Tests as Documentation

- Lasting, runnable and reliable documentation on the capabilities of the classes you write.
  - Can replace a lot of comments
  - Tests cannot completely replace comments, but you often do not have to write a looooong Javadoc comments.
- Write single-purpose tests.
  - Have each test case (test method) focus on a distinctive behavior of a tested class
  - Do not test multiple/many behaviors in a single test case
    - e.g., divide5by4, multiply3By4
      - Rather than testCalculator, testCalculation

- Give a specific and meaningful name to each test case.
  - e.g., divide5by4, multiply3By4, divide5By0
    - Rather than testDivide (or testDivision), testMultiply (or testMultiplication)
    - Do not even name it like ATest, BasicTest or ErrorTest.
  - Not only suggesting what ***context*** to be tested, suggest what ***happens*** as well by invoking some ***behavior***.
    - *doingSomethingGeneratesSomeResult*
    - divide5By0 v.s. divisionBy0GeneratesIllegalArgumentException

- Suggest what ***happens*** by invoking some ***behavior*** under a certain ***context***.
  - *doingSomethingGeneratesSomeResult*
    - divisionBy0GeneratesIllegalArgumentException
  - *someResultOccursUnderSomeCondition*
    - illegalArgumentExceptionOccursUnderDivisionBy0
  - *givenSomePreconditionWhenDoingSomethingThenSomeResultOccurs*
    - givenTwoNumbersWhenDivisionBy0ThenIllegalArgumentExceptionOccurs
      - divide(5,0)
    - givenTwoStringsWhenDivisionBy0ThenIllegalArgumentExceptionOccurs
      - divide(“5”, “0”)
    - “Given-When-Then” style
    - “*givenSomePrecondition” can be dropped → doingSomethingGeneratesSomeResult*

Many Many Naming Conventions Exist

- 7 popular conventions
  - https://dzone.com/articles/7-popular-unit-test-naming
- No single “correct” way exists to name test methods.
  - Personal taste, project history…

- Like to include the name of a tested method?
  - divide5By0GeneratesIllegalArgumentException
    - v.s. divisionBy0GeneratesIllegalArgumentException
  - isAdultFalseIfAgeLessThan18
    - v.s. isNotAnAdultIfAgeLessThan18
  - Like to explicitly state which method is tested?
  - Like to focus on a behavior/feature that a method under test implements, not method name itself?
  - What if it is renamed?
    - Often need to rename test methods manually.
    - Method calls in test code can be automatically refactored.

- Class under test

```
public class Calculator{
 public float multiply(float x,
                       float y){
   return x * y;
 }
 public float divide(float x,
                     float y){
   if(y==0){ throw
    new IllegalArgumentException(
     “division by zero”);}
   return x/y;
 }
]
```

- Test class

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class CalculatorTest{
 @Test
 public void multiply3By4(){
   Calculator cut = new Calculator();
   float expected = 12;
   float actual = cut.multiply(3,4);
   assertThat(actual, is(expected); }

 @Test
 public void divide3By2(){
   Calculator cut = new Calculator();
   float expected = 1.5f;
   float actual = cut.divide(3,2);
   assertThat(actual, is(expected)); }


@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```

- Like to use underscores (_)?
  - givenTwoStringsWhenDivisionBy0ThenIllegalArgumentExceptionOccurs
  - given_TwoStrings_When_DivisionBy0_Then_IllegalArgumentExceptionOccurs
- Like to keep the name of a test method as short as possible?
  - Up to 7 or so words?

- No need to include the prefix “test” in each test method
  - JUnit now encourages you to use @Test.

```
• @Test
  public void divide3By2(){
     ...
  }
• public void testDivide3By2(){
     ...
  }
```

Testing Exceptions to be Thrown

- *Positive* tests
  - Verifying tested code runs without throwing exceptions
- *Negative* tests
  - Testing is not always about ensuring that tested code runs without errors/exceptions.
  - Sometimes need to verify that tested code throws an exception(s) when expected.

Positive Tests

-
  ```
  @Test
  public void readFromTestFile(){
   BufferedWriter writer = new BufferedWriter(
                               new FileWriter(“test.txt”);
   try{
     writer.write(“test data”);
   }
   catch(IOException ex){
     fail();
   }
   finally{
     writer.close();
   }
  }
  ```
- When `write()` throws an `IOException`, this test case fails with `fail()`. Otherwise, the test case passes.
- Clear, logic-wise, but try-catch-finally blocks can clutter a test case.

- Alternative strategy
  - Have a test case *re-throw* an exception
    - rather than *catching* it.

-
  ```
  @Test
  public void readFromTestFile() throws IOException {
   BufferedWriter writer = new BufferedWriter(
                              new FileWriter(“test.txt”);
   writer.write(“test data”);
   writer.close();
  }
  ```
- JUnit’s test runner (i.e. the client code that calls `readFromTestFile()`) will catch an `IOException`.
  - `write()` throws it originally, and `readFromTestFile()` re-throws it.

Negative Tests

- Verify that tested code throws an exception(s) when expected.
  - Understand the conditions that cause tested code to throw each exception and test those conditions in test cases
- 3 Common ways
  - Specify an expected exception(s) with @Test
  - Write a test case with try-catch blocks
  - Specify an expected exception(s) with @Rule

-
  ```
  @Test(expected=illegalArgumentException.class)
  public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
  }
  ```

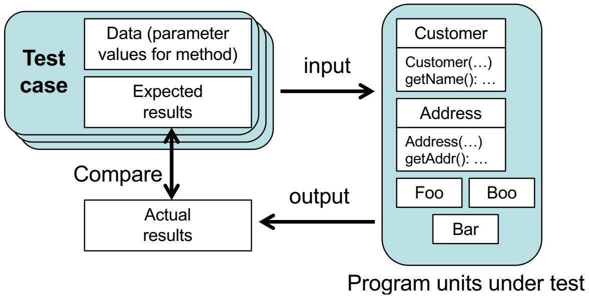
-
  ```
  public void divide5By0(){
   Calculator cut = new Calculator();
   try{
     cut.divide(5,0);
     fail();
   }
   catch(IllegalArgumentException ex){
     assertThat(ex.getMessage(),
                equalTo("division by zero"));
   }
  }
  ```

-
  ```
  @Rule
  public ExpectedException thrown = ExpectedException.none();

  public void divide5By0(){
   thrown.expect(illegalArgumentException.class);
   thrown.expectMessage(“division by zero”);
   Calculator cut = new Calculator();
   cut.divide(5,0);
  }
  ```
- `@Rule`
  - `org.junit.Rule`
  - Used to annotate data fields that reference *rules*.
- `org.junit.rules.ExpectedException`
  - Used to verify that tested code throws a specific exception.
  - `none()`: Returns a rule that expects no exception to be thrown (identical to behavior without this rule).

-
  ```
  @Rule
  public ExpectedException thrown = ExpectedException.none();

  public void divide5By0(){
   thrown.expect(illegalArgumentException.class);
   thrown.expectMessage(“division by zero”);
   Calculator cut = new Calculator();
   cut.divide(5,0);
  }
  ```
- `org.junit.rules.ExpectedException`
  - Used to verify that tested code throws a specific exception.
  - `expect()`: Specify an exception to be thrown.
- The test case passes if the specified exception is thrown at some point when running the rest of the test.
  - It fails otherwise.

Test Fixtures

- Fixture
  - An instance of a class under test
    - A state of the class instance
  - An instance of another class that the class under test depends on
    - A state of that class instance
  - Input data
  - Expected result(s)
  - Set up of a file(s) and other resources
    - e.g., Socket
  - Set up of external systems/frameworks
    - e.g. Database, web server, web app framework, emulator (e.g. Android emulator)



Setting up Fixtures

- Class under test

-
```
public class Calculator{
 public int multiply(int x, int y){
   return x * y;
 }
 public float divide(int x, int y){
   if(y==0) throw
    new IllegalArgumentException(
     “division by zero”);
   return (float)x / (float)y;
 }
]
```

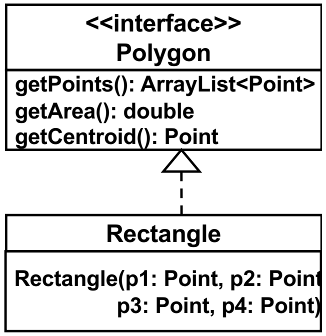
Setting up fixtures

- Test class

-
```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class CalculatorTest{
 @Test
 public void multiply3By4(){
   Calculator cut = new Calculator();
   int expected = 12;
   int actual = cut.multiply(3,4);
   assertThat(actual, is(expected); }

 @Test
 public void divide3By2(){
   Calculator cut = new Calculator();
   float expected = (float)1.5;
   float actual = cut.divide(3,2);
   assertThat(actual, is(expected)); }

@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```

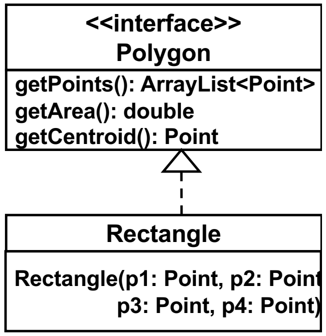
Inline Setup

-
```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class RectangleTest{
 @Test
 public void constructorTest(){
   Rectangle cut = new Rectangle(
                      new Point(0,0),
                      new Point(2,0),
                      new Point(2,2),
                      new Point(0,2) );
   assertThat(cut.getPoints(), contains(...));  }

 @Test
 public void getArea2By2(){
   Rectangle cut = new Rectangle(
                      new Point(0,0),
                      new Point(2,0),
                      new Point(2,2),
                      new Point(0,2) );
   assertThat(cut.getArea(), is(4)); }
}
```



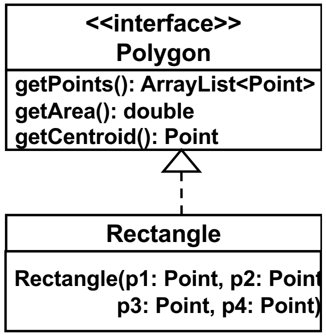
Implicit Setup

-
```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;
import org.junit.Before;
import org.junit.After;

public class RectangleTest{
 private Rectangle cut;

 @Before
 public void setUp(){
   cut = new Rectangle( new Point(0,0),
                        new Point(2,0),
                        new Point(2,2),
                        new Point(0,2) );  }

 @Test
 public void constructorTest(){
   assertThat(cut.getPoints(), contains(...));  }

 @Test
 public void getArea2By2(){
   assertThat(cut.getArea(), is(4)); }

 @After
 public void releaseResources(){...}  }
```



Implicit Setup

-
```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;
import org.junit.Before;

public class RectangleTest{
 private Rectangle cut;

 @Before
 public void setUp(){
   cut = new Rectangle( new Point(0,0),
                        new Point(2,0),
                        new Point(2,2),
                        new Point(0,2) );  }

 @Test
 public void constructorTest(){
   assertThat(cut.getPoints(), contains(...));  }

 @Test
 public void getArea2By2(){
   assertThat(cut.getArea(), is(4)); }
}
```



- Implicit setup makes a test class less redundant.
- Flow of execution
  - @Before setUp()
  - @Test constructorTest()
  - @Before setUp()
  - @Test getArea2By2()

  - The @Before method runs before every test method.
  - JUnit may run the test methods in an order different from their ordering in source code.

FAQs

Why Not Just Use System.out.println() for Testing?

- Your code gets cluttered with println() statements. They will be packaged into the production code.
- You usually scan println() outputs manually every time your code runs to ensure that it behaves as expected.
- It is often hard to understand/remember the intent of each println()-based test.
  - What is tested? What is expected?

Why Not Just Write main() for Testing?

- Your classes get cluttered with test code in main(). The test code will be packaged into the production code.
- If you have many classes to test, you need to run main() in each of them.
- If one method fails, subsequent method calls are not executed.

  -
    ```
    calc.divide(5, 0); // This call fails.
    calc.divide(10, 2);// This is not executed.
    ```
- If you like to display test results in a GUI or record them in a file (e.g. HTML), you will have to write code for that.
- When you join a project, you may see a completely different testing practice with main(). Extra learning time/efforts. Few things are standardized.

Why Not Just Use a Debugger for Testing?

- A debugger can be used for unit testing. However, it is designed for *manual* (or step by step) program execution.
  - i.e. for *manual* debugging and *manual* unit testing.

- JUnit (or any other unit testing frameworks) is designed for *automated* unit testing.

Coverage of Unit Tests

Code Coverage

- How much code is *executed* by test cases.
  - Higher coverage means/implies…
    - You have executed (~ tested) your code more thoroughly.
    - You have lower chances to have bugs in your code.

- Metrics to calculate coverage
  - Line coverage
    - Each line has been executed at least once?
  - Branch coverage
    - Each branch of each control structure (e.g. if, switch, try-catch structures) has been executed at least once?
  - Condition coverage
    - Each combination of true-false conditions has been executed at least once?

Example Coverage Calculation

- Class under test

```
public class Calculator{
 public int multiply(int x, int y){
   return x * y;
 }
}
```

- Test class

```
public class CalculatorTest{
 @Test
 public void multiply3By4(){
   Calculator cut = new Calculator();
   int expected = 12;
   int actual = cut.multiply(3,4);
   assertThat(actual, is(expected); }
}
```

- Line coverage=100% (1/1)
- Branch coverage=100% (1/1)

- Class under test

```
public class Calculator{
 public float divide(int x, int y){
   if(y==0){
    throw
     new IllegalArgumentException(
      “division by zero”);
   }
   return (float)x / (float)y;
 }
}
```

- Test class

```
public class CalculatorTest{
 @Test
 public void divide3By2(){
   Calculator cut = new Calculator();
   float expected = (float)1.5;
   float actual = cut.divide(3,2);
   assertThat(actual, is(expected)); }
}
```

- Line coverage=66% (2/3)
- Branch coverage=50% (1/2)

- Class under test

```
public class Calculator{
 public float divide(int x, int y){
   if(y==0){
    throw
     new IllegalArgumentException(
      “division by zero”);
   }
   return (float)x / (float)y;
 }
}
```

- Test class

```
public class CalculatorTest{

@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```

- Line coverage=66% (2/3)
- Branch coverage=50% (1/2)

- Class under test

```
public class Calculator{
 public float divide(int x, int y){
   if(y==0){
    throw
     new IllegalArgumentException(
      “division by zero”);
   }
   return (float)x / (float)y;
 }
}
```

- Test class

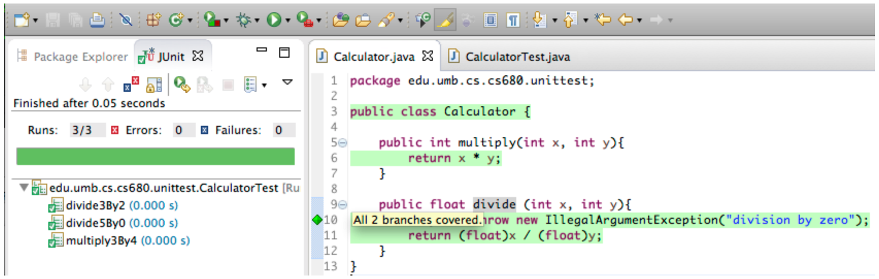
```
public class CalculatorTest{
 @Test
 public void divide3By2(){
   Calculator cut = new Calculator();
   float expected = (float)1.5;
   float actual = cut.divide(3,2);
   assertThat(actual, is(expected)); }


@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```

- Line coverage=100% (3/3)
- Branch coverage=100% (2/2)

EclEmma: A Code Coverage Tool

- A code coverage tool for Eclipse
  - http://eclemma.org/
- Can examine how much code JUnit test cases cover/execute.

- Metrics
  - Line coverage
  - Instruction coverage
  - Branch coverage
  - Method coverage
    - How many methods are executed at least once per class.
    - Useful to find which methods are not tested yet.
  - Type coverage
    - How many classes are executed with 100% method coverage.
    - Useful to find which classes are not fully tested yet.

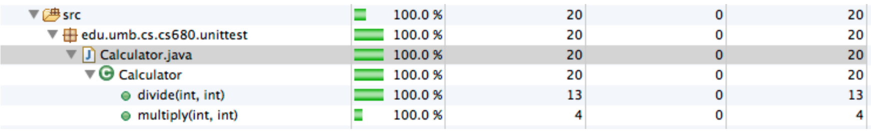


Finished after 0.05 seconds

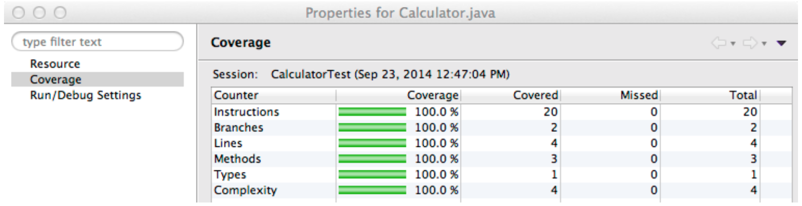
Runs: 3/3 Errors: 0 Failures: 0

- edu.umb.cs.cs680.unittest.CalculatorTest [Ru
  - divide3By2 (0.000 s)
  - divide5By0 (0.000 s)
  - multiply3By4 (0.000 s)

```
package edu.umb.cs.cs680.unittest;

public class Calculator {

    public int multiply(int x, int y){
        return x * y;
    }

    public float divide (int x, int y){
All 2 branches covered.hrow new IllegalArgumentException("division by zero");
        return (float)x / (float)y;
    }
}
```



src	100.0 %	20	0	20
edu.umb.cs.cs680.unittest	100.0 %	20	0	20
Calculator.java	100.0 %	20	0	20
Calculator	100.0 %	20	0	20
divide(int, int)	100.0 %	13	0	13
multiply(int, int)	100.0 %	4	0	4



Session: CalculatorTest (Sep 23, 2014 12:47:04 PM)

Counter	Coverage	Covered	Missed	Total
Instructions	100.0 %	20	0	20
Branches	100.0 %	2	0	2
Lines	100.0 %	4	0	4
Methods	100.0 %	3	0	3
Types	100.0 %	1	0	1
Complexity	100.0 %	4	0	4

- Integration with Ant
  - Use a coverage measurement engine, JaCoCo, which is a part of EclEmma
    - http://www.eclemma.org/jacoco/
  - Jacoco provides ant tasks
    - e.g., <coverage> and <report>
    - http://www.eclemma.org/jacoco/trunk/doc/integrations.html

How to do Code Coverage?

- Rule of thumb: Keep maintaining a reasonably high coverage
  - Need to seek 100% coverage in all metrics? No.
    - ~100% for the method and class coverage metrics
    - 80-90% in the line and branch coverage metrics

    - Depends on the nature of a project, the use of external libraries (e.g., Swing and DBs), etc.
      - c.f. DBUnit

  - You as a programmer is responsible for that.
    - How often?
      - Whenever code is written/revised, ideally.
      - Everyday, once a week, twice a week, etc.
      - When the coverage goes below a threshold.
    - Coverage can decrease very fast.
    - It can be time-consuming to recover it.

Is Coverage Maintenance Effective for Quality Assurance?

- Yes, as far as you have “good” test cases.
  - This test case can yield 100% method coverage for multiply(), but it doesn’t actually test anything.

    -
      ```
      Calculator cut = new Calculator();
      int expected = 12;
      int actual = cut.multiply(3,4);
      //assertThat(actual, is(expected));
      ```

- Note: 100% coverage doesn’t mean bug-free.
  - It simply means that test cases have run your code thoroughly.
  - It’s not a quality indicator.

- Your goal is not reaching the coverage of 100%.

Some Notes

- Utility class
  - Provide a series of utility methods.
    - e.g., java.lang.Math, java.util.Collections
  - Not intended to be instantiated.

```
– final public class SomeUtils{
    private SomeUtils(){}
    public static String aUsefulUtilMethod(int n){
      ...
    } }
```

  - The private constructor is defined to prevent a Java compiler from implicitly inserting a public constructor when no constructors are explicitly defined.
  - No test cases can call it. Coverage decreases.
  - Forget about it.
    - There are some tricks to call it from a test case, but it wouldn't be worth doing that.

- Some exceptions may rarely occur.
  - e.g. IOExcepetion for file I/O operations
  - Test cases may not be able to reproduce all error cases to throw all exceptions. Coverage decreases.
  - Forget about it.
- Branching may be decided at random.

```
– If( Math.random() >= 0.5 ){ do this }else{ do that }
```

  - Both branches may not be covered by running a test case twice.
  - It may be possible to cover all branches by repeating the test case multiple times, but…

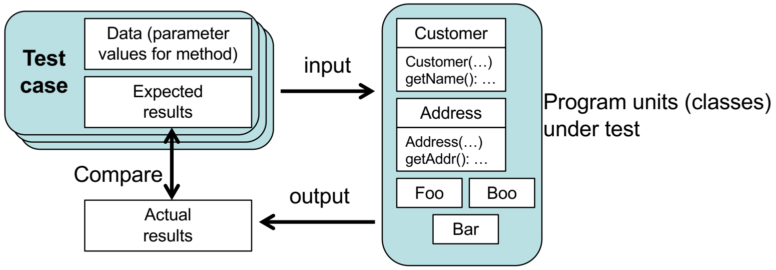
What to Do in Unit Testing?

- 4 tests (test types)
  - CS680 focuses on 3 of them: *functional*, *structural* and *confirmation* tests.

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test				
System test				
Integration test				
Unit test	X (B-box)	?	X (W-box)	X
Code rev&insp.				

Functional Test in Unit Testing

- Ensure that each method of a class successfully performs a set of specific tasks.
  - Each test case confirms that a method produces the expected output when given a known input.
    - Black-box test
  - Well-known techniques: equivalence test, boundary value test

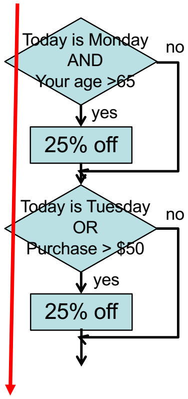


Structural Test in Unit Testing

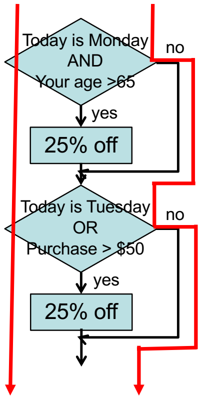
- Verify the structure of each class.
- Revise the structure, if necessary, to improve maintainability, flexibility and extensibility.
  - White-box test
- To-dos
  - Refactoring
  - Use of design pattern
  - Control flow test
  - Data flow test

Control Flow Test

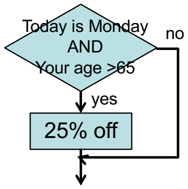
- Verify the flow of program execution
  - White-box test
- Need to decide the coverage metric to be used.
  - Line coverage
  - Branch coverage
  - Condition coverage
- To reach 100% line coverage, use a case where
  - Monday? Y, Age? > 65, Purchase > $50



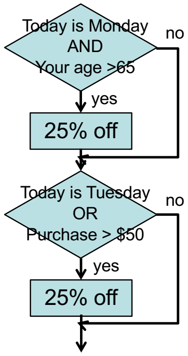
- To reach 100% branch coverage, use 2 cases
  - For example:
    - Monday? Y, Age > 65, Tue? N, Purchase > $50
    - Monday? N, Age > 65, Tue? Y, Purchase > $50



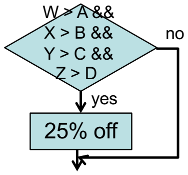
- Condition coverage
  - How many combinations of true-false conditions have been executed at least once?
  - EclEmma does not support it.
    - Need to manually keep track of it.



- Monday?, >65?
  - 4 true-false combinations
    - Y-Y, Y-N, N-Y, N-N
- Need 4 tests to reach 100% condition coverage
  - 4 tests may be in a single test case or 4 different test cases
- 2 tests required for branch coverage
  - Condition coverage requires more tests than branch coverage
    - Condition > branch > line

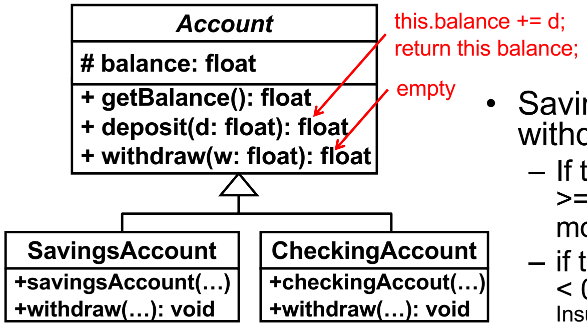


- Monday?, >65?
  - 4 true-false combinations (Y_1-Y_1, Y_1-N_1, N_1-Y_1, N_1-N_1)
- Tuesday?, >$50?
  - 4 true-false combinations (Y_2-Y_2, Y_2-N_2, N_2-Y_2, N_2-N_2)
- Need 8 tests to reach 100% condition coverage
  - Y_1-Y_1, Y_1-N_1, N_1-Y_1, N_1-N_1
  - Y_2-Y_2, Y_2-N_2, N_2-Y_2, N_2-N_2
- Just need 4 tests in fact.
  - Mon, >65, >$50: Y_1-Y_1, N_2-Y_2
  - Mon, >65, <=$50: Y_1-Y_1, N_2-N_2 (redundant)
  - Mon, <=65, >$50: Y_1-N_1, N_2-Y_2 (redundant)
  - Mon, <=65, <=$50:$Y_1$-$N_1$, N_2-N_2
  - Tue, >65, >$50: N_1-Y_1, Y_2-Y_2
  - Tue, >65, <=$50: N_1-Y_1, Y_2-N_2 (redundant)
  - Tue, <=65, >50: N_1-N_1, Y_2-Y_2 (redundant)
  - Tue, <=65, <=$50: N_1-N_1, Y_2-N_2



- 16 (2^4) true-false combinations
  - Y-Y-Y-Y
  - Y-Y-Y-N
  - Y-Y-N-N
  - Y-Y-N-Y
  - …etc.

HW 6: Implement and Test This.



this.balance += d;
return this balance;

empty

- SavingsAccount's withdraw()
  - If this.getBalance() – w >= 0, withdraw the money.
  - if this.getBalance() – w < 0, throw an InsufficientFundsException.
- CheckingAccount's withdraw()
  - If this.getBalance() > w, withdraw the money.
  - If savingsAccount.getBalance() + this.getBalance() >= w, withdraw the money and charge a $50 penalty.
  - If savingsAccount.getBalance() + this.getBalance() < w, throw an InsufficientFundsException.

- Implement the class diagram.
  - It is not complete. You can complete it as you like.
  - Follow the specified rules to implement withdraw().
- Test all methods including constructors with JUnit.
  - You can use any naming convention for test method.
- Measure and report coverage with JaCoCo
  - Reach 100% coverage in all metrics.
  - Reach 100% condition coverage for withdraw().
- Have your Ant script to
  - compile Java code
  - invoke JUnit to run all test cases
  - invoke JaCoCo to generate a coverage report in HTML in the “test” directory.

- Turn in your Ant script, “src” and “test.”
  - Do not send me binary files (Jar and .class files).