# Roman Numerals Kata

## Slide 12: Set Up Java Project

Extract project from RomanConversionStarting to project directory. Recommend this be done ahead of presentation where you can show only one class set up so far

## Slide 15: Let’s Get Coding – Refactoring Specs

Create a file called RomanNumerals.feature in the src/test/resources/cucumber/kata directory:

**Feature: In order to reduce confusion with Roman Numerals**

**As A reader of roman numerals**

**I want to translate the numbers to Arabic numbers**

**Scenario Outline:** translate simple numbers to Arabic  
 **When** I ask for a translation of a "<Roman Numeral>"  
 **Then** I get the correct "<Arabic Number>"  
 **Examples:  
 |** *Roman Numeral* **|** *Arabic Number* **|  
 | I | 1 |  
 | V | 5 |  
 | X | 10 |  
 | L | 50 |  
 | C | 100 |  
 | D | 500 |  
 | M | 1,000|**

Now lets create a separate feature file that will hold the error handling, call it RomanNumeralErrors.feature file with this description:

**Feature**: **In order to reduce confusion with incorrect numbers**

**As A user of roman numerals**

**I want to capture incorrect translation attempts**

**Scenario Outline:** translation error numbers  
 **When** I ask for a translation of a "<Roman Numeral>"  
 **Then** I get the invalid numeral error  
 **Examples:  
 |** *Roman Numeral* **|  
 | B |  
 | # |  
 | 2 |**

Move the errors from above into this feature file.

## Slide 17: IDE Will Generate Glue Stubs

Use Intellij to generate the step definition stubs. Demo the creation of the step definition

Key item is the class name of **TestRomanNumeralFeatureSteps** and java (not Java 8) coding style into the **test\java\cucumber\kata\step\_definitions** directory. Ensure it is in the **cucumber.kata** package!

Now we need the code to make use of this helper class in the testing (variables and two method bodies):

(add now to speed up coding process)

**private** ConversionEntry **aRomanConversion**;  
**private** ConversionEntry **aComplexConversion**;  
**private** ConversionEntry **anErrorConversion**;  
**private** ConversionEntry **anArabicConversion**;

Code for the When clause:

**aRomanConversion** = **new** ConversionEntry();  
**aRomanConversion**.setRomanNumeral(romanNumeral);  
*//no test done, this is setup for the next step*

Code for the Then clause:

*assertEquals*(Long.*valueOf*(anArabicNumber), Long.valueOf(RomanNumbers.toArabic(**aRomanConversion**.getRomanNumeral())));

## Slide 18: Using Examples in Spec

First the helper **test** class to hold values from When to Then clause:

*//helper class*

public class ConversionEntry {

private String romanNumeral;

private Integer arabicNumber;

public void setRomanNumeral(String romanNumeral) {

this.romanNumeral = romanNumeral;

}

public void setArabicNumber(int anArabicNumber) {

this.arabicNumber = anArabicNumber;

}

public String getRomanNumeral() {

return romanNumeral;

}

public int getArabicNumber() {

return arabicNumber;

}

}

## Slide 20: Let’s Get Coding - TDD

Add a class to run the unit tests **TestRomanNumeralUnitTests**:

package cucumber.kata.junit;

/\*\*

\* Created by Shane on 3/1/2017.

\*/

public class **TestRomanNumeralUnitTests** {

@Test

public void testSimpleArabicToRoman () {

*assertEquals*(**"1 should be the outcome"**,1, RomanNumbers.*toArabic*(**"I"**));  
*assertEquals*(**"5 should be the outcome"**,5, RomanNumbers.*toArabic*(**"V"**));

}

}

## Slide 21: Now For Application Coding

Now to start coding in the application, add:

* The directory **main**, then **java** under src
* Add the package **cucumber.kata** to the java directory
* Now add the class ~src/main/java/cucumber/kata **RomanNumbers**

Put this class method in:

public static long toArabic(String aRomanNumber) throws IllegalArgumentException{

RomanNumeral aRomanNumeral = RomanNumeral.valueOf(aRomanNumber);

switch (aRomanNumeral) {

case I:

return RomanNumeral.I.getArabic();

case V:

return RomanNumeral.V.getArabic();

default:

throw new IllegalArgumentException("Invalid Digit " + aRomanNumber);

}

}

I decided to use an enum as part of my design – thus the “Design” aspect of TDD. Add java class file called /src/main/java/cucumber/kata/RomanNumeral:

**package** cucumber.kata;

public enum RomanNumeral {

V(5,"V"),

I(1,"I");

private int arabic;

private String name;

private RomanNumeral(int arabic, String name){

this.arabic = arabic;

this.name = name;

}

public int getArabic() {

return arabic;

}

public String getName() {

return name;

}

}

Now use the IDE to fix the remaining compilation errors by having it generate the import lines. Once all of the errors are clear, we can try running the **UNIT** test.

Now we can run our acceptance test, based on what we copied in from our story. Now **two of our acceptance tests pass**! We are on the right track.

## Slide 17 to Slide 24: Our Testing Allocation Decision

Now get more of the acceptance tests to pass:

* Talk about **the three amigos evaluating and deciding on a test approach to balance unit and functional tests**. Basically do the minimum on unit tests as the functional tests cover a lot of territory.
* We will use the unit tests to drive the rough design, but leave the full scope to the functional tests.

Update our basic translation code in the RomanNumbers class:

case X:

return RomanNumeral.X.getArabic();

case C:

return RomanNumeral.C.getArabic();

case D:

return RomanNumeral.D.getArabic();

case M:

return RomanNumeral.M.getArabic();

case L:

return RomanNumeral.L.getArabic();

And lets fix up the enum to contain the rest of the definitions:

// sequence is important here

M(1000, "M"),

CM(900, "CM"),

D(500,"D"),

CD(400,"CD"),

C(100, "C"),

XC(90,"XC"),

L(50,"L"),

XL(40,"XL"),

X(10,"X"),

IX(9,"IX"),

V(5,"V"),

IV(4,"IV"),

I(1,"I");

On code coverage, Now go ahead and run with coverage. Wow, that is cool!

## Slide 25: On To Complex Numbers

Here is the spec:

Scenario Outline: translate common number combinations

When I ask for a translation of a combination "<Complex Roman Numeral>"

Then I get the correct combination translation "<Arabic Number>"

Examples:

| Complex Roman Numeral | Arabic Number |

| IV | 4 |

| IX | 9 |

| XC | 90 |

| XL | 40 |

| CD | 400 |

| CM | 900 |

| MMXIII | 2013 |

| MMMCMXCIX | 3999 |

Generate the steps for complex Roman Numerals into the same class **TestRomanNumeralFeatureSteps**.

Code for When (remember to adjust the input parameter name:

aComplexConversion = new ConversionEntry();

aComplexConversion.setRomanNumeral(romanNumeral);

//no test done, this is setup for the next step

Code for Then:

*assertEquals*(Long.*valueOf*(anArabicNumber), Long.*valueOf*(RomanNumbers.*toArabic*(**aComplexConversion**.getRomanNumeral())));

Now we have more failing functional tests. Need to write another unit test so I can write some more code!

## Slide 26: Code for Complex Numbers

We have our functional test failing – our design does not cover complex numbers. Now to create a unit test to drive this part of our design by adding this unit test to **TestRomanNumeralUnitTests**:

@Test

public void testComplexArabicToRoman () {

*assertEquals*(**"256 should be the outcome"**, 256, RomanNumbers.*toArabic*(**"CCLVI"**) );

}

When I run this test, it now fails – there is not a enum of CCLVI. Now **back to refactoring** and getting rid of the case statement in **RomanNumbers**. We can confidently and quickly refactor our application because we have an extensive safety net that points out where our changes fail to meet prior specs and designs!

public static long toArabic(String aRomanNumber) throws IllegalArgumentException{

int arabic = 0;

String romanName;

for (RomanNumeral currentRomanNumber : RomanNumeral.values()) { //Iterate through all of the enums

romanName = currentRomanNumber.getName();

while (aRomanNumber.startsWith(romanName)) { //We might have many instances of the same letter(s)

arabic += currentRomanNumber.getArabic(); //based on finding a match, grab the associated value

aRomanNumber = aRomanNumber.substring(romanName.length()); //remove the letter(s) we just found

}

}

if (aRomanNumber.length() > 0) {

throw new IllegalArgumentException("Invalid Digit " + aRomanNumber); //if we have anything left over, that is bad

}

return arabic;

}

Now the unit tests and functional tests work.

## Slide 29: Refactoring Edge Specs

Move **all** of the error handling specs into the **RomanNumeralErrors.feature** file.

**Feature:** In order to reduce confusion with incorrect numbers  
 As A user of roman numerals  
 I want to capture incorrect translation attempts

Scenario Outline: translation error numbers

When I ask for a translation of an invalid "<Roman Numeral>"

Then I get the invalid numeral error

Examples:

| Roman Numeral |

| B |

| # |

| 2 |

Scenario Outline: flag malformed numbers

When I ask for a translation of an invalid "<Bad Roman Numeral>"

Then I get the invalid numeral error

Examples:

| Bad Roman Numeral |

| VC |

| LD |

| IIXV |

| IC |

| CM |

| XD |

| XM |

Scenario Outline: flag bad Arabic numbers

When I ask for a Roman translation of a invalid "<Bad Arabic Numeral>"

Then I get the invalid Arabic error

Examples:

| Bad Arabic Numeral |

| 0 |

| 4000 |

## Slide 30: Add Roman Error Step/Glue Code

Now add code for translation error numbers.

For when:

anErrorConversion = new ConversionEntry();

anErrorConversion.setRomanNumeral(romanNumeral);

//no test done, this is setup for the next step

Code for then:

**try** {  
 RomanNumbers.*toArabic*(**anErrorConversion**.getRomanNumeral());  
 *fail*(**"Invalid RomanNumeral, no exception thrown."**); *//if the exception is not thrown, then this code will run*}  
**catch** (IllegalArgumentException e) {  
 *//do nothing here as the exception was expected*}

Add **two** error unit tests: A simple error unit test:

@Test  
**public void** testInvalidRomanNumeralB() {  
 **try** {  
 RomanNumbers.*toArabic*(**"B"**);  
 *fail*(**"Invalid Digit B"**);  
 }  
 **catch** (IllegalArgumentException expectedException)  
 {*//do nothing* }  
}

Let’s add our second unit test to cover complex number errors:

@Test  
**public void** testInvalidRomanNumeralComboVC() {  
 **try** {  
 RomanNumbers.*toArabic*(**"VC"**);  
 *fail*(**"Invalid Digit C"**);  
 }  
 **catch** (IllegalArgumentException expectedException)  
 {*//do nothing* }  
}

Now run and eliminate **surprise error in specs**. “CM” is a valid combination!

# Slide 32: Now Arabic to Roman Code

Last bit of acceptance criteria – story 4. Add the following to specify the normal flow from Roman Numeral to Arabic number feature file.

Scenario Outline: translate simple numbers to Roman

When I ask for a Roman translation of a "<Arabic Number>"

Then I get the correct Roman Numeral "<Roman Numeral>"

Examples:

| Arabic Number | Roman Numeral |

| 1 | I |

| 5 | V |

| 256 | CCLVI |

| 1001 | MI |

| 1993 | MCMXCI |

Now add the step code to **TestRomanNumeralFeature** and see the test fail:

Here is the When code:

anArabicConversion = new ConversionEntry();

anArabicConversion.setArabicNumber(Integer.parseInt(anArabicNumber));

//no test done, this is setup for the next step

Now the Then code:

assertEquals(aRomanNumber, RomanNumbers.toRoman(anArabicConversion.getArabicNumber()));

Won’t compile as we need a **new method**.

Add this unit test to **TestRomanNumeralUnitTests**:

@Test

public void testArabicToRoman () {

*assertEquals*(**"I should be the outcome"**, **"I"**, RomanNumbers.*toRoman*(1));  
*assertEquals*(**"V should be the outcome"**, **"V"**, RomanNumbers.*toRoman*(5));

}

## Slide 33: Arabic to Roman Code

Add the following method to **RomanNumbers** class:

public static String toRoman(int arabic) {

StringBuilder result = new StringBuilder();

for (RomanNumeral numeral : RomanNumeral.values()) { //Iterate through all of the enums, large to small

while (arabic >= numeral.getArabic()) { //starting with the larger values, subtract them out

arabic -= numeral.getArabic(); //reduce number with matching number

result.append(numeral.name()); //append found roman value

}

}

return result.toString();

}

## Slide 34: Fix the Specs

Change the spec to this:

| 1993 | MCMXCIII |

## Slide 35: Arabic Error Step Code

Add step test code to **TestRomanNumeralFeatureSteps** :

Code for When:

**anArabicConversion** = **new** ConversionEntry();  
**anArabicConversion**.setArabicNumber(Integer.*parseInt*(anArabicNumber));  
*//no test done, this is setup for the next step*

Code for When:

**try** {  
 RomanNumbers.*toRoman*(**anArabicConversion**.getArabicNumber());  
 *fail*(**"Invalid Arabic Numeral, no exception raised."**);  
 }  
 **catch** (IllegalArgumentException e) {  
 *//do nothing here as the exception was expected* }

Add a unit test in **TestRomanNumeralUnitTests** to drive the code:

@Test

public void testInvalidRomanNumeralOne() {

try {

RomanNumbers.toRoman(0);

fail("Only numbers between 1 and 3999 is supported.");

}

catch (IllegalArgumentException expectedException)

{//do nothing

}

}

This test fails, we should have seen an exception. So let’s add the code to get the correct exception in **RomanNumbers**:

New variables in the class:

private static final int LOWER\_NUMERAL\_LIMIT = 1;

private static final int UPPER\_NUMERAL\_LIMIT = 3999;

New code:

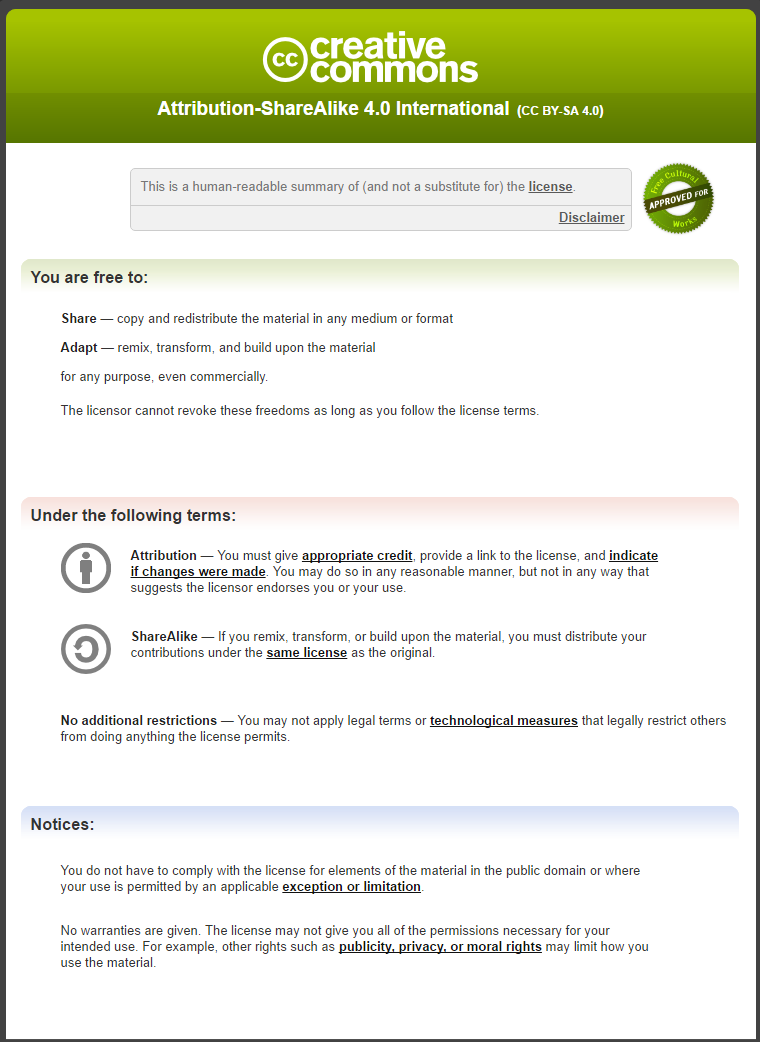
**public static** String toRoman(**int** arabic)**throws** IllegalArgumentException {

//this is the new code to create the exception  
 **if** (arabic < ***LOWER\_NUMERAL\_LIMIT*** || arabic > ***UPPER\_NUMERAL\_LIMIT***) {  
 **throw new** IllegalArgumentException(String.*format*(**"Only numbers between %s and %s is supported."**, ***LOWER\_NUMERAL\_LIMIT***, ***UPPER\_NUMERAL\_LIMIT***));  
 }

StringBuilder result = **new** StringBuilder();  
 **for** (RomanNumeral numeral : RomanNumeral.*values*()) { *//Iterate through all of the enums, large to small* **while** (arabic >= numeral.getArabic()) { *//starting with the larger values, subtract them out* arabic -= numeral.getArabic(); *//reduce number with matching number* result.append(numeral.name()); *//append found roman value* }  
 }  
 **return** result.toString();  
}

We have completed the coding. Now re-check the **code coverage** and finish up with the slides.

Scroll through the tests results, showing all of the business rule examples and the count of tests.

[](https://creativecommons.org/licenses/by-sa/4.0/)