
- **Manually verify**

What About Large Projects?

- There may be 100's of files and 1000's of methods
- Any change in a function might break any code that calls that function
- Will you manually verify all that code for each change?

- **Unit Testing**
  - Automate testing
  - Provide structure to testing

Unit Testing

- Run a series of tests on your code
- If the code is correct, all tests should pass
- If the code is incorrect, at least one test should fail
- A set of tests should test every possible error that could occur

Scala Unit Testing

```
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Doubles are checked for size in each category") {
    val largeDouble: Double = 70.0
    val mediumDouble: Double = 50.0
    val smallDouble: Double = 10.0

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDouble) == "medium", mediumDouble)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

Use Maven to download scalatest (see pom.xml on the example repo and project code)

Click Maven in the IntelliJ sidebar to interact with pom.xml

Scala Unit Testing

```
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Doubles are checked for size in each category") {
    val largeDouble: Double = 70.0
    val mediumDouble: Double = 50.0
    val smallDouble: Double = 10.0

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDouble) == "medium", mediumDouble)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

Import everything from the org.scalatest package

_ is a Scala wildcard

Scala Unit Testing

```scala
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Doubles are checked for size in each category") {
    val largeDouble: Double = 70.0
    val mediumDouble: Double = 50.0
    val smallDouble: Double = 10.0

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDouble) == "medium", mediumDouble)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

Create a new class of type FunSuite (Function Suite)

*More detail on this syntax throughout LO2. This is inheritance

Scala Unit Testing

```
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Doubles are checked for size in each category") {
    val largeDouble: Double = 70.0
    val mediumDouble: Double = 50.0
    val smallDouble: Double = 10.0

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDouble) == "medium", mediumDouble)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

Create a new test that will be executed when this file is ran

No main method

IntelliJ will run a test runner with a main method that calls your code

Scala Unit Testing

```scala
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Doubles are checked for size in each category") {
    val largeDouble: Double = 70.0
    val mediumDouble: Double = 50.0
    val smallDouble: Double = 10.0

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDouble) == "medium", mediumDouble)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

Call assert to test values

First argument is a boolean that must be true for the test to pass
-Should return false if the code is not correct
Second argument is optional. Is printed if the test fails

Testing Demo

Scala Unit Testing

```
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Doubles are checked for size in each category") {
    val largeDouble: Double = 70.0
    val mediumDouble: Double = 50.0
    val smallDouble: Double = 10.0

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDouble) == "medium", mediumDouble)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

This class tests if the inputs 70.0, 50.0, and 10.0 return "large", "medium", and "small" respectively

Is this enough testing?

A Correct Solution

```
package example

object Conditional {

  def computeSize(input: Double): String = {
    val large: Double = 60.0
    val medium: Double = 30.0
    if (input >= large) {
      "large"
    } else if (input >= medium) {
      "medium"
    } else {
      "small"
    }
  }

}
```

Incorrect Solution
-Passes the tests-

```scala
package example

object Conditional {

  def computeSize(input: Double): String = {
    val large: Double = 65.0
    val medium: Double = 20.0
    if (input >= large) {
      "large"
    } else if (input >= medium) {
      "medium"
    } else {
      "small"
    }
  }

}
```

Scala Unit Testing

```
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Size boundaries are checked"){
    val largeDouble: Double = 60.0
    val mediumDoubleUpperBound: Double = 59.99
    val mediumDoubleLowerBound: Double = 30.0
    val smallDouble: Double = 29.99

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDoubleUpperBound) == "medium", mediumDoubleUpperBound)
    assert(Conditional.computeSize(mediumDoubleLowerBound) == "medium", mediumDoubleLowerBound)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

Check the boundaries for more accurate testing

Is this enough testing?

Scala Unit Testing

```
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Size boundaries are checked"){
    val largeDouble: Double = 60.0
    val mediumDoubleUpperBound: Double = 59.99
    val mediumDoubleLowerBound: Double = 30.0
    val smallDouble: Double = 29.99

    assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    assert(Conditional.computeSize(mediumDoubleUpperBound) == "medium", mediumDoubleUpperBound)
    assert(Conditional.computeSize(mediumDoubleLowerBound) == "medium", mediumDoubleLowerBound)
    assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
  }

}
```

Check the boundaries for more accurate testing

Is this enough testing?

We could reasonable stop here.. but we could do more thorough testing

Scala Unit Testing

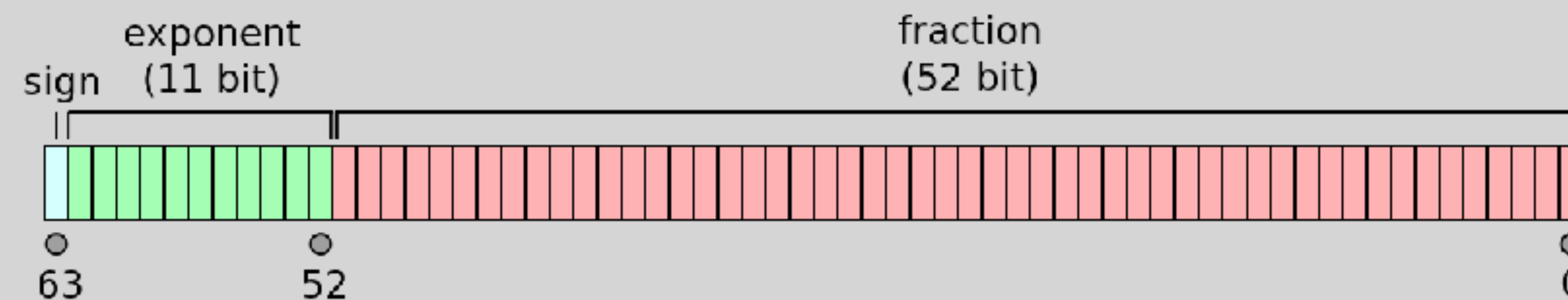
```
package tests

import org.scalatest._
import example.Conditional

class TestComputeSize extends FunSuite {

  test("Use many test cases for each category"){
    // notice largeDoubles must be declared with var we change its value
    var largeDoubles: List[Double] = List(60.0, 60.01, 70.0, 90.0, 1000.0)
    val mediumDoubles: List[Double] = List(59.9, 30.0, 30.01, 40.0, 50.0)
    val smallDoubles: List[Double] = List(29.99, 20.0, 10.0, 0.0, -100.0, -10000.0)

    largeDoubles = largeDoubles :+ 10000.0 // Example of adding an element to a List

    for(largeDouble <- largeDoubles){
      assert(Conditional.computeSize(largeDouble) == "large", largeDouble)
    }
    for(mediumDouble <- mediumDoubles){
      assert(Conditional.computeSize(mediumDouble) == "medium", mediumDouble)
    }
    for(smallDouble <- smallDoubles){
      assert(Conditional.computeSize(smallDouble) == "small", smallDouble)
    }

  }

}
```

Use data structures to run many test cases

Testing Doubles

Testing Doubles

- Number with a whole number and a decimal portion
- 64 bit representation
- Values are truncated to fit in 64 bits
  - Loss of precision!

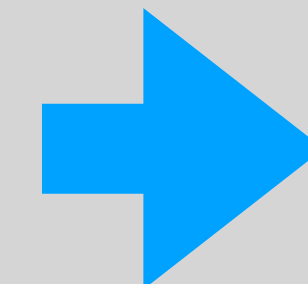


https://en.wikipedia.org/wiki/Double-precision_floating-point_format

Testing Doubles

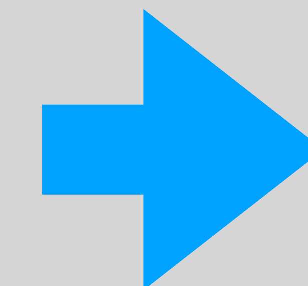
- Checking for equality with Doubles
- Allow a small amount of tolerance when comparing two doubles
- Math.abs(x - y) < small_value
    - As long as x and y are within a small value of each other this will be true

```
val b: Double = 0.1
val c: Double = b * 3
val expected: Double = 0.3
assert(c == expected)
```



test fails

```
val epsilon: Double = 0.001
val b: Double = 0.1
val c: Double = b * 3
val expected: Double = 0.3
assert(Math.abs(c - 0.3) < epsilon)
```

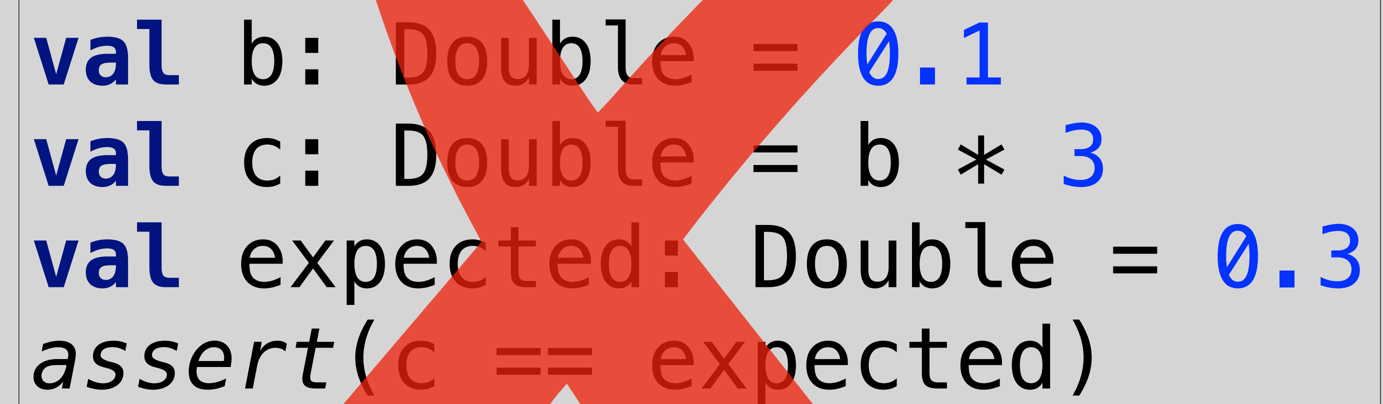


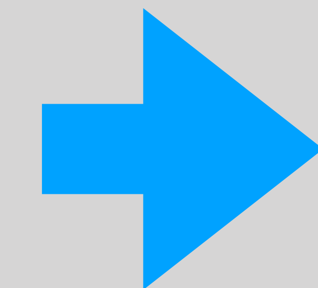
test passes

Testing Doubles

- **Never use == when testing Doubles!!!!**

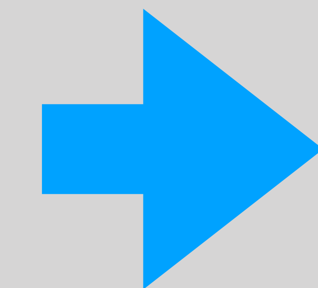
```
val b: Double = 0.1
val c: Double = b * 3
val expected: Double = 0.3
assert(c == expected)
```





test fails

```
val epsilon: Double = 0.001
val b: Double = 0.1
val c: Double = b * 3
val expected: Double = 0.3
assert(Math.abs(c - 0.3) < epsilon)
```



test passes

Testing in CSE116

Unit Testing Tasks

- Most/all tasks will require thorough testing
- When these tasks are graded, your test suite is ran:
  - Against your solution
  - Against a correct solution stored on the server
  - Against a variety of incorrect solutions stored on the server
- Your test suite should pass both your solution and the correct solution
- Your test suite should fail all the incorrect solutions

Incorrect Solutions for Task 1

always_returns_empty_string

```
def getCountryCode(countriesFilename: String, countryName: String): String = {
  ""
}
```

requires_matching_case

```
def getCountryCode(countriesFilename: String, countryName: String): String = {
  val countriesFile: BufferedSource = Source.fromFile(countriesFilename)
  var answer = ""
  for (line <- countriesFile.getLines()) {
    val splits: Array[String] = line.split("#")
    val name = splits(0)
    val code = splits(1)
    if (name == countryName) {
      answer = code.toLowerCase()
    }
  }
  answer
}
```

returns_uppercase_country_code

```
def getCountryCode(countriesFilename: String, countryName: String): String = {
  val countriesFile: BufferedSource = Source.fromFile(countriesFilename)
  var answer = ""
  for (line <- countriesFile.getLines()) {
    val splits: Array[String] = line.split("#")
    val name = splits(0)
    val code = splits(1)
    if (name.toLowerCase() == countryName.toLowerCase()) {
      answer = code
    }
  }
  answer
}
```

Your Test Must Fail Every Incorrect Solution

requires_matching_case

```
def getCountryCode(countriesFilename: String, countryName: String): String = {
  val countriesFile: BufferedSource = Source.fromFile(countriesFilename)
  var answer = ""
  for (line <- countriesFile.getLines()) {
    val splits: Array[String] = line.split("#")
    val name = splits(0)
    val code = splits(1)
    if (name == countryName) {
      answer = code.toLowerCase()
    }
  }
  answer
}
```

- This test will pass requires_matching_case
- All test cases match the upper/lower-case in the countries file
- Add tougher test cases to thoroughly test the code

```
test("1 - Country names that have proper capitalization") {

  val testCases: Map[String, String] = Map(
    "Uganda" -> "ug",
    "Japan" -> "jp",
    "South Africa" -> "za",
    "Peru" -> "pe",
    "Belgium" -> "be",
    "Albania" -> "al"
  )

  for ((input, expectedOutput) <- testCases) {
    val computedOutput: String = PaleBlueDot.getCountryCode(countriesFile, input)
    assert(computedOutput == expectedOutput, input + " -> " + computedOutput)
  }

}
```

Thorough Testing

- Thorough testing will test every feature of the method
  - If you coded it, test it!
  - Go through the Task description and write lots of tests for every piece of behavior
- Include common cases
  - We would expect users to have proper capitalization for country names
- Include uncommon cases
  - Some users may capitalize random letters
- Include edge cases
  - Inputs that are unlike any other input
  - Ex. "", "Not a real country"
- Write lots of tests!
  - When in doubt, write more tests!

```
test("1 - Country names that have proper capitalization") {
  val testCases: Map[String, String] = Map(
    "Uganda" -> "ug",
    "Japan" -> "jp",
    "South Africa" -> "za",
    "Peru" -> "pe",
    "Belgium" -> "be",
    "Albania" -> "al"
  )
  for ((input, expectedOutput) <- testCases) {
    val computedOutput: String = PaleBlueDot.getCountryCode(countriesFile, input)
    assert(computedOutput == expectedOutput, input + " -> " + computedOutput)
  }
}

test("2 - Country names with random upper/lower-case") {
  val testCases: Map[String, String] = Map(
    "hEaRd IsLaNd AnD mCdOnAlD iSlAnDs" -> "hm",
    "UGANDA" -> "ug",
    "south africa" -> "za",
    "jAPAn" -> "jp",
    "PEru" -> "pe",
    "chilE" -> "cl"
  )
  for ((input, expectedOutput) <- testCases) {
    val computedOutput: String = PaleBlueDot.getCountryCode(countriesFile, input)
    assert(computedOutput == expectedOutput, input + " -> " + computedOutput)
  }
}

test("3 - Test cases that are not countries in the data file") {
  val testCases: Map[String, String] = Map(
    "" -> "",
    "Not a real country" -> "",
    "j a p a n" -> ""
  )
  for ((input, expectedOutput) <- testCases) {
    val computedOutput: String = PaleBlueDot.getCountryCode(countriesFile, input)
    assert(computedOutput == expectedOutput, input + " -> " + computedOutput)
  }
}
```

Tips for Testing Task 2

- averagePopulation returns a Double
  - Do not use == in your testing!!
  - Allow some tolerance in the returned values
- A common mistake is using integer division
  - Include tests that will fail code that uses integer division
- Think of other potential mistakes that could be made
  - Write tests for all of them