

# 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

```
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"  
    }  
  }  
}
```

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

# Recall

```
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 100's of files and 1000's of functions
- 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
  - Provides structure

# 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 course website)

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

```
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 details on this syntax next week

# 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

FunSuite controls execution instead of main

# 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)
```

```
  }
```

```
}
```

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

# 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?

# 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

```
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

# 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

- \* Later lecture questions won't be available until lecture
- \* This question will be open until midnight