HW 1-2: Step 2

- Implement `CalculatorTest`
  - Package: `edu.umb.cs680.hw01`
  - Define some extra test methods in addition to `multiply3By4()`, `divide3By2()`, `divide5By0()`.
    - e.g., A float number times a float number
      - `Multiple2_5By5_5()`
    - e.g., A float number over a float number
      - `Multiple2_5By5_5()`
- Follow the expected directory structure

- Place all test classes under the "test" directory
  - [project directory]
    - build.xml (You can name it as you like.)
    - src [source code directory]
      - edu/umb/cs680/hw01/Foo.java
      - edu/umb/cs680/hw02/Boo.java
    - bin [byte code directory]
      - edu/umb/cs680/hw01/Foo.class
      - edu/umb/cs680/hw02/Boo.class
    - test [test code directory]
      - src
        - » edu/umb/cs680/hw01/FooTest.java
        - » edu/umb/cs680/hw02/BooTest.java
      - bin
        - » edu/umb/cs680/hw01/FooTest.class
        - » edu/umb/cs680/hw02/BooTest.class

- Use Ant to build and run `Calculator` and `CalculatorTest`
  - Set up the directory where `Calculator.class` is placed.
    - `<proj dir>/bin/edu/umb/cs680/hw01`
  - Set up the directory where `CalculatorTest.class is placed.`
    - `<proj dir>/test/bin/edu/umb/cs680/hw01`
  - Set up CLASSPATH
    - `<proj dir>/bin`
    - `<proj dir>/test/bin`
    - JUnit JAR files (see the next slide)
  - Compile `Calculator.java` and generate `Calculator.class` to `<proj dir>/bin/edu/.../hw01`
  - Compile `CalculatorTest.java` and generate `CalculatorTest.class` to `<proj dir>/test/bin/edu/.../hw01`
  - Run `CalculatorTest.class` with JUnit
  - Run `Calculator.class`

JUnit JAR Files

- JUnit API JAR files
  - `junit-jupitar-api.jar`
  - `junit-jupitar-engine.jar`
  - `junit-jupitar-params.jar`
  - `apiguardian.jar`
  - `opentest4j.jar`
  - No need to use `junit-vintage-*.jar`
- JUnit Platform JAR files
  - `junit-platform-commons.jar`
  - `junit-platform-engine.jar`
  - `junit-platform.launcher.jar`
  - `junit-platform-runner.jar:`
  - `junit-platform-suite-api.jar`
- Include BOTH types of JAR files in CLASSPATH
  - https://ant.apache.org/manual/Tasks/junitlauncher.html
  - https://github.com/junit-team/junit5-samples/tree/r5.5.2/junit5-jupiter-starter-ant

- Include JUnit JAR files to CLASSPATH in an OS-independent way.
  - Never reference those JAR files with absolute paths in your build file. Rely on relative paths.
  - Create a new environment variable JUNIT and set the directory that contains those JAR files.
    - You do that on your OS.
    - I do the same on my OS.

  - Reference the env variable JUNIT to configure CLASSPATH in your build script.
    - `<property environment="env"/>`

```
<path id="classpath">
  <pathelement location="bin" />
  <pathelement location="test/bin" />
  <fileset dir="${env.JUNIT}"
    includes="org.junit*.jar" excludes="org.junit.vintage.*.jar"/>
  <fileset dir="${env.JUNIT}" includes="org.opentest4j*.jar" />
  <fileset dir=... />
</path>

<javac …>
<classpath refid="classpath"/>
  …
</javac>

<junitlauncher …>
<classpath refid="classpath"/>
  …
</junitlauncher>
```

    - Useful tasks for debugging
      - `<echo message="${env.JUNIT}" />`
      - `<echo message="${toString:classpath}" />`

Your Machine/OS

Environment variables:
JUNIT=C:\junit\...

My Machine/OS

Environment variables:
JUNIT=~/code/junit/…

Shared across two machines

Your build.xml

```
<path id="classpath">
    <pathelement location="bin" />
    <pathelement location="test/bin" />
    <fileset dir="${env.JUNIT}"..../>
</path>
```

- Your build script should reference the env variable JUNIT, so it does not have to include absolute paths.
- Keep your build script OS-independent.

  - Run JUnit from Ant. Use <junitlauncher> task in Ant.
    - c.f. JUnit documentations (API docs, user manual, etc.)
    - c.f. Ant documentations

```
- <junitlauncher printSummary="true">
    <classpath refid="classpath" />
    <test outputdir="test"
          name="edu.umb.cs680.hw01.CalculatorTest"/>
    <listener type="legacy-plain" sendSysOut="true"/>
  </junitlauncher>

- <junitlauncher printSummary="true">
    <classpath refid="classpath" />
    <testclasses outputdir="test">
      <fileset dir="${test.bin}">
        <include name="edu/umb/cs680/hw01/*Test.class"/>
      </fileset>
      <listener type="legacy-plain" sendSysOut="true"/>
    </testclasses>
  </junitlauncher>
```

  - Use this option (the second) option.

HW Submission

- Turn in an archive file (.rar, .tar.gz, .7z, etc.) that contains
  - build.xml
  - "src" sub-directory
  - "test/src" sub-directory
- Send the archive to **umasscs680@gmail.com**
  - Avoid the .zip format. Google sometimes/often filter it out.
  - Or, place the archive somewhere online and email me a link to it.
- **DO NOT include binary files (.class and .jar files)**
- Deadline: Oct 15 (Tue) midnight

FYI: For Running Ant on Your IDE

- Your IDE may not be able to capture your OS's environment variables by default.

```
<property environment="env"/>
<path id="classpath">
    <pathelement location="bin" />
    <pathelement location="test/bin" />
    <pathelement path="${env.JUNIT}" />
</path>
```

  - Useful tasks for debugging
    - `<echo message="${env.JUNIT}" />`
    - `<echo message="${toString:classpath}" />`
- If your IDE doesn't, check out how you can have it do that.
  - e.g., Run Eclipse from a shell (not by double-clicking an Eclipse icon)
    - e.g., `/Applications/Eclipse.app/Contents/MacOS/eclipse`

Suggestions

- Start as early as possible.
- Go step by step
  - If you are not familiar with Ant…
    - compile and run your code for Step 1 without using Ant first and then use Ant to build the code
  - Step 1 and then Step 2
- Expect "Death by XML"
  - You may need to spend a few hours, a half day or even a full day to make your build.xml run correctly.
  - That's a part of your learning curve...

Extra Notes on Unit Testing

What to Test?

- In principle, you should write a unit test(s) for every public method of your class.
- However, methods with very obvious functionalities/behaviors do not need unit tests.
  - e.g. simple getter and setter methods
- Write a unit test whenever you feel you need to comment the behavior of a method.

- In CS680, write as many test cases as possible.
  - If you test all public methods, you will get some extra points in grading.
  - If you test simple getter/setter methods only and skips testing more important methods, you will lose most points in the testing portion of your work.
  - If you test major/important methods but skips testing simple getter/setter methods, you will not lose any points.

Keep Test Cases Simple

- Write *single-purpose* tests.
  - Have each test case (test method) focus on a distinctive (external) behavior of a tested class
  - Do not test multiple behaviors in a single test case
    - e.g., divide5by0, multiply3By4
      - Rather than testCalculator, testDivision
- Give a specific name to your test method, so many others can understand what is being tested.
- In CS680, if you use vague method names, you will lose points in the testing portion of your work.
  - Long, weird-looking method names are fine!

Use Test Cases as Documentation

- Lasting, runnable and reliable documentation on the capabilities of the classes you write.
- Can serve as sample code to use your classes/methods.
  - Useful when you forgot how to use a class/method you implemented.
  - Useful when you use a class/method that someone else implemented.
  - No need to write sample use cases and sample code in API documentation and other docs.
- Can replace a lot of comments.
  - Cannot completely replace comments, but you often do not have to write a looooong (Javadoc) comments.

Use Specific and Meaningful Names

- 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
  - *given*SomePrecondition*When*DoingSomething*Then*SomeResultOccurs
    - given**TwoNumbers**When**DivisionBy0**Then**IllegalArgumentExceptionOccurs**
      - divide(5,0)
    - given**TwoStrings**When**DivisionBy0**Then**IllegalArgumentExceptionOccurs**
      - 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.

- 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?

JUnit API (cont'd)

Key API in JUnit: `Assertions`

- `org.junit.jupiter.api.Assertions`
  - Contains a series of *static* assertion methods.

```
- assertTrue( boolean condition )
- assertFalse( boolean condition )
```

  - » Returns if `condition` is true/false.
    - »
```
Calculator cut = new Calculator();
assertTrue( cut.multiply(3, 4) > 0 );
```
    - »
```
assertTrue( cut instanceof Calculator);
```
  - » Throws an `org.opentest4j.AssertionFailedError` if `condition` is not equal to the expected boolean state.
    - » JUnit catches it; your test cases don't have to.
  - » JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`


```
- assertEquals( int expected, int actual )
- assertEquals( float expected, float actual )
...
- assertEquals( Object expected, Object actual )
```

  - » Defined for each primitive type and `Object`
  - » Returns if two values (expected and actual values) match.
    - »
```
float expected = 12;
float actual = cut.multiply(3,4);
assertEquals( expected, actual );
```
  - » Throws an `org.opentest4j.AssertionFailedError` if two values do not match.
    - » JUnit catches it; your test cases don't have to.
  - » JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

- Assertion methods perform *auto-boxing* and *auto-unboxing* wherever necessary and possible.
  - `assertEquals( int expected, int actual )`
  - expected: int, actual: int
    - »
```
int expected = 10;
int actual = 20;
assertEquals( expected, actual );
```
  - expected: int, actual: Integer
    - »
```
int expected = 10;
Integer actual = 20;
assertEquals( expected, actual );
```
  - expected: Integer, actual: Integer
    - »
```
Integer expected = 10;
Integer actual = 20;
assertEquals( expected, actual );
```


Just in case… Auto-boxing and Auto-unboxing

- Automatic conversion between a primitive type value and a wrapper class instance
  -
```
int numInt = 10;
Integer numInteger = numInt;
```
    - No need to write…
    -
```
int numInt = 10;
Integer numInteger = new Integer(numInt);
// OR
Integer numInteger = Integer.valueOf(numInt);
```
  -
```
Integer numInteger = Integer.valueOf(10);
int numInt = numInteger;
```
    - No need to write…
    -
```
Integer numInteger = Integer.valueOf(10);
int numInt = numInteger.intValue();
```

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Key API in JUnit: `Assertions`

- Assertion methods perform *auto-boxing* and *auto-unboxing* wherever necessary and possible.
  - `assertNull( Object actual )`
  - `assertNonNull( Object actual )`
    - » Defined for `Object`
      - » Not for primitive types.
    - » Returns if a value is null (or NOT null).
      - »
```
Calculator actual;
assertNull( actual );
```
      - »
```
Calculator actual = new Calculator();
assertNonNull( actual );
```
      - »
```
float actual = 12;
assertNonNull( actual );
```
    - » JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

  -
```
assertArrayEquals(int[] expected, int[] actual)
assertArrayEquals(float[] expected, float[] actual)
...
assertArrayEquals(Object[] expected, Object[] actual)
```
    - » Defined for each primitive type and `Object`
    - » Returns if two arrays (expected and actual arrays) match.
      -
```
String[] s1 = {"UMass", "Boston"};
String[] s2 = {"UMass", "Amherst"};
assertArrayEquals(s1, s2);
```
    - » Throws an `org.opentest4j.AssertionFailedError` if two values do not match.
    - » JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

Positive and Negative Tests

- *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) as expected.

Positive Tests

```
@Test
public void writeToTestFile(){
 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 method fails with `fail()`. Otherwise, the test case passes.
- Clear, logic-wise, but try-catch-finally blocks may clutter a test case.

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

```
@Test
public void writeToTestFile() throws IOException {
 BufferedWriter writer = new BufferedWriter(
                         new FileWriter("test.txt");
 writer.write("test data");
 writer.close();
}
```

- JUnit catches an `IOException` and judges that the test method fails.
  - `write()` throws it originally, and `readFromTestFile()` re-throws it (to JUnit)

Negative Tests

- Verify that tested code throws an exception(s) as expected.
  - Understand the conditions that cause tested code to throw an exception and test those conditions in test methods
- 2 Common ways
  - Write a test case with try-catch blocks
  - Use `Assertions.assertThrows()`
    - To be introduced later in this semester.

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


HW 2: `PrimeGenerator`

- Generates prime numbers in between two input numbers (`from` and `to`)
  - ```
    Class PrimeGenerator {
      protected long from, to;
      protected LinkedList<Long> primes;

      public void generatePrimes(){ ... }
      public LinkedList<Long> getPrimes(){ return primes };
      ...
    ```
- Client code (test case)
  - ```
    PrimeGenerator gen = new PrimeGenerator(1, 10);
    gen.generatePrimes();
    Long[] expectedPrimes = {2L, 3L, 5L, 7L};
    assertArrayEquals( expectedPrimes,
                       gen.getPrimes().toArray() );
    ```


- Place `PrimeGenerator` in the package `edu.umb.cs680.hw02`
- Define **`PrimeGeneratorTest`**
  - Write more than one test method
    - Test a regular case
      - Use `assertArrayEquals()`
    - Test error cases where wrong ranges are given.
      - e.g., [-10, 10], [100, 1]
  - You can name test methods as you like. Make sure to give them specific names.
- Deadline: Oct 17 (Thu) midnight