Identity and Equality

- `org.hamcrest.CoreMatchers`
  - Contains static methods, each returning a matcher object that performs matching logic.
  - `assertThat(actual, is(sameInstance(expected)))`
    - » Asserts "actual" and "expected" are ***identical*** instance with the same object ID.
      - » `assertThat(new Foo(), is(sameInstance(new Foo())));`
      - » `Foo f = new Foo();`
        `assertThat(f, is(sameInstance(f)));`
  - `assertThat(actual, is(expected))`
    - » A shortcut of `assertThat(actual, is(equalTo(expected)))`
  - `assertThat(actual, is(not(expected)))`
    - » A shortcut of `assertThat(actual, is(not(equalTo(expected))))`
    - » Asserts "actual" is ***logically equal*** to "expected," as determined by calling `Object.equals(java.lang.Object)` on "actual."
      - » `actual.equals(expected);`

- `String str1 = "umb";`
  `String str2 = "umb0".substring(0,2); // "umb0" -> "umb"`
- `assertThat(actual, is(sameInstance(expected))); // FAIL`
  `assertThat(actual, is(equalTo(expected))); // PASS`

- `equalTo()` calls `String.equals()` on "actual."
  - `String.equals()` overrides `Object.equals()` and returns true if two string values match.
  - c.f. Java API doc

- Note: `Object.equals(java.lang.Object)`
  - Implements the most discriminating possible equivalence relation on objects.
    - Returns true if two objects refer to the same instance (x == y has the value true): identity check.
    - c.f. Java API doc
  - Most pre-defined (API-defined) classes override equals() to perform appropriate equality check.

- `Date d1 = new Date(); //java.util.Date`
  `Date d2 = new Date();`
- `assertThat(actual, is(sameInstance(expected))); // FAIL`
  `assertThat(actual, is(equalTo(expected))); // PASS, most likely`

- `equalTo()` calls `Date.equals()` on "actual."
  - `Date.equals()` overrides `Object.equals()` and returns true if two Date objects represent the same timestamp in millisecond.
  - c.f. Java API doc

- Most pre-defined (API-defined) classes override equals() to perform appropriate equality check.

- You need to override equals() in your own (i.e. user-defined) class, if you want to do equality check.

Equality Check for a User-defined Class

- `Person p1 = new Person("John","Doe");`
  `Person p2 = new Person("John","Doe");`
  `Person p3 = new Person("Jane", "Doe");`
- `assertThat(p1, is(sameInstance(p1))); // PASS`
  `assertThat(p1, is(sameInstance(p2))); // FAIL`
  `assertThat(p1, is(equalTo(p2))); // FAIL`

- Person just inherits equals() from Object. The method just do identity check.
  - You need to override equals() in Person if you want equality check.

Person
- firstName: String **- lastName: String**
+ Person(first:String, last:String) **+ getFirstName(): String** **+ getLastName(): String**

```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
```

```
assertThat(p1, is(sameInstance(p2)));  // FAIL
assertThat(p1, is(equalTo(p2)));       // PASS
assertThat(p1, is(sameInstance(p3)));  // FAIL
assertThat(p1, is(equalTo(p3)));       // FAIL
```

Person
- firstName: String - lastName: String
+ Person(first:String, last:String) + getFirstName(): String + getLastName(): String + equals(anotherPerson:Object): boolean

```
if( this.firstName.equals(((Person)anotherPerson).getFirstName())
    && this.lastName.equals(((Person)anotherPerson).getLastName())){
  return true;
}
else{
  return false;
}
```


Alternatively…

```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
```

```
assertThat(p1, is(sameInstance(p2)));                                  // FAIL
assertThat(p1.getFirstName(), is(equalTo(p2.getFirstName())));         // PASS
assertThat(p1.getLastName(), is(equalTo(p2.getLastName())));           // PASS

assertThat(p1, is(sameInstance(p3)));                                  // FAIL
assertThat(p1.getFirstName(), is(equalTo(p3.getFirstName())));         // FAIL
assertThat(p1.getLastName(), is(equalTo(p3.getLastName())));           // FAIL
```

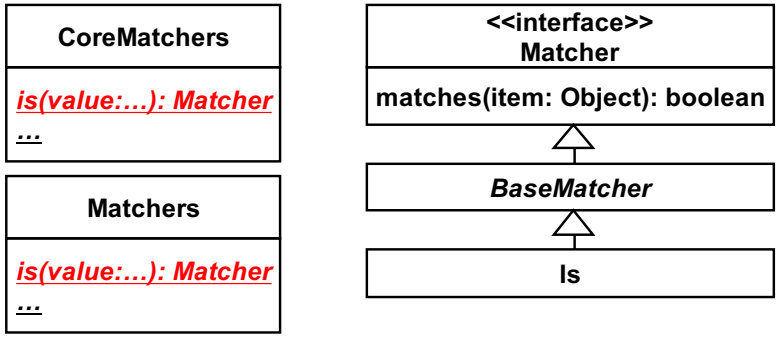

JUnit and Hamcrest

JUnit and Hamcrest

- Hamcrest provides many useful matchers for JUnit
  - junit.jar and hamcrest-core.jar from http://junit.org
  - hamcrest-all.jar from http://hamcrest.org
  - hamcrest-all.jar is a superset of hamcrest-core.jar.

Matchers in hamcrest-all.jar

- `org.hamcrest.CoreMatchers`
- `org.hamcrest.Matchers`
  - Contains static methods, each returning a matcher object that performs matching logic.
  - `Matchers` is a superset of `CoreMatchers`.



Matchers in hamcrest-all

- Examine String data
  - `assertThat("UMass Boston", containsString("UMass"))`
  - `assertThat("UMass Boston", startsWith("UMass"))`
  - `assertThat("UMass Boston", endsWith("Boston"))`
  - `assertThat("UMass Boston", equalToIgnoringCase("UMASS BOSTON"))`
  - `assertThat("  UMass", equalToIgnoringWhiteSpace("UMass"))`
  - `assertThat(actual, isEmptyOrNullString())`
  - `assertThat("", isEmptyString())`
  - `assertThat("UMass Boston", stringContainsInOrder(`
    `Arrays.asList("UM", "Boston"));`
  -

- Examine numbers
  - `assertThat(10, is(10));`
  - `assertThat(10.3, closeTo(10, 0.3)); // PASS`
  - `assertThat(10, is(greaterThan(9));`
  - `assertThat(10, is(greaterThanOrEqualTo(10));`
  - `assertThat(10, is(lessThan(11));`
  - `assertThat(10, is(lessThanOrEqualTo(10));`

- Examine arrays
  - `String[] strArray = {"UMass","Boston"};`
    `assertThat(strArray, array( is("UMass"),`
    `startWith("B")));`
  - `assertThat(strArray, arrayContaining("UMass","Boston");`
  - `assertThat(strArray, arrayContainingInAnyOrder("Boston","UMass");`
  - `assertThat(strArray, arrayWithSize(2));`
  - `assertThat(strArray, is(not(emptyArray())));`
  - `assertThat(strArray, hasItemInArray("UMass"));`
  - `assertArrayEquals(strArray,strArray);`

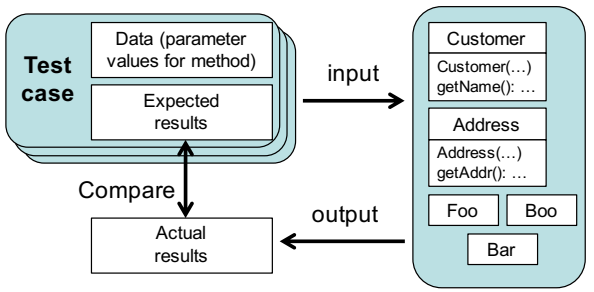
- Examine collections
  - `String[] strArray = {"UMass","Boston"};`
    `ArrayList<String> actual = Array.asList(strArray);`
    `String[] expected = {"UMass","Boston"};`
  - `assertThat(actual, contains(expected));`
  - `assertThat(actual, contains("UMass", "Boston");`
  - `assertThat(actual, contains( is("Umass"),`
    `startWith("B")));`
  - `String[] expected2 = {"Boston","UMass"};`
    `assertThat(actual, containsInAnyOrder(expected2));`
  - `assertThat(actual, hasItem("UMass"));`
  - `assertThat(actual, hasItems("Boston")));`
  - `assertThat(actual, hasItems("UMass", "Boston")));`
  - `assertThat(actual, hasItem( containsString("Mass") ));`
    `assertThat(actual, hasItem( endsWith("Mass") );`
    `assertThat(actual, everyItem( containsString("s"));`

- `String[] strArray = {"UMass","Boston"};`
  `ArrayList<String> actual = Array.asList(strArray);`
  `String[] expected = {"UMass","Boston"};`
- `assertThat(actual, hasSize(2));`
- `assertThat("UMass", isIn(strArray));`
- `assertThat(actual, not(empty());`

- Examine maps
  - `HashMap<String, Integer> actual = new HashMap<String, Integer>();`
    `actual.put("foo", 0);`
    `actual.put("boo", 10);`
    `actual.put("bar", 100);`
  - `assertThat(actual, hasEntry("foo", 0));`
  - `assertThat(actual, hasEntry( endWith("oo"), greaterThan(5)) );`
  - `assertThat(actual, hasKey("bar"));`
  - `assertThat(actual, hasKey( startWith("b") ));`
  - `assertThat(actual, hasValue(0));`
  - `assertThat(actual, hasValue( lessThanOrEqualTo(100) ));`

Test Fixtures

- Fixture
  - An instance of a class under test
    - A state(s) of the class instance
  - An instance of another class that the class under test depends on
    - A state(s) of the 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;
 }
]
```

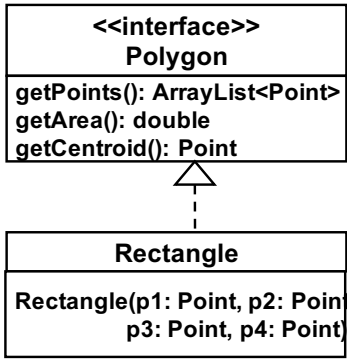
- Test class

```
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); }
}
```

Setting up fixtures

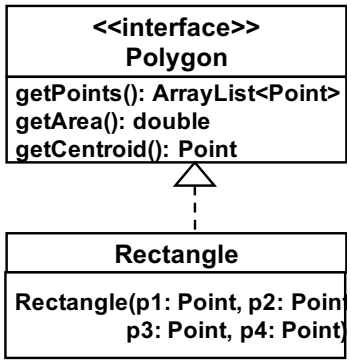
Inline Setup

```
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(4, is(cut.getArea())); }
}
```



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(4, is(cut.getArea())); }

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



Coverage of Unit Tests

Code Coverage

- How much code is *executed* by test cases.
  - Higher coverage means…
    - 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;
 }
}
```

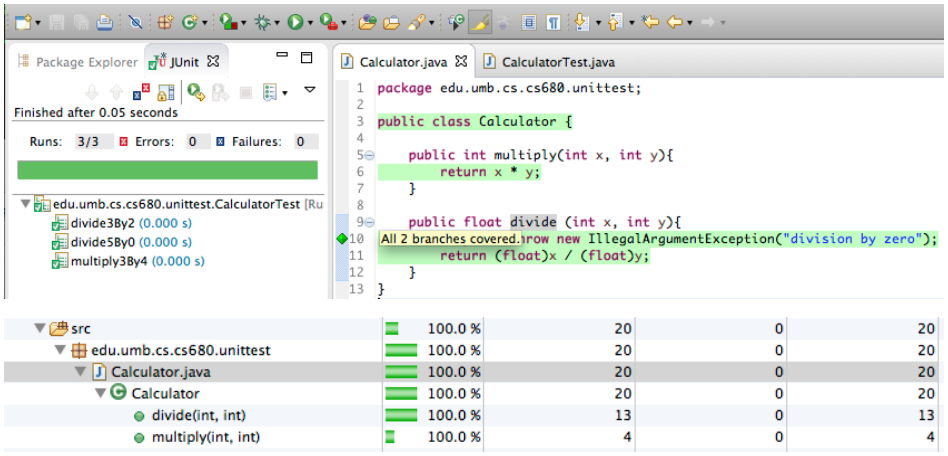
- Test class

```
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); }
}
```

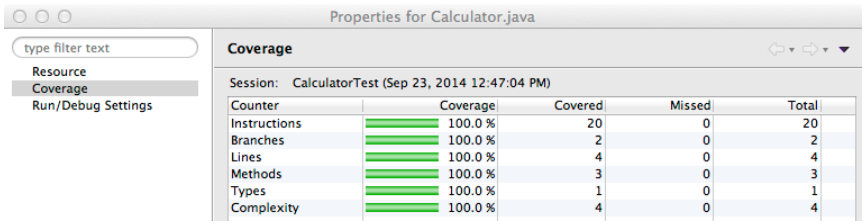
- Line coverage=100% (3/3)
- Branch coverage=100% (2/2)

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.



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



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
  - Jacoco provides ant tasks
    - e.g., <coverage> and <report>

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% coverage, 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 implicitly inserts 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.

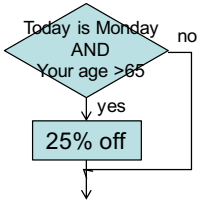
- Some exceptions may rarely occur.
  - e.g. IOExcepetion for file I/O operations
  - Test cases may not be able to reproduce all error cases to throw all exceptions. Coverage decreases.
  - Forget about it.
- Branching may be decided at random.

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

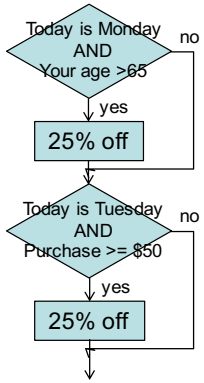
  - Both branches may not be covered by running a test case twice.
  - Repeat the test case some more times.

What about Condition Coverage?

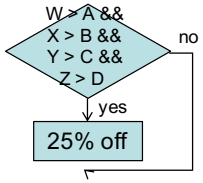
- Condition coverage
  - Each combination of true-false conditions has been executed at least once?
  - EclEmma does not support it.
    - Need to manually keep track of it.



- Monday?, >65?
  - 4 true-false combinations
    - Y-Y, Y-N, N-Y, N-N
- Need 4 tests
  - 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
  - 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-N_1, N_2-N_2
  - Mon, <=65, >=$50: Y_1-N_1, N_2-Y_2 (redundant)
  - Mon, <=65, <$50: Y_1-N_1, N_2-N_2 (redundant)
  - Tue, >65, >=$50: N_1-Y_1, Y_1-Y_1
  - Tue, >65, <$50: N_1-Y_1, Y_1-N_1 (redundant)
  - Tue, <=65, >=50: N_1-N_1, Y_1-Y_1 (redundant)
  - Tue, <=65, <$50: N_1-N_1, Y_1-N_1



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

HW 8

- Measure and report coverage for your HW 7-1 code with JaCoCo (and Ant)
  - Reach 100% coverage in all metrics.
- Turn in build.xml, src and test/src.
  - Generate a coverage report in HTML at the “test” directory.