# KataBankOCR

**Problem Description**

User Story 1

You work for a bank, which has recently purchased an ingenious machine to assist in reading letters and faxes sent in by branch offices. The machine scans the paper documents, and produces a file with a number of entries which each look like this:

\_ \_ \_ \_ \_ \_ \_

| \_| \_||\_||\_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_|

Each entry is 4 lines long, and each line has 27 characters. The first 3 lines of each entry contain an account number written using pipes and underscores, and the fourth line is blank. Each account number should have 9 digits, all of which should be in the range 0-9. A normal file contains around 500 entries.

Your first task is to write a program that can take this file and parse it into actual account numbers.

User Story 2

Having done that, you quickly realize that the ingenious machine is not in fact infallible. Sometimes it goes wrong in its scanning. The next step therefore is to validate that the numbers you read are in fact valid account numbers. A valid account number has a valid checksum. This can be calculated as follows:

account number: 3 4 5 8 8 2 8 6 5

position names: d9 d8 d7 d6 d5 d4 d3 d2 d1

checksum calculation:

(d1+2\*d2+3\*d3 +..+9\*d9) mod 11 = 0

So now you should also write some code that calculates the checksum for a given number, and identifies if it is a valid account number.

User Story 3

Your boss is keen to see your results. He asks you to write out a file of your findings, one for each input file, in this format:

457508000

664371495 ERR

86110??36 ILL

ie the file has one account number per row. If some characters are illegible, they are replaced by a ?. In the case of a wrong checksum, or illegible number, this is noted in a second column indicating status.

User Story 4

It turns out that often when a number comes back as ERR or ILL it is because the scanner has failed to pick up on one pipe or underscore for one of the figures. For example

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| || ||\_ | | ||\_

| \_||\_||\_||\_| | | | \_|

The 9 could be an 8 if the scanner had missed one |. Or the 0 could be an 8. Or the 1 could be a 7. The 5 could be a 9 or 6. So your next task is to look at numbers that have come back as ERR or ILL, and try to guess what they should be, by adding or removing just one pipe or underscore. If there is only one possible number with a valid checksum, then use that. If there are several options, the status should be AMB. If you still can't work out what it should be, the status should be reported ILL.

**Clues**

I recommend finding a way to write out 3x3 cells on 3 lines in your code, so they form an identifiable digits. Even if your code actually doesn't represent them like that internally. I'd much rather read

" " +

"|\_|" +

" |"

than

" |\_| |"

anyday.

When Christophe and Emmanuel presented this Kata at XP2005 they worked on a solution that made extensive use of recursion rather than iteration. Many people are more comfortable with iteration than recursion. Try this kata both ways.

Some gotchas to avoid:

- be very careful to read the definition of checksum correctly. It is not a simple dot product, the digits are reversed from what you expect.

- The spec does not list all the possible alternatives for valid digits when one pipe or underscore has been removed or added

- don't forget to try to work out what a ? should have been by adding or removing one pipe or underscore.

**Suggested Test Cases**

If you want to just copy and paste these test cases into your editor, I suggest first clicking "edit this page" so you can see the source. Then you can be sure to copy across all the whitespace necessary. Just don't save any changes by mistake.

use case 1

\_ \_ \_ \_ \_ \_ \_ \_ \_

| || || || || || || || || |

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 000000000

| | | | | | | | |

| | | | | | | | |

=> 111111111

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

=> 222222222

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 333333333

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

| | | | | | | | |

=> 444444444

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 555555555

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 666666666

\_ \_ \_ \_ \_ \_ \_ \_ \_

| | | | | | | | |

| | | | | | | | |

=> 777777777

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 888888888

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 999999999

\_ \_ \_ \_ \_ \_ \_

| \_| \_||\_||\_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_|

=> 123456789

use case 3

\_ \_ \_ \_ \_ \_ \_ \_

| || || || || || || ||\_ |

|\_||\_||\_||\_||\_||\_||\_| \_| |

=> 000000051

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| || ||\_ | | | \_

| \_||\_||\_||\_| | | | \_|

=> 49006771? ILL

\_ \_ \_ \_ \_ \_ \_

| \_| \_||\_| \_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_

=> 1234?678? ILL

use case 4

| | | | | | | | |

| | | | | | | | |

=> 711111111

\_ \_ \_ \_ \_ \_ \_ \_ \_

| | | | | | | | |

| | | | | | | | |

=> 777777177

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_|| || || || || || || || |

|\_ |\_||\_||\_||\_||\_||\_||\_||\_|

=> 200800000

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 333393333

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 888888888 AMB ['888886888', '888888880', '888888988']

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 555555555 AMB ['555655555', '559555555']

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 666666666 AMB ['666566666', '686666666']

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 999999999 AMB ['899999999', '993999999', '999959999'] **Problem Description**

User Story 1

You work for a bank, which has recently purchased an ingenious machine to assist in reading letters and faxes sent in by branch offices. The machine scans the paper documents, and produces a file with a number of entries which each look like this:

\_ \_ \_ \_ \_ \_ \_

| \_| \_||\_||\_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_|

Each entry is 4 lines long, and each line has 27 characters. The first 3 lines of each entry contain an account number written using pipes and underscores, and the fourth line is blank. Each account number should have 9 digits, all of which should be in the range 0-9. A normal file contains around 500 entries.

Your first task is to write a program that can take this file and parse it into actual account numbers.

User Story 2

Having done that, you quickly realize that the ingenious machine is not in fact infallible. Sometimes it goes wrong in its scanning. The next step therefore is to validate that the numbers you read are in fact valid account numbers. A valid account number has a valid checksum. This can be calculated as follows:

account number: 3 4 5 8 8 2 8 6 5

position names: d9 d8 d7 d6 d5 d4 d3 d2 d1

checksum calculation:

(d1+2\*d2+3\*d3 +..+9\*d9) mod 11 = 0

So now you should also write some code that calculates the checksum for a given number, and identifies if it is a valid account number.

User Story 3

Your boss is keen to see your results. He asks you to write out a file of your findings, one for each input file, in this format:

457508000

664371495 ERR

86110??36 ILL

ie the file has one account number per row. If some characters are illegible, they are replaced by a ?. In the case of a wrong checksum, or illegible number, this is noted in a second column indicating status.

User Story 4

It turns out that often when a number comes back as ERR or ILL it is because the scanner has failed to pick up on one pipe or underscore for one of the figures. For example

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| || ||\_ | | ||\_

| \_||\_||\_||\_| | | | \_|

The 9 could be an 8 if the scanner had missed one |. Or the 0 could be an 8. Or the 1 could be a 7. The 5 could be a 9 or 6. So your next task is to look at numbers that have come back as ERR or ILL, and try to guess what they should be, by adding or removing just one pipe or underscore. If there is only one possible number with a valid checksum, then use that. If there are several options, the status should be AMB. If you still can't work out what it should be, the status should be reported ILL.

**Clues**

I recommend finding a way to write out 3x3 cells on 3 lines in your code, so they form an identifiable digits. Even if your code actually doesn't represent them like that internally. I'd much rather read

" " +

"|\_|" +

" |"

than

" |\_| |"

anyday.

When Christophe and Emmanuel presented this Kata at XP2005 they worked on a solution that made extensive use of recursion rather than iteration. Many people are more comfortable with iteration than recursion. Try this kata both ways.

Some gotchas to avoid:

- be very careful to read the definition of checksum correctly. It is not a simple dot product, the digits are reversed from what you expect.

- The spec does not list all the possible alternatives for valid digits when one pipe or underscore has been removed or added

- don't forget to try to work out what a ? should have been by adding or removing one pipe or underscore.

**Suggested Test Cases**

If you want to just copy and paste these test cases into your editor, I suggest first clicking "edit this page" so you can see the source. Then you can be sure to copy across all the whitespace necessary. Just don't save any changes by mistake.

use case 1

\_ \_ \_ \_ \_ \_ \_ \_ \_

| || || || || || || || || |

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 000000000

| | | | | | | | |

| | | | | | | | |

=> 111111111

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

=> 222222222

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 333333333

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

| | | | | | | | |

=> 444444444

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 555555555

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 666666666

\_ \_ \_ \_ \_ \_ \_ \_ \_

| | | | | | | | |

| | | | | | | | |

=> 777777777

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 888888888

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 999999999

\_ \_ \_ \_ \_ \_ \_

| \_| \_||\_||\_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_|

=> 123456789

use case 3

\_ \_ \_ \_ \_ \_ \_ \_

| || || || || || || ||\_ |

|\_||\_||\_||\_||\_||\_||\_| \_| |

=> 000000051

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| || ||\_ | | | \_

| \_||\_||\_||\_| | | | \_|

=> 49006771? ILL

\_ \_ \_ \_ \_ \_ \_

| \_| \_||\_| \_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_

=> 1234?678? ILL

use case 4

| | | | | | | | |

| | | | | | | | |

=> 711111111

\_ \_ \_ \_ \_ \_ \_ \_ \_

| | | | | | | | |

| | | | | | | | |

=> 777777177

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_|| || || || || || || || |

|\_ |\_||\_||\_||\_||\_||\_||\_||\_|

=> 200800000

\_ \_ \_ \_ \_ \_ \_ \_ \_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 333393333

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 888888888 AMB ['888886888', '888888880', '888888988']

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 555555555 AMB ['555655555', '559555555']

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_ |\_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

=> 666666666 AMB ['666566666', '686666666']

\_ \_ \_ \_ \_ \_ \_ \_ \_

|\_||\_||\_||\_||\_||\_||\_||\_||\_|

\_| \_| \_| \_| \_| \_| \_| \_| \_|

=> 999999999 AMB ['899999999', '993999999', '999959999']

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| || ||\_ | | ||\_

| \_||\_||\_||\_| | | | \_|

=> 490067715 AMB ['490067115', '490067719', '490867715']

\_ \_ \_ \_ \_ \_ \_

\_| \_| \_||\_||\_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_|

=> 123456789

\_ \_ \_ \_ \_ \_ \_

| || || || || || || ||\_ |

|\_||\_||\_||\_||\_||\_||\_| \_| |

=> 000000051

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| ||\_||\_ | | | \_

| \_||\_||\_||\_| | | | \_|

=> 490867715

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| || ||\_ | | ||\_

| \_||\_||\_||\_| | | | \_|

=> 490067715 AMB ['490067115', '490067719', '490867715']

\_ \_ \_ \_ \_ \_ \_

\_| \_| \_||\_||\_ |\_ ||\_||\_|

||\_ \_| | \_||\_| ||\_| \_|

=> 123456789

\_ \_ \_ \_ \_ \_ \_

| || || || || || || ||\_ |

|\_||\_||\_||\_||\_||\_||\_| \_| |

=> 000000051

\_ \_ \_ \_ \_ \_ \_

|\_||\_|| ||\_||\_ | | | \_

| \_||\_||\_||\_| | | | \_|

=> 490867715

# KataFizzBuzz

**Problem Description**

Imagine the scene. You are eleven years old, and in the five minutes before the end of the lesson, your Maths teacher decides he should make his class more "fun" by introducing a "game". He explains that he is going to point at each pupil in turn and ask them to say the next number in sequence, starting from one. The "fun" part is that if the number is divisible by three, you instead say "Fizz" and if it is divisible by five you say "Buzz". So now your maths teacher is pointing at all of your classmates in turn, and they happily shout "one!", "two!", "Fizz!", "four!", "Buzz!"... until he very deliberately points at you, fixing you with a steely gaze... time stands still, your mouth dries up, your palms become sweatier and sweatier until you finally manage to croak "Fizz!". Doom is avoided, and the pointing finger moves on.

So of course in order to avoid embarassment infront of your whole class, you have to get the full list printed out so you know what to say. Your class has about 33 pupils and he might go round three times before the bell rings for breaktime. Next maths lesson is on Thursday. Get coding!

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz[?](http://codingdojo.org/cgi-bin/index.pl?action=edit&id=FizzBuzz)".

Sample output:

1

2

Fizz

4

Buzz

Fizz

7

8

Fizz

Buzz

11

Fizz

13

14

FizzBuzz

16

17

Fizz

19

Buzz

... etc up to 100

**Stage 2 - new requirements**

\* A number is fizz if it is divisible by 3 or if it has a 3 in it

\* A number is buzz if it is divisible by 5 or if it has a 5 in it

# KataPotter

**Problem Description**

Once upon a time there was a series of 5 books about a very English hero called Harry. (At least when this Kata was invented, there were only 5. Since then they have multiplied) Children all over the world thought he was fantastic, and, of course, so did the publisher. So in a gesture of immense generosity to mankind, (and to increase sales) they set up the following pricing model to take advantage of Harry's magical powers.

One copy of any of the five books costs 8 EUR. If, however, you buy two different books from the series, you get a 5% discount on those two books. If you buy 3 different books, you get a 10% discount. With 4 different books, you get a 20% discount. If you go the whole hog, and buy all 5, you get a huge 25% discount.

Note that if you buy, say, four books, of which 3 are different titles, you get a 10% discount on the 3 that form part of a set, but the fourth book still costs 8 EUR.

Potter mania is sweeping the country and parents of teenagers everywhere are queueing up with shopping baskets overflowing with Potter books. Your mission is to write a piece of code to calculate the price of any conceivable shopping basket, giving as big a discount as possible.

For example, how much does this basket of books cost?

2 copies of the first book

2 copies of the second book

2 copies of the third book

1 copy of the fourth book

1 copy of the fifth book

(answer: 51.20 EUR)

**Clues**

You’ll find that this Kata is easy at the start. You can get going with tests for baskets of 0 books, 1 book, 2 identical books, 2 different books… and it is not too difficult to work in small steps and gradually introduce complexity.

However, the twist becomes apparent when you sit down and work out how much you think the sample basket above should cost. It isn’t 5\*8\*0.75+3\*8\*0.90. It is in fact 4\*8\*0.8+4\*8\*0.8. So the trick with this Kata is not that the acceptance test you’ve been given is wrong. The trick is that you have to write some code that is intelligent enough to notice that two sets of four books is cheaper than a set of five and a set of three.

You will have to introduce a certain amount of clever optimization algorithm. But not too much! This problem does not require a fully fledged general purpose optimizer. Try to solve just this problem well in order to share it for everyone or even in the ??? . Trust that you can generalize and improve your solution if and when new requirements come along.

* This application has nice application for

**Suggested Test Cases**

(Originally posted at xp-france.net/cgi-bin/wiki.pl?[KataPotter](http://codingdojo.org/cgi-bin/index.pl?KataPotter))

def testBasics

assert\_equal(0, price([]))

assert\_equal(8, price([0]))

assert\_equal(8, price([1]))

assert\_equal(8, price([2]))

assert\_equal(8, price([3]))

assert\_equal(8, price([4]))

assert\_equal(8 \* 2, price([0, 0]))

assert\_equal(8 \* 3, price([1, 1, 1]))

end

def testSimpleDiscounts

assert\_equal(8 \* 2 \* 0.95, price([0, 1]))

assert\_equal(8 \* 3 \* 0.9, price([0, 2, 4]))

assert\_equal(8 \* 4 \* 0.8, price([0, 1, 2, 4]))

assert\_equal(8 \* 5 \* 0.75, price([0, 1, 2, 3, 4]))

end

def testSeveralDiscounts

assert\_equal(8 + (8 \* 2 \* 0.95), price([0, 0, 1]))

assert\_equal(2 \* (8 \* 2 \* 0.95), price([0, 0, 1, 1]))

assert\_equal((8 \* 4 \* 0.8) + (8 \* 2 \* 0.95), price([0, 0, 1, 2, 2, 3]))

assert\_equal(8 + (8 \* 5 \* 0.75), price([0, 1, 1, 2, 3, 4]))

end

def testEdgeCases

assert\_equal(2 \* (8 \* 4 \* 0.8), price([0, 0, 1, 1, 2, 2, 3, 4]))

assert\_equal(3 \* (8 \* 5 \* 0.75) + 2 \* (8 \* 4 \* 0.8),

price([0, 0, 0, 0, 0,

1, 1, 1, 1, 1,

2, 2, 2, 2,

3, 3, 3, 3, 3,

4, 4, 4, 4]))

end

# KataRomanNumerals

**Problem Description**

The Romans were a clever bunch. They conquered most of Europe and ruled it for hundreds of years. They invented concrete and straight roads and even bikinis[[1]](http://sights.seindal.dk/sight/456_Roman_Villa_of_Piazza_Armerina.html). One thing they never discovered though was the number zero. This made writing and dating extensive histories of their exploits slightly more challenging, but the system of numbers they came up with is still in use today. For example the BBC uses Roman numerals to date their programmes.

The Romans wrote numbers using letters - I, V, X, L, C, D, M. (notice these letters have lots of straight lines and are hence easy to hack into stone tablets)

The Kata says you should write a function to convert from normal numbers to Roman Numerals: eg

1 --> I

10 --> X

7 --> VII

etc.

For a full description of how it works, take a look at [[this useful reference website]](http://www.novaroma.org/via_romana/numbers.html): which includes an implementation of the Kata in javascript.

There is no need to be able to convert numbers larger than about 3000. (The Romans themselves didn't tend to go any higher)

Note that you can't write numerals like "IM" for 999. Wikipedia says: *Modern Roman numerals ... are written by expressing each digit separately starting with the left most digit and skipping any digit with a value of zero. To see this in practice, consider the ... example of 1990. In Roman numerals 1990 is rendered: 1000=M, 900=CM, 90=XC; resulting in MCMXC. 2008 is written as 2000=MM, 8=VIII; or MMVIII.*

Part II of the Kata

* Write a function to convert in the other direction, ie numeral to digit

**Clues**

* can you make the code really beautiful and highly readable?

\* does it help to break out lots of small named functions from the main function, or is it better to keep it all in one function?

* if you don't know an algorithm to do this already, can you derive one using strict TDD?

\* does the order you take the tests in affect the final design of your algorithm?

\* Would it be better to work out an algorithm first before starting with TDD?

* if you do know an algorithm already, can you implement it using strict TDD?

\* Can you think of another algorithm?

* what are the best data structures for storing all the numeral letters? (I, V, D, M etc)
* can you define the test cases in a csv file and use FIT, or generate test cases in xUnit?
* what is the best way to verify your tests are correct?

**Suggested Test Cases**

Exercise left to the reader. You could use 1999 as an acceptance test.

# KataRomanCalculator

**Problem Description**

"As a Roman Bookkeeper I want to add Roman numbers because doing it manually is too tedious." Given the Roman numerals, (IVXLCDM which means one, five, ten, fifty, hundred, fivehundred and a thousand respectively), create two numbers and add them. As we are in Rome there is no such thing as decimals or int, we need to do this with the strings. An example would be "XIV" + "LX" = "LXXIV"

There are some rules to a Roman number:

* Numerals can be concatenated to form a larger numeral ("XX" + "II" = "XXII")
* If a lesser numeral is put before a bigger it means subtraction of the lesser from the bigger ("IV" means four, "CM" means ninehundred)
* If the numeral is I, X or C you can't have more than three ("II" + "II" = "IV")
* If the numeral is V, L or D you can't have more than one ("D" + "D" = "M")

**Clues**

String grouping and concatenation is key to solving this kata. But remember the rule that lesser numerals can preceede bigger ones.

**Suggested Test Cases**

The first test case(s) are pretty easy. I won't put the test cases here since I feel this kata is one of the better ones I have done when it comes to practicing finding out what the next test case is.

**Comments from those who are working on this Kata**

When I invented it I did not know how well it would work. I was pleasantly surprised to see it work out to be a kata very focused on finding test cases. -- [ThomasNilsson](http://codingdojo.org/cgi-bin/index.pl?ThomasNilsson)

This kata does not immediately show the "behaviour" aspect as does e.g. the Bowling kata. By this I mean the behaviour of a class under test, as opposed to testdriving a single method. However, in the Linköping dojo, we decided on adding this aspect by changing the story to "As a Polish born Roman Accountant..." and thereby introducing the Polish reverse notation to the calculation. The first test would then become:

public void testOnePlusOneShouldEqualTwo() {

RomanPolishCalculator calculator = new RomanPolishCalculator();

calculator.enter("I");

calculator.enter("I");

assertEquals("II", calculator.add());

}

And we also decided to add this aspect on an already running partial solution in the middle of the kata. This creates the situation of refactoring the API into another structure. We'll see how this turns out

# [KataNumbersInWords](http://codingdojo.org/cgi-bin/index.pl?back=KataNumbersInWords)

## Problem Description

It occurs now and then in real life that people want to write about money, especially about a certain amount of money. If it comes to cheques or contracts for example some nations have laws that state that you should write out the amount in words additionally to the amount in numbers to avoid fraud and mistakes. So if you want to transfer 745 $ to someone via cheque you have to fill out to fields:

745.00 $ (amount in numbers)

seven hundred and fourty five dollars (amount in words)

### Step 1

The Kata is now to write a little converter class or function (depends on your favourite language and flavour) to convert numbers into words.

### Step 2

Convert it back.

### Step 3

Do all of it test driven.

# [KataArgs](http://codingdojo.org/cgi-bin/index.pl?back=KataArgs)

**Problem Description**

Most of us have had to parse command-line arguments from time to time. If we don't have a convenient utility, then we simply walk the array of strings that is passed into the *main* function. There are several good utilities available from various sources, but they probably don't do exactly what we want. So let's write another one!

The arguments passed to the program consist of flags and values. Flags should be one character, preceded by a minus sign. Each flag should have zero, or one value associated with it.

You should write a parser for this kind of arguments. This parser takes a schema detailing what arguments the program expects. The schema specifies the number and types of flags and values the program expects.

Once the schema has been specified, the program should pass the actual argument list to the args parser. It will verify that the arguments match the schema. The program can then ask the args parser for each of the values, using the names of the flags. The values are returned with the correct types, as specified in the schema.

For example if the program is to be called with these arguments:

-l -p 8080 -d /usr/logs

this indicates a schema with 3 flags: l, p, d. The "l" (logging) flag has no values associated with it, it is a boolean flag, True if present, False if not. the "p" (port) flag has an integer value, and the "d" (directory) flag has a string value.

If a flag mentioned in the schema is missing in the arguments, a suitable default value should be returned. For example "False" for a boolean, 0 for a number, and "" for a string. If the arguments given do not match the schema, it is important that a good error message is given, explaining exactly what is wrong.

If you are feeling ambitious, extend your code to support lists eg

-g this,is,a,list -d 1,2,-3,5

So the "g" flag indicates a list of strings, ["this", "is", "a", "list"] and the "d" flag indicates a list of integers, [1, 2, -3, 5].

Make sure your code is extensible, in that it is straightforward and obvious how to add new types of values.

**Clues**

What the schema should look like and how to specify it is deliberately left vague in the Kata description. An important part of the Kata is to design a concise yet readable format for it.

**Suggested Test Cases**

* make sure you have a test with a negative integer (confusing - sign)
* the order of the arguments need not match the order given in the schema.
* have some tests that suitable default values are correctly assigned if flags given in the schema are missing in the args given.

# [KataAnagram](http://codingdojo.org/cgi-bin/index.pl?back=KataAnagram)

**About this Kata**

This Kata was posted as the problem to be solved in a "self-documenting code contest": selfexplanatorycode.blogspot.com/2008/08/announcing-self-documenting-code.html

**Problem Description**

Write a program that generates all two-word anagrams of the string "documenting". Here's a word list you might want to use: pragdave.pragprog.com/data/wordlist.txt.

**Clues**

You should think about the performance of your solution and whether making it faster affects readability.

**Suggested Test Cases**

none as yet.

**Comments from those who are working on this Kata**

The results of the "self documenting code contest" are at selfexplanatorycode.blogspot.com/2008/09/results.html

# [KataDepthFirstSearch](http://codingdojo.org/cgi-bin/index.pl?back=KataDepthFirstSearch)

**Problem Description**

Depth-first search is not just an AI technique it is also the standard way that trees (ubiquitous in computer science) are traversed.

**Clues**

For this (easiest) form of depth-first search, steer towards using the function-call stack, rather than a stack data structure.

Try to avoid implementing a Graph or Maze or Problem class (even as a mocked interface) for your depth-first-searcher to traverse. Instead, ask questions of a "human" user (e.g. standard in/standard out or whatever your language uses). For example, "What are the exits?" and "Where are we?". This allows exploratory testing, which is fun. Having to build a Graph or Maze by initializing data, or parsing a file, slows down the group and is distant from the main thrust of the kata.

Of course, you should mock the conversation in your unit tests! This demonstrates that exploratory testing can be recorded in repeatable automated tests, which is valuable.

**Suggested Test Cases**

The one-node graph. The two-node graph. A 2x2 maze. A full two-level binary tree. A 3x3 maze.

# [KataNumberToLCD](http://codingdojo.org/cgi-bin/index.pl?back=KataNumberToLCD)

I saw a screen cast that Corey Haines did on this at <http://vodpod.com/watch/2353528-number-to-lcd>

The code for his solution is at <http://github.com/coreyhaines/kata-number-to-led>

The basic idea is the inverse of [KataBankOCR](http://codingdojo.org/cgi-bin/index.pl?KataBankOCR)

# [KataBowling](http://codingdojo.org/cgi-bin/index.pl?back=KataBowling)

This description is based on that at www.xprogramming.com/xpmag/acsBowling.htm

Problem Description

Create a program, which, given a valid sequence of rolls for one line of American Ten-Pin Bowling, produces the total score for the game. Here are some things that the program will not do:

* We will not check for valid rolls.
* We will not check for correct number of rolls and frames.
* We will not provide scores for intermediate frames.

Depending on the application, this might or might not be a valid way to define a complete story, but we do it here for purposes of keeping the kata light. I think you'll see that improvements like those above would go in readily if they were needed for real.

We can briefly summarize the scoring for this form of bowling:

* Each game, or "line" of bowling, includes ten turns, or "frames" for the bowler.
* In each frame, the bowler gets up to two tries to knock down all the pins.
* If in two tries, he fails to knock them all down, his score for that frame is the total number of pins knocked down in his two tries.
* If in two tries he knocks them all down, this is called a "spare" and his score for the frame is ten plus the number of pins knocked down on his next throw (in his next turn).
* If on his first try in the frame he knocks down all the pins, this is called a "strike". His turn is over, and his score for the frame is ten plus the simple total of the pins knocked down in his next two rolls.
* If he gets a spare or strike in the last (tenth) frame, the bowler gets to throw one or two more bonus balls, respectively. These bonus throws are taken as part of the same turn. If the bonus throws knock down all the pins, the process does not repeat: the bonus throws are only used to calculate the score of the final frame.
* The game score is the total of all frame scores.

More info on the rules at: www.topendsports.com/sport/tenpin/scoring.htm

Clues

What makes this game interesting to score is the lookahead in the scoring for strike and spare. At the time we throw a strike or spare, we cannot calculate the frame score: we have to wait one or two frames to find out what the bonus is.

Suggested Test Cases

(When scoring "X" indicates a strike, "/" indicates a spare, "-" indicates a miss)

* "XXXXXXXXXXXX" (12 rolls: 12 strikes) = 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 + 10+10+10 = 300
* "9-9-9-9-9-9-9-9-9-9-" (20 rolls: 10 pairs of 9 and miss) = 9 + 9 + 9 + 9 + 9 + 9 + 9 + 9 + 9 + 9 = 90
* "5/5/5/5/5/5/5/5/5/5/5" (21 rolls: 10 pairs of 5 and spare, with a final 5) = 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 + 10+5 = 150

Comments from those who have mastered this Kata

Write some thoughts here about what you have learnt from this Kata. You don't have to post all the code of your solution - I think the solution in itself is less interesting than the path you took to get there and what decisions you made. Just seeing the code won't necessarily help me to reproduce it for myself. So in this section various people might go through the main parts of the problem and how they tackled them, what design ideas were discarded, and which order the test cases were implemented in.

* One interesting point to note is that without counting frames in any way (although I don't think this was intended as a 'hard' requirement for the initial Kata completion), finding an elegant way to identify the end of the game/last "real" frame becomes difficult (ie: assuming there are final 'bonus' rolls included in a given test case). **Update**: After trying various things, including writing out a logic matrix for possible end-of-game combinations, I'm not sure it's possible to detect whether a final 'throw' counts as bonus-only or as part of an actual frame, unless you're counting frames. -- [RudyXDesjardins](http://codingdojo.org/cgi-bin/index.pl?RudyXDesjardins)

# [KataGameOfLife](http://codingdojo.org/cgi-bin/index.pl?back=KataGameOfLife)

Similar Katas - [KataMinesweeper](http://codingdojo.org/cgi-bin/index.pl?KataMinesweeper), [KataReversi](http://codingdojo.org/cgi-bin/index.pl?KataReversi)

**Problem Description**

This Kata is about calculating the next generation of Conway's game of life, given any starting position. See <http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life> for background.

You start with a two dimensional grid of cells, where each cell is either alive or dead. In this version of the problem, the grid is finite, and no life can exist off the edges. When calcuating the next generation of the grid, follow these rules:

1. Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.

2. Any live cell with more than three live neighbours dies, as if by overcrowding.

3. Any live cell with two or three live neighbours lives on to the next generation.

4. Any dead cell with exactly three live neighbours becomes a live cell.

You should write a program that can accept an arbitrary grid of cells, and will output a similar grid showing the next generation.

**Clues**

The input starting position could be a text file that looks like this:

Generation 1:

4 8

........

....\*...

...\*\*...

........

And the output could look like this:

Generation 2:

4 8

........

...\*\*...

...\*\*...

........

The input format is similar to that in [KataMinesweeper](http://codingdojo.org/cgi-bin/index.pl?KataMinesweeper), and is easily parsed in most languages.

**Suggested Test Cases** Make sure you have enough coverage of edge cases - where there are births and deaths at the edge of the grid.

# [KataMinesweeper](http://codingdojo.org/cgi-bin/index.pl?back=KataMinesweeper)

**Problem Description**

Have you ever played Minesweeper? It's a cute little game which comes within a certain Operating System whose name we can't really remember. Well, the goal of the game is to find all the mines within an MxN field. To help you, the game shows a number in a square which tells you how many mines there are adjacent to that square. For instance, take the following 4x4 field with 2 mines (which are represented by an \* character):

\*...

....

.\*..

....

The same field including the hint numbers described above would look like this:

\*100

2210

1\*10

1110

You should write a program that takes input as follows:

The input will consist of an arbitrary number of fields. The first line of each field contains two integers n and m (0 < n,m <= 100) which stands for the number of lines and columns of the field respectively. The next n lines contains exactly m characters and represent the field. Each safe square is represented by an "." character (without the quotes) and each mine square is represented by an "\*" character (also without the quotes). The first field line where n = m = 0 represents the end of input and should not be processed.

Your program should produce output as follows:

For each field, you must print the following message in a line alone:

Field #x:

Where x stands for the number of the field (starting from 1). The next n lines should contain the field with the "." characters replaced by the number of adjacent mines to that square. There must be an empty line between field outputs.

**Clues**

As you may have already noticed, each square may have at most 8 adjacent squares.

**Suggested Test Cases**

This is the acceptance test input:

4 4

\*...

....

.\*..

....

3 5

\*\*...

.....

.\*...

0 0

and output:

Field #1:

\*100

2210

1\*10

1110

Field #2:

\*\*100

33200

1\*100

# [KataPacMan](http://codingdojo.org/cgi-bin/index.pl?back=KataPacMan)

**Problem Description**

Pacman finds himself in a grid filled with monsters. Will he be able to eat all the dots on the board before the monsters eat him?

Incomplete list of things the game needs:

\* pacman is on a grid filled with dots

\* pacman has a direction

\* pacman moves on each tick

\* user can rotate pacman

\* pacman eats dots

\* pacman wraps around

\* pacman stops on wall

\* pacman will not rotate into a wall

\* game score (levels completed, number of dots eaten in this level)

\* monsters...

\* levels

\* animate pacman eating (mouth opens and closes)

**Clues**

You probably won't be able to complete all of the list in one night of dojo, however having the list (or starting with part of it and letting the participants brainstorm) makes for good design discussions. As in the game of life, a board representation does not have to be difficult. E.g. pacman starts in the centre of the board and is looking up (notice that pacman eats, so the V points downward because pacman has his mouth open):

. . .

.V.

. . .

Pacman looks continuous, however the game state changes in discrete steps. Creating a tick() method/function or somesuch, or passing a board to a function which returns a 'next state' board makes it easy to test the various conditions.

**Suggested Test Cases**

see the pacman project on github.com (mostalive/pacman) for actual steps from the first run.

# [KataPokerHands](http://codingdojo.org/cgi-bin/index.pl?back=KataPokerHands)

**About this Kata**

This kata is blatantly stolen from acm.uva.es/p/v103/10315.html It is a subset of [KataTexasHoldEm](http://codingdojo.org/cgi-bin/index.pl?KataTexasHoldEm), which is a very large Kata.

**Problem Description**

Your job is to compare several pairs of poker hands and to indicate which, if either, has a higher rank.

**Poker rules description**

A poker deck contains 52 cards - each card has a suit which is one of clubs, diamonds, hearts, or spades (denoted C, D, H, and S in the input data). Each card also has a value which is one of 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king, ace (denoted 2, 3, 4, 5, 6, 7, 8, 9, T, J, Q, K, A). For scoring purposes, the suits are unordered while the values are ordered as given above, with 2 being the lowest and ace the highest value.

A poker hand consists of 5 cards dealt from the deck. Poker hands are ranked by the following partial order from lowest to highest.

* High Card: Hands which do not fit any higher category are ranked by the value of their highest card. If the highest cards have the same value, the hands are ranked by the next highest, and so on.
* Pair: 2 of the 5 cards in the hand have the same value. Hands which both contain a pair are ranked by the value of the cards forming the pair. If these values are the same, the hands are ranked by the values of the cards not forming the pair, in decreasing order.
* Two Pairs: The hand contains 2 different pairs. Hands which both contain 2 pairs are ranked by the value of their highest pair. Hands with the same highest pair are ranked by the value of their other pair. If these values are the same the hands are ranked by the value of the remaining card.
* Three of a Kind: Three of the cards in the hand have the same value. Hands which both contain three of a kind are ranked by the value of the 3 cards.
* Straight: Hand contains 5 cards with consecutive values. Hands which both contain a straight are ranked by their highest card.
* Flush: Hand contains 5 cards of the same suit. Hands which are both flushes are ranked using the rules for High Card.
* Full House: 3 cards of the same value, with the remaining 2 cards forming a pair. Ranked by the value of the 3 cards.
* Four of a kind: 4 cards with the same value. Ranked by the value of the 4 cards.
* Straight flush: 5 cards of the same suit with consecutive values. Ranked by the highest card in the hand.

**Clues**

None as yet.

**Suggested Test Cases**

Sample input:

Black: 2H 3D 5S 9C KD White: 2C 3H 4S 8C AH

Black: 2H 4S 4C 2D 4H White: 2S 8S AS QS 3S

Black: 2H 3D 5S 9C KD White: 2C 3H 4S 8C KH

Black: 2H 3D 5S 9C KD White: 2D 3H 5C 9S KH

Each row of input is a game with two players. The first five cards belong to the player named "Black" and the second five cards belong to the player named "White".

Sample output:

White wins. - with high card: Ace

White wins. - with flash

Black wins. - with high card: 9

Tie.

**Comments from those who are working on this Kata**

I found a reference to this Kata on [IvanSanchez](http://codingdojo.org/cgi-bin/index.pl?IvanSanchez) blog: isanchez.net/2009/04/19/ruby-coding-dojo-this-week/

The 3d rows should result in Black winning, not White since 2H 3D 5S 9C KD is higher than 2C 3H 4S 8C KH

# [KataReversi](http://codingdojo.org/cgi-bin/index.pl?back=KataReversi)

**Problem Description**

Reversi is a board game for two players. More information can be found on Wikipedia en.wikipedia.org/wiki/Reversi[?](http://codingdojo.org/cgi-bin/index.pl?action=edit&id=KataReversi/Reversi). This Kata is to write a program that takes a current board position together with information about whose turn it is, and returns a list of the legal moves for that player. A move is only legal if it results in at least one of the opponent's counters being flipped.

**Suggested Test Cases**

........

........

........

...BW...

...WB...

........

........

........

B

(A "." indicates an empty square. A "B" indicates a black piece and a "W" represents a white piece. The trailing "B" indicates that it is black's turn)

You could either output the possible moves as co-ordinates (columns labelled A - H, rows labelled 1 - 8 starting from top left hand corner) like this: [C5, D6, E3, F4]

or graphically like this:

........

........

....0...

...BW0..

..0WB...

...0....

........

........

B

**Comments from those who are working on this Kata**

Please try this Kata and comment! We could probably do with some more test cases too.

I like this Kata, it looks nicer when there are spaces between the columns:

. . . . . . . .

. . . . . . . .

. . . . . . . .

. . . B W . . .

. . . W B . . .

. . . . . . . .

. . . . . . . .

. . . . . . . .

B

# [KataTennis](http://codingdojo.org/cgi-bin/index.pl?back=KataTennis)

**About this Kata**

This Kata is about implementing a simple tennis game. I came up with it while thinking about Wii tennis, where they have simplified tennis, so each set is one game.

The scoring system is rather simple:

1. Each player can have either of these points in one game 0 15 30 40

2. If you have 40 and you win the ball you win the game, however there are special rules.

3. If both have 40 the players are deuce. a. If the game is in deuce, the winner of a ball will have advantage and game ball. b. If the player with advantage wins the ball he wins the game c. If the player without advantage wins they are back at deuce.

===== Alternate description of the rules per Wikipedia (<http://en.wikipedia.org/wiki/Tennis#Scoring>):

1. A game is won by the first player to have won at least four points in total and at least two points more than the opponent.

2. The running score of each game is described in a manner peculiar to tennis: scores from zero to three points are described as "love", "fifteen", "thirty", and "forty" respectively.

3. If at least three points have been scored by each player, and the scores are equal, the score is "deuce".

4. If at least three points have been scored by each side and a player has one more point than his opponent, the score of the game is "advantage" for the player in the lead.

**Example solutions**

* <http://github.com/follesoe/TennisKataJava> Example solution in Java from Trondheim Coding Dojo
* <http://bitbucket.org/alf.lervag/tenniskata> Example solution in .NET from Trondheim Coding Dojo
* <http://github.com/goeran/Katas/tree/master/Tennis/csharp/2ndTry/> Example solution in .NET
* <https://github.com/lroal/Roald/tree/master/src/Roald.Katas> Example solution in .NET with state transition tree

# [KataTexasHoldEm](http://codingdojo.org/cgi-bin/index.pl?back=KataTexasHoldEm)

If you want to try this Kata for yourself or at your dojo meeting, read the problem description and see if the Kata appeals to you. The rest is commentary and helpful clues for if you get stuck solving it. I would recommend trying the Kata for yourself before reading too much of it. For an easier kata, try [KataPokerHands](http://codingdojo.org/cgi-bin/index.pl?KataPokerHands) first.

Difficulty: hard

**Problem description**

You work for a cable network; specifically, you are the resident hacker for a Texas Hold'Em Championship show.

The show's producer has come to you for a favor. It seems the play-by-play announcers just can't think very fast. All beauty, no brains. The announcers could certainly flap their jaws well enough, if they just knew what hands the players were holding and which hand won the round. Since this is live TV, they need those answers quick. Time to step up to the plate. Bob, the producer, explains what you need to do.

BOB: Each player's cards for the round will be on a separate line of the input. Each card is a pair of characters, the first character represents the face, the second is the suit. Cards are separated by exactly one space. Here's a sample hand.

Kc 9s Ks Kd 9d 3c 6d

9c Ah Ks Kd 9d 3c 6d

Ac Qc Ks Kd 9d 3c

9h 5s

4d 2d Ks Kd 9d 3c 6d

7s Ts Ks Kd 9d

YOU: Okay, I was going ask what character to use for 10, but I guess 'T' is it. And 'c', 'd', 'h' and 's' for the suits, makes sense. Why aren't seven cards listed for every player?

BOB: Well, if a player folds, only his hole cards and the community cards he's seen so far are shown.

YOU: Right. And why did the fifth player play with a 4 and 2? They're suited, but geez, talk about risk...

BOB: Stay on topic. Now, the end result of your code should generate output that looks like this:

Kc 9s Ks Kd 9d 3c 6d Full House (winner)

9c Ah Ks Kd 9d 3c 6d Two Pair

Ac Qc Ks Kd 9d 3c

9h 5s

4d 2d Ks Kd 9d 3c 6d Flush

7s Ts Ks Kd 9d

YOU: Okay, so I repeat the cards, list the rank or nothing if the player folded, and the word "winner" in parenthesis next to the winning hand. Do you want the cards rearranged at all?

BOB: Hmmm... we can get by without it, but if you have the time, do it. Don't bother for folded hands, but for ranked hands, move the cards used to the front of the line, sorted by face. Kickers follow that, and the two unused cards go at the end, just before the rank is listed.

YOU: Sounds good. One other thing, I need to brush up on the hand ranks. You have any good references for Texas Hold'Em?

BOB: Yeah, do an internet search on Poker Hand Rankings. And if you need it, the Rules of Texas Hold'Em. While ranking, don't forget the kicker, the next highest card in their hand if player's are tied. And of course, if -- even after the kicker -- player's are still tied, put "(winner)" on each appropriate line of output.

YOU: Ok. I still don't understand one thing...

BOB: What's that?

YOU: Why he stayed in with only the 4 and 2 of diamonds? That's just...

BOB: Hey! Show's on in ten minutes! Get to work!

**Clues**

There are two dimensions to this problem. Firstly how to rank a particular hand (“Flush” or “Two Pairs” etc) and secondly how to compare hands and determine which will win. A very easy way to get going with this Kata is just to concentrate on the first dimension, and write lots of code that can successfully assigns a rank to all the different kinds of hand. This is all well and good, but I think you’ll find some refactoring in order when you come to tackle the second dimension. My recommendation is to write just enough code to identify one or two ranks, then start working on comparing hands. You can fill in the details of all the different ranks when you have a basic structure that can both rank and compare hands.

One interesting aspect of this Kata is the OO design that you come up with. What classes do you need? What responsibilities do they have? Have you removed all duplication?

Another interesting aspect which you may or may not want to go into, is how you express the rules of Poker. These rules are both repetitive and arbitrary and in many ways resemble the kinds of business rules you meet in the real world. Can you find a way of expressing them that will leave an expert poker player able to verify that they are implemented correctly, even if they are not themselves a programmer?

# [KataTradingCardGame](http://codingdojo.org/cgi-bin/index.pl?back=KataTradingCardGame)

In this Kata you will be implementing a rudimentary two-player trading card game. The rules are loosely based on Blizzard Hearthstone (<http://us.battle.net/hearthstone/en/>) which itself is an already much simpler and straight-forward game compared to other TCGs like *Magic: The Gathering*(<http://www.wizards.com/magic/>).

Difficulty: Medium/Hard[?](http://codingdojo.org/cgi-bin/index.pl?action=edit&id=KataTradingCardGame/Hard)

Good for teaching: Test Doubles, Mocking, TDD, Refactoring, Clean Code

Problem Description

1. Preparation

* Each player starts the game with 30 Health and 0 Mana slots
* Each player starts with a deck of 20 Damage cards with the following Mana costs: 0,0,1,1,2,2,2,3,3,3,3,4,4,4,5,5,6,6,7,8
* From the deck each player receives 3 random cards has his initial hand

2. Gameplay

* The active player receives 1 Mana slot up to a maximum of 10 total slots
* The active player's empty Mana slots are refilled
* The active player draws a random card from his deck
* The active player can play as many cards as he can afford. Any played card empties Mana slots and deals immediate damage to the opponent player equal to its Mana cost.
* If the opponent player's Health drops to or below zero the active player wins the game
* If the active player can't (by either having no cards left in his hand or lacking sufficient Mana to pay for any hand card) or simply doesn't want to play another card, the opponent player becomes active

3. Special Rules

* Bleeding Out: If a player's card deck is empty before the game is over he receives 1 damage instead of drawing a card when it's his turn.
* Overload: If a player draws a card that lets his hand size become >5 that card is discarded instead of being put into his hand.
* Dud Card: The 0 Mana cards can be played for free but don't do any damage either. They are just annoyingly taking up space in your hand.

Clues

When approached iteratively with TDD you can take different starting points, like the player state or the game loop. It is your own choice whether you implement the game for human or computer players - or both. Game visualization can be anything between System.out and a GUI. You can also increase the difficulty by adding more rules, like Healing cards, Damage independent from Mana cost or introducing individual Deck building. You will find some examples of Advanced Variations from the Kata's author at <https://github.com/bkimminich/kata-tcg>. Even without extra rules the toughest part of this Kata might be coming up with actually smart CPU player decision-making algorithms.

Example Solution

* <https://github.com/bkimminich/kata-tcg> - Example solution in Java 8 from the Kata's author [BjoernKimminich](http://codingdojo.org/cgi-bin/index.pl?BjoernKimminich)

# [KataYahtzee](http://codingdojo.org/cgi-bin/index.pl?back=KataYahtzee)

**About this Kata**

This problem is based on Ruby Quiz #19 (<http://rubyquiz.com/quiz19.html>). The kata takes about 20 minutes with practice and 2 hours in a dojo setting.

**Problem Description**

The game of yahtzee is a simple dice game. Each round, each player rolls five six sided dice. The player may choose to reroll some or all of the dice up to three times (including the original roll). The player then places the roll at a category, such as ones, twos, sixes, pair, two pairs etc. If the roll is compatible with the score, the player gets a score for this roll according to the rules. If the roll is not compatible, the player gets a score of zero for this roll.

The kata consists of creating the rules to score a roll in any of a predefined category. Given a roll and a category, the final solution should output the score for this roll placed in this category.

**Yahtzee rules description and suggested test cases**

The following categories exists:

* Ones, Twos, Threes, Fours, Fives, Sixes: The player scores the sum of the dice that reads one, two, three, four, five or six, respectively. For example, 1, 1, 2, 4, 4 placed on "fours" gives 8 points.
* Pair: The player scores the sum of the two highest matching dice. For example, 3, 3, 3, 4, 4 placed on "pair" gives 8.
* Two pairs: If there are two pairs of dice with the same number, the player scores the sum of these dice. If not, the player scores 0. For example, 1, 1, 2, 3, 3 placed on "two pairs" gives 8.
* Three of a kind: If there are three dice with the same number, the player scores the sum of these dice. Otherwise, the player scores 0. For example, 3, 3, 3, 4, 5 places on "three of a kind" gives 9.
* Four of a kind: If there are four dice with the same number, the player scores the sum of these dice. Otherwise, the player scores 0. For example, 2, 2, 2, 2, 5 places on "four of a kind" gives 8.
* Small straight: If the dice read 1,2,3,4,5, the player scores 15 (the sum of all the dice), otherwise 0.
* Large straight: If the dice read 2,3,4,5,6, the player scores 20 (the sum of all the dice), otherwise 0.
* Full house: If the dice are two of a kind and three of a kind, the player scores the sum of all the dice. For example, 1,1,2,2,2 placed on "full house" gives 8. 4,4,4,4,4 is not "full house".
* Yahtzee: If all dice are the have the same number, the player scores 50 points, otherwise 0.
* Chance: The player gets the sum of all dice, no matter what they read.

The practitioner can feel free to create new categories as well.

# [KataRange](http://codingdojo.org/cgi-bin/index.pl?back=KataRange)

Range has a lot of nifty issues.

* integer range contains

[2,6) contains {2,4}

[2,6) doesn't contain {-1,1,6,10}

* getAllPoints[?](http://codingdojo.org/cgi-bin/index.pl?action=edit&id=AllPoints)

[2,6) allPoints = {2,3,4,5}

* ContainsRange[?](http://codingdojo.org/cgi-bin/index.pl?action=edit&id=ContainsRange)

[2,5) doesn't contain [7,10)

[2,5) doesn't contain [3,10)

[3,5) doesn't contain [2,10)

[2,10) contains [3,5]

[3,5] contains [3,5)

* endPoints

[2,6) allPoints = {2,3,4,5}

[2,6] allPoints = {2,3,4,5,6}

(2,6) allPoints = {3,4,5}

(2,6] allPoints = {3,4,5,6}

* overlapsRange

[2,5) doesn't overlap with [7,10)

[2,10) overlaps with [3,5)

[3,5) overlaps with [3,5)

[2,5) overlaps with [3,10)

[3,5) overlaps with [2,10)

* Equals

[3,5) equals [3,5)

[2,10) neq [3,5)

[2,5) neq [3,10)

[3,5) neq [2,10)

# [KataDictionaryReplacer](http://codingdojo.org/cgi-bin/index.pl?back=KataDictionaryReplacer)

This kata is about making a simple string replacer. It is inspired by Corey Haines Lightning talk about practicing. (aac2009.confreaks.com/06-feb-2009-20-30-lightning-talk-under-your-fingers-corey-haines.html)

Create a method that takes a string and a dictionary, and replaces every key in the dictionary pre and suffixed with a dollar sign, with the corresponding value from the Dictionary.

Tests

input : "", dict empty, output:""

input : "$temp$", dict ["temp", "temporary"], output: "temporary"

input : "$temp$ here comes the name $name$", dict ["temp", "temporary"] ["name", "John Doe"], output : "temporary here comes the name John Doe"

# Refactori,ng Legacy Code

* <https://github.com/lucaminudel/TDDwithMockObjectsAndDesignPrinciples/tree/master/TDDMicroExercises#readme>