1. Opposite number

Very simple, given a number, find its opposite.
Examples:
1: -1
14: -14
-34: 34

2. Basic Mathematical Operations

Your task is to create a function that does four basic mathematical operations. The function should take three arguments - operation(string/char), value1(number), value2(number).The function should return result of numbers after applying the chosen operation.
Examples

```
basicOp('+', 4, 7)         // Output: 11
basicOp('-', 15, 18)       // Output: -3
basicOp('*', 5, 5)         // Output: 25
basicOp('/', 49, 7)        // Output: 7
basicOp('+', 4, 7)         // Output: 11
basicOp('-', 15, 18)       // Output: -3
basicOp('*', 5, 5)         // Output: 25
basicOp('/', 49, 7)        // Output: 7
basicOp('+', 4, 7)         // Output: 11
basicOp('-', 15, 18)       // Output: -3
basicOp('*', 5, 5)         // Output: 25
basicOp('/', 49, 7)        // Output: 7
basicOp('+', 4, 7)         // Output: 11
basicOp('-', 15, 18)       // Output: -3
basicOp('*', 5, 5)         // Output: 25
basicOp('/', 49, 7)        // Output: 7
basicOp('+', 4, 7)         // Output: 11
basicOp('-', 15, 18)       // Output: -3
basicOp('*', 5, 5)         // Output: 25
basicOp('/', 49, 7)        // Output: 7
basicOp '+' 4 7            -- Output: 11
basicOp '-' 15 18          -- Output: -3
basicOp '*' 5 5            -- Output: 25
basicOp '/' 49 7           -- Output: 7

basicOp '/' 50 7           -- Output: 7 -- because integer division
basic_op('+', 4, 7)         # Output: 11
basic_op('-', 15, 18)       # Output: -3
basic_op('*', 5, 5)         # Output: 25
basic_op('/', 49, 7)        # Output: 7
basic_op('+', 4, 7)         # Output: 11
basic_op('-', 15, 18)       # Output: -3
basic_op('*', 5, 5)         # Output: 25
basic_op('/', 49, 7)        # Output: 7
mov dil, '+'
mov rax, __float64__(4.0)
mov rdx, __float64__(7.0)
movq xmm0, rax
movq xmm1, rdx
call basic_op        ; XMM0 <- 11.0

mov dil, '-'
mov rax, __float64__(15.0)
mov rdx, __float64__(18.0)
movq xmm0, rax
movq xmm1, rdx
call basic_op        ; XMM0 <- -3.0

mov dil, '*'
mov rax, __float64__(5.0)
movq xmm0, rax
movq xmm1, rax
call basic_op        ; XMM0 <- 25.0

mov dil, '/'
mov rax, __float64__(49.0)
mov rdx, __float64__(7.0)
movq xmm0, rax
movq xmm1, rdx
call basic_op        ; XMM0 <- 7.0
SimpleMath.basic_op("+", 4, 7)     # Output: 11
SimpleMath.basic_op("-", 15, 18)   # Output: -3
SimpleMath.basic_op("*", 5, 5)     # Output: 25
SimpleMath.basic_op("/", 49, 7)    # Output: 7
```

3. Printing Array elements with Comma delimiters

```
Input: Array of elements
["h","o","l","a"]
Output: String with comma delimited elements of the array in th same order.
"h,o,l,a"
```

4. Transportation on vacation

After a hard quarter in the office you decide to get some rest on a vacation. So you will book a flight for you and your girlfriend and try to leave all the mess behind you.

You will need a rental car in order for you to get around in your vacation. The manager of the car rental makes you some good offers.

Every day you rent the car costs $40. If you rent the car for 7 or more days, you get $50 off your total. Alternatively, if you rent the car for 3 or more days, you get $20 off your total.

Write a code that gives out the total amount for different days(d).

5. Get the Middle Character

You are going to be given a word. Your job is to return the middle character of the word. If the word's length is odd, return the middle character. If the word's length is even, return the middle 2 characters.

#Examples:

```
Kata.getMiddle("test") should return "es"

Kata.getMiddle("testing") should return "t"

Kata.getMiddle("middle") should return "dd"

Kata.getMiddle("A") should return "A"
```

#Input

A word (string) of length 0 < str < 1000 (In javascript you may get slightly more than 1000 in some test cases due to an error in the test cases). You do not need to test for this. This is only here to tell you that you do not need to worry about your solution timing out.

#Output

The middle character(s) of the word represented as a string.

6. Partition On

Write a function which partitions a list of items based on a given predicate.

After the partition function is run, the list should be of the form [F, F, F, T, T, T] where the Fs (resp. Ts) are items for which the predicate function returned false (resp. true).

NOTE: the partitioning should be stable; in other words: the ordering of the Fs (resp. Ts) should be preserved relative to each other.

For convenience and utility, the partition function should return the boundary index. In other words: the index of the first T value in items.

For example:

```
items = [1, 2, 3, 4, 5, 6]
isEven = (n) -> n % 2 == 0
i = partitionOn isEven, items
# items should now be [1, 3, 5, 2, 4, 6]
# i     should now be 3
var items = [1, 2, 3, 4, 5, 6];
function isEven(n) {return n % 2 == 0}
var i = partitionOn(isEven, items);
// items should now be [1, 3, 5, 2, 4, 6]
// i     should now be 3
bool is_even(const void *ptr) { return *((const int *) ptr) % 2 == 0; }
int items[] = {1, 2, 3, 4, 5, 6};
size_t i = partition_on(items, 6, sizeof(int), is_even);
// items should now be {1, 3, 5, 2, 4, 6}
// i     should not be 3
```

7. Word Count

```
Can you realize a function that returns word count from a given string?
You have to ensure that spaces in string is a whitespace for real.
What we want and finish of work:
countWords("Hello"); // returns 1 as int
countWords("Hello, World!") // returns 2
countWords("No results for search term `s`") // returns 6
countWords(" Hello") // returns 1
// ... and so on
count_words("Hello"); # returns 1 as int
count_words("Hello, World!") # returns 2
count_words("No results for search term `s`") # returns 6
count_words(" Hello") # returns 1
# ... and so on
count_words("Hello"); # returns 1 as int
count_words("Hello, World!") # returns 2
count_words("No results for search term `s`") # returns 6
count_words(" Hello") # returns 1
# ... and so on
What kind of tests we got for your code:

Function have to count words, but not spaces, so be sure that it does right.
Empty string has no words.
String with spaces around should be trimmed.
Non-whitespace (ex. breakspace, unicode chars) should be assumed as delimiter
Be sure that words with chars like -, ', ` are counted right.
```

8. Remove First and Last Character Part Two

```
This is a spin off of my first kata. You are given a list of character sequences
 as a comma separated string. Write a function which returns another string cont
aining all the character sequences except the first and the last ones. If the in
put string is empty, or the removal of the first and last items would cause the
string to be empty, return a null value.
```

9. Implement a Filter function

```
What we want to implement is a filter function, like Array.filter(), also simila
r to the _.filter() in underscore.js and lodash.js.
The usage is quite simple, like:
[1,2,3,4].filter((num)=>{ return num > 3})should output   [4]
```

10. Prefill an Array

```
Create the function prefill that returns an array of n elements that all have th
e same value v.  See if you can do this without using a loop.
You have to validate input:

v can be anything (primitive or otherwise)
if v is ommited, fill the array with undefined
if n is 0, return an empty array
if n is anything other than an integer or integer-formatted string (e.g. '123')
that is >=0, throw a TypeError

When throwing a TypeError, the message should be n is invalid, where you replace
 n for the actual value passed to the function.
Code Examples
    prefill(3,1) --> [1,1,1]
prefill(2,"abc") --> ['abc','abc']

prefill("1", 1) --> [1]

prefill(3, prefill(2,'2d'))
  --> [['2d','2d'],['2d','2d'],['2d','2d']]

prefill("xyz", 1)
  --> throws TypeError with message "xyz is invalid"
prefill(3,1) --> [1,1,1]

prefill(2,"abc") --> ['abc','abc']

prefill("1", 1) --> [1]

prefill(3, prefill(2,'2d'))
  --> [['2d','2d'],['2d','2d'],['2d','2d']]

prefill("xyz", 1)
  --> throws TypeError with message "xyz is invalid"
prefill(3,1) --> [1,1,1]

prefill(2,"abc") --> ['abc','abc']

prefill("1", 1) --> [1]

prefill(3, prefill(2,'2d'))
  --> [['2d','2d'],['2d','2d'],['2d','2d']]

prefill("xyz", 1)
  --> throws TypeError with message "xyz is invalid"
prefill 3, 1 #returns [1, 1, 1]

prefill 2, "abc" #returns ["abc","abc"]

prefill "1", 1 #returns [1]

prefill 3, prefill(2, "2d")
  #returns [['2d','2d'],['2d','2d'],['2d','2d']]

prefill "xyz", 1
  #throws TypeError with message "xyz is invalid"
```

11. Cross Product of Vectors

Make a function called crossProduct that takes two 3 dimensional vectors (in the form of two arrays) and returns their cross product.
You need to check if the passed arguments are of the expected format, otherwise throw the message: "Arguments are not 3D vectors!".

```
crossProduct([1,0,0],[0,1,0]) //should return [0,0,1]
crossProduct('gobbledigook', [1,1,1]) //should throw the string "Arguments are not 3D vectors!"
crossProduct([1,0,0],[0,1,0]) #should return [0,0,1]
crossProduct('gobbledigook', [1,1,1]) #should throw the string "Arguments are not 3D vectors!"
```

Your function should handle non integers.
More info on cross products: https://en.wikipedia.org/wiki/Cross_product

```
crossprod([1,0,0], [0,1,0], 3, 3)         /* should return [0,0,1] */
crossprod([1,2,3,4,5], [5,4,3,2,1], 5, 5) /* should return NULL */
crossprod([6,6,6], NULL, 3, 3)            /* should return NULL */
crossprod(NULL, NULL, 3, 3)               /* should return NULL */
```

12. Sequence generator

Implement function sequence, which returns new n-size Array filled according to pattern.
pattern may be:

a function that takes two: (element, index), one: (element) or any arguments (similar to map function), then filled running this function, in other words: function describes sequence,
number, string or any other object, then filled by copying, this object n-times.

Examples:

```
sequence(3, 4); // [4, 4, 4]
sequence(5, []); // [[], [], [], [], []]
sequence(2, "s"); // ["s", "s"]
sequence(5, (x, idx) => idx%2) // [0, 1, 0, 1, 0];
sequence(10, (x, idx) => idx+1) // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note: Sequences are great to work with functional methods like map, reduce, forEach, every or any. For example:

```
// sum of numbers 1-10
let sum = sequence(10, (x, idx) => idx+1).reduce((sum, num) => sum + num);
```

Be careful with long sequences. They are just arrays, every element is created when function is called.
For lazy sequences (elements created when needed) use Iterator.

13. Base Conversion

In this kata you have to implement a base converter, which converts positive integers between arbitrary bases / alphabets. Here are some pre-defined alphabets:

```
var Alphabet = {
  BINARY:        '01',
  OCTAL:         '01234567',
  DECIMAL:       '0123456789',
  HEXA_DECIMAL:  '0123456789abcdef',
  ALPHA_LOWER:   'abcdefghijklmnopqrstuvwxyz',
  ALPHA_UPPER:   'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  ALPHA:         'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',
  ALPHA_NUMERIC: '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
};
public class Alphabet
{
   public const string BINARY = "01";
   public const string OCTAL = "01234567";
   public const string DECIMAL = "0123456789";
   public const string HEXA_DECIMAL = "0123456789abcdef";
   public const string ALPHA_LOWER = "abcdefghijklmnopqrstuvwxyz";
   public const string ALPHA_UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   public const string ALPHA = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
   public const string ALPHA_NUMERIC = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
}
bin      = '01'
oct      = '01234567'
dec      = '0123456789'
hex      = '0123456789abcdef'
allow    = 'abcdefghijklmnopqrstuvwxyz'
allup    = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
alpha    = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
alphanum = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
bin      = '01'
oct      = '01234567'
dec      = '0123456789'
hex      = '0123456789abcdef'
allow    = 'abcdefghijklmnopqrstuvwxyz'
allup    = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
alpha    = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
alphanum = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
newtype Alphabet = Alphabet { getDigits :: [Char] } deriving (Show)
bin, oct, dec, hex, alphaLower, alphaUpper, alpha, alphaNumeric :: Alphabet
bin = Alphabet $ "01"
oct = Alphabet $ ['0'..'7']
dec = Alphabet $ ['0'..'9']
hex = Alphabet $ ['0'..'9'] ++ ['a'..'f']
alphaLower    = Alphabet $ ['a'..'z']
alphaUpper    = Alphabet $ ['A'..'Z']
alpha         = Alphabet $ ['a'..'z'] ++ ['A'..'Z']
alphaNumeric  = Alphabet $ ['0'..'9'] ++ ['a'..'z'] ++ ['A'..'Z']
const char * bin = "01";
const char * oct = "01234567";
const char * dec = "0123456789";
const char * hex = "0123456789abcdef";
const char * allow = "abcdefghijklmnopqrstuvwxyz";
const char * alup = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
const char * alpha = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
const char * alnum = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
WXYZ";
The function convert() should take an input (string), the source alphabet (strin
g) and the target alphabet (string). You can assume that the input value always
consists of characters from the source alphabet. You don't need to validate it.
Examples
// convert between numeral systems
convert("15", Alphabet.DECIMAL, Alphabet.BINARY); // should return "1111"
convert("15", Alphabet.DECIMAL, Alphabet.OCTAL); // should return "17"
convert("1010", Alphabet.BINARY, Alphabet.DECIMAL); // should return "10"
convert("1010", Alphabet.BINARY, Alphabet.HEXA_DECIMAL); // should return "a"

// other bases
convert("0", Alphabet.DECIMAL, Alphabet.ALPHA); // should return "a"
convert("27", Alphabet.DECIMAL, Alphabet.ALPHA_LOWER); // should return "bb"
convert("hello", Alphabet.ALPHA_LOWER, Alphabet.HEXA_DECIMAL); // should return
"320048"
convert("SAME", Alphabet.ALPHA_UPPER, Alphabet.ALPHA_UPPER); // should return "S
AME"
// convert between numeral systems
Convert("15", Alphabet.DECIMAL, Alphabet.BINARY); // should return "1111"
Convert("15", Alphabet.DECIMAL, Alphabet.OCTAL); // should return "17"
Convert("1010", Alphabet.BINARY, Alphabet.DECIMAL); // should return "10"
Convert("1010", Alphabet.BINARY, Alphabet.HEXA_DECIMAL); // should return "a"

// other bases
Convert("0", Alphabet.DECIMAL, Alphabet.ALPHA); // should return "a"
Convert("27", Alphabet.DECIMAL, Alphabet.ALPHA_LOWER); // should return "bb"
Convert("hello", Alphabet.ALPHA_LOWER, Alphabet.HEXA_DECIMAL); // should return
"320048"
Convert("SAME", Alphabet.ALPHA_UPPER, Alphabet.ALPHA_UPPER); // should return "S
AME"
convert("15", dec, bin)       ==>  "1111"
convert("15", dec, oct)       ==>  "17"
convert("1010", bin, dec)     ==>  "10"
convert("1010", bin, hex)     ==>  "a"
convert("0", dec, alpha)      ==>  "a"
convert("27", dec, allow)     ==>  "bb"
convert("hello", allow, hex)  ==>  "320048"
convert("15", dec, bin)   # should return "1111"
convert("15", dec, oct)   # should return "17"
convert("1010", bin, dec) # should return "10"
convert("1010", bin, hex) # should return "a"
convert("0", dec, alpha)  # should return "a"
convert("27", dec, allow) # should return "bb"
convert("hello", allow, hex) # should return "320048"
convert dec bin "15"   `shouldBe` "1111"
convert dec oct "15"   `shouldBe` "17"
convert bin dec "1010" `shouldBe` "10"
convert bin hex "1010" `shouldBe` "a"
convert dec alpha      "0"     `shouldBe` "a"
convert dec alphaLower "27"    `shouldBe` "bb"
convert alphaLower hex "hello" `shouldBe` "320048"
convert("15", dec, bin)       // should return "1111"
convert("15", dec, oct)       // should return "17"
convert("1010", bin, dec)     // should return "10"
convert("1010", bin, hex)     // should return "a"
convert("0", dec, alpha)      // should return "a"
convert("27", dec, allow)     // should return "bb"
convert("hello", allow, hex)  // should return "320048"
```

Additional Notes:

The maximum input value can always be encoded in a number without loss of precision in JavaScript. In Haskell, intermediate results will probably be too large for Int.
The function must work for any arbitrary alphabets, not only the pre-defined ones
You don't have to consider negative numbers

14. Closures and Scopes

We want to create a function, which returns an array of functions, which return their index in the array. For better understanding, here an example:

```
var callbacks = createFunctions(5); // create an array, containing 5 functions

callbacks[0](); // must return 0
callbacks[3](); // must return 3
```

We already implemented that function, but when we actually run the code, the result doesn't look like what we expected. Can you spot, what's wrong with it? A test fixture is also available

15. A function within a function

Given an input n, write a function always that returns a function which returns n. Ruby should return a lambda or a proc.

```
var three = always(3);
three(); // returns 3
three = always(3)
three() # returns 3
three = always(3)
three.call # returns 3
three = always(3)
three() /* returns 3 */
let three = always 3
three () -- returns 3
(def three (always 3))
(three) ;; returns 3
three = always(3)
three.() #=> 3
```

```
function three = always(3);
three(); // returns 3
Func three = Kata.Always(3);
three(); // returns 3
```

16. Can you keep a secret?

There's no such thing as private properties on a coffeescript object!
But, maybe there are?
Implement a function createSecretHolder(secret) which accepts any value as secret and returns an object with ONLY two methods

getSecret() which returns the secret
setSecret() which sets the secret

```
obj = createSecretHolder(5)
obj.getSecret() # returns 5
obj.setSecret(2)
obj.getSecret() # returns 2
```

17. Using closures to share class state

In object-oriented programming, it is sometimes useful to have private shared state among all instances of a class; in other languages, like ruby, this shared state would be tracked with a class variable. In javascript we achieve this through closures and immediately-invoked function expressions.
In this kata, I want you to write make a Cat constructor that takes arguments name and weight to instantiate a new cat object. The constructor should also have an averageWeight method that returns the average weight of cats created with the constructor.

```
garfield = new Cat('garfield', 25);
Cat.averageWeight(); // 25

felix = new Cat('felix', 15);
Cat.averageWeight();   // now 20
```

But that's not all. Cats can change weight. Use Object.defineProperty to write custom setters and getters for the weight property so that the following works properly even as instances change their weight value:

```
felix.weight = 25;
felix.weight // 25
Cat.averageWeight(); // now 25
```

Object.defineProperty must be used to pass all tests. Storing a reference to all instances and recalculating the average weight each time is easier, but would prevent garbage collection from working properly if used in a production environment.
Finally, since average weight is an aggregate statistic it's important that we validate constructor arguments so that no cats are created without a specified weight; so, make sure to throw an error if both arguments are not recieved by the constructor.
Summary of requirements:

Cat constructor, requiring arguments for name and weight
Throw an error if name or weight not specified when invoking the constructor.
Cat.averageWeight() method should give the average weight of all cat instances created with Cat, even after if the instance's properties have changed.
Must use Object.defineProperty

18. A Chain adding function

We want to create a function that will add numbers together when called in succession.

```
add(1)(2);
// returns 3
add(1).(2);
// returns 3
```

We also want to be able to continue to add numbers to our chain.

```
add(1)(2)(3); // 6
add(1)(2)(3)(4); // 10
add(1)(2)(3)(4)(5); // 15
add(1).(2).(3); // 6
add(1).(2).(3).(4); // 10
add(1).(2).(3).(4).(5); // 15
```

and so on.

A single call should return the number passed in.

```
add(1); // 1
add(1); // 1
```

We should be able to store the returned values and reuse them.

```
var addTwo = add(2);
addTwo; // 2
addTwo + 5; // 7
addTwo(3); // 5
addTwo(3)(5); // 10
var addTwo = add(2);
addTwo; // 2
addTwo + 5; // 7
addTwo(3); // 5
addTwo(3).(5); // 10
```

We can assume any number being passed in will be valid whole number.

19. Function Cache

If you are calculating complex things or execute time-consuming API calls, you sometimes want to cache the results. In this case we want you to create a function wrapper, which takes a function and caches its results depending on the arguments, that were applied to the function.

Usage example:

```
var complexFunction = function(arg1, arg2) { /* complex calculation in here */ };
var cachedFunction = cache(complexFunction);

cachedFunction('foo', 'bar'); // complex function should be executed
cachedFunction('foo', 'bar'); // complex function should not be invoked again, instead the cached result should be returned
cachedFunction('foo', 'baz'); // should be executed, because the method wasn't invoked before with these arguments
```

20. Function Composition

Function composition is a mathematical operation that mainly presents itself in lambda calculus and computability. It is explained well here, but this is my explanation, in simple mathematical notation:

```
f3 = compose( f1 f2 )
   Is equivalent to...
```

f3(a) = f1(f2(a))Your task is to create a compose function to carry out this task, which will be passed two functions or lambdas. Ruby functions will be passed, and should return, either a proc or a lambda. Remember that the resulting composed function may be passed multiple arguments!

```
compose(f , g)(x)
```

```
=> f( g( x ) )
compose(f , g).(x)
=> f.( g.( x ) )
compose(f , g)(x)
=> f( g( x ) )
((compose f  g) x)
=> (f (g x) )
compose(f , g)(x)
=> f( g( x ) )
```

This kata is not available in haskell; that would be too easy!

21. Function composition

Javascript functions can be combined to form new functions. For example the functions addOne and multTwo can be combined to form a new function which first adds one and then multiplies by two, as follows:

```
const addOne = (a) => a + 1
const multTwo = (b) => b * 2
const addOneMultTwo = (c) => multTwo(addOne(c))

addOneMultTwo(5) // returns 12
```

Combining functions like this is called function composition. Functional programming libraries in Javascript such as Ramda include a generic compose function which does the heavy lifting of combining functions for you. So you could implement addOneMultTwo as follows:

```
const addOneMultTwo = compose(multTwo, addOne)

addOneMultTwo(5) // returns 12
```

A simple implementation of compose, could work as follows:

```
const compose = (f, g) => (a) => f(g(a))
```

The arguments f and g are unary functions (i.e. functions which take one argument). The problem with this compose function is that it only composes two functions. Your task is to write a compose function which can compose any number of functions together.

22. Stringing me along

Create a function that will allow you to pass in a string, with the ability to add to this with more function calls. When it is finally passed an empty argument return the full concatinated string of all arguments pased previously.

For example:

```
createMessage("Hello")("World!")("how")("are")("you?")();
```

This will return the following:

```
"Hello World! how are you?"
```

23. I Spy

NOTE: The test cases for this kata are broken, but for some reason CodeWars has locked them and I cannot edit them. Specifically, the returned function is not properly testing that old values are remembered. If and when I can fix the problem, I will, but I don't see any way to do that due to the lock.

In testing, a spy function is one that keeps track of various metadata regarding its invocations. Some examples of properties that a spy might track include:

Whether it was invoked
How many times it was invoked
What arguments it was called with
What contexts it was called in
What values it returned
Whether it threw an error

For this kata, implement a spyOn function which takes any function func as a parameter and returns a spy for func. The returned spy must be callable in the same manner as the original func, and include the following additional properties/methods:

.callCount() ÃƒÂ¢Ã‚ ÂÃ‚ Â” returns the number of times spy has been called
.wasCalledWith(val) ÃƒÂ¢Ã‚ ÂÃ‚ Â“ returns true if spy was ever called with val, else returns false.
.returned(val) ÃƒÂ¢Ã‚ ÂÃ‚ Â” returns true if spy ever returned val, else returns false

Below is a specific example of how spyOn might work in the wild.

```
function adder(n1, n2) { return n1 + n2; }
var adderSpy = spyOn( adder );

adderSpy(2, 4); // returns 6
adderSpy(3, 5); // returns 8
adderSpy.callCount(); // returns 2
adderSpy.wasCalledWith(4); // true
adderSpy.wasCalledWith(0); // false
adderSpy.returned(8); // true
adderSpy.returned(0); // false
```

24. Calculating with Functions

This time we want to write calculations using functions and get the results. Let's have a look at some examples:

```
seven(times(five())); // must return 35
four(plus(nine())); // must return 13
eight(minus(three())); // must return 5
six(dividedBy(two())); // must return 3
seven(times(five)) # must return 35
four(plus(nine)) # must return 13
eight(minus(three)) # must return 5
six(divided_by(two)) # must return 3
seven(times(five())) # must return 35
four(plus(nine())) # must return 13
eight(minus(three())) # must return 5
six(divided_by(two())) # must return 3
```

Requirements:

There must be a function for each number from 0 ("zero") to 9 ("nine")
There must be a function for each of the following mathematical operations: plus, minus, times, dividedBy (divided_by in Ruby and Python)
Each calculation consist of exactly one operation and two numbers
The most outer function represents the left operand, the most inner function represents the right operand
Divison should be integer division. For example, this should return 2, not 2.666666...:

```
eight(dividedBy(three()));
eight(divided_by(three))
eight(divided_by(three()))
```

25. SantaClausable Interface

You probably know, that in Javascript (and also Ruby) there is no concept of interfaces. There is only a concept of inheritance, but you can't assume that a certain method or property exists, just because it exists in the parent prototype / class. We want to find out, whether a given object fulfils the requirements to implement the "SantaClausable" interface. We need to implement a method which checks for this interface.

```
Rules
The SantaClausable interface is implemented, if all of the following methods are
 defined on an object:

sayHoHoHo() / say_ho_ho_ho
distributeGifts() / distribute_gifts
goDownTheChimney() / go_down_the_chimney

Example
var santa = {
    sayHoHoHo: function() { console.log('Ho Ho Ho!') },
    distributeGifts: function() { console.log('Gifts for all!'); },
    goDownTheChimney: function() { console.log('*whoosh*'); }
};

var notSanta = {
    sayHoHoHo: function() { console.log('Oink Oink!') }
    // no distributeGifts() and no goDownTheChimney()
};

isSantaClausable(santa); // must return TRUE
isSantaClausable(notSanta); // must return FALSE
santa =
  sayHoHoHo: ->
    console.log "Ho Ho Ho!"

  distributeGifts: ->
    console.log "Gifts for all!"

  goDownTheChimney: ->
    console.log "*whoosh*"

notSanta = sayHoHoHo: ->
  console.log "Oink Oink!"
  # no distributeGifts() and no goDownTheChimney()

isSantaClausable santa # must return TRUE
isSantaClausable notSanta # must return FALSE
class SantaClaus
    def say_ho_ho_ho
        # Ho Ho Ho!
    end

    def distribute_gifts
        # Gifts for all!
    end

    def go_down_the_chimney
        # Whoosh!
    end
end

class NotSantaClaus
    def say_ho_ho_ho
    end
end

is_santa_clausable(SantaClaus.new) # must return TRUE
is_santa_clausable(NotSantaClaus.new) # must return FALSE
Additional Information on this Topic

Duck Typing (Wikipedia)
```

26. new with apply

```
In JavaScript we can create objects using the new operator.
For example, if you have this constructor function:
function Greeting(name) {
  this.name = name;
}

Greeting.prototype.sayHello = function() {
  return "Hello " + this.name;
};


Greeting.prototype.sayBye = function() {
  return "Bye " + this.name;
};
You can create a Greeting object in this way:
  var greeting = new Greeting('John');
new operator is evil because it produces a highly coupled code, difficult to mai
ntain and test.
Some patterns to reduce coupling are object factories or dependency injection.
These patterns can benefit of the construct() function.
This function receives a constructor function and possibly some arguments and it
 returns a new object constructed with the function and the passed arguments.
This is another way to create the greeting object:
var greeting = construct(Greeting, 'John');
And a factory could use like this:
  function factory() {
    return {
      createGreeting() {
        return construct(Greeting, arguments);
      }
      ...
    }
  }Your work is to implement the construct() function.
```

27. Extract Nested Object Reference

```
You are given a complex object that has many deeply nested variables. You don't
want to go the usual if obj.property == null route. Create a prototype method th
at given a nested path, either return the value or undefined.
var obj = {
  person: {
    name: 'joe',
    history: {
      hometown: 'bratislava',
      bio: {
        funFact: 'I like fishing.'
      }
    }
  }
};

obj.hash('person.name'); // 'joe'
obj.hash('person.history.bio'); // { funFact: 'I like fishing.' }
obj.hash('person.history.homeStreet'); // undefined
obj.hash('person.animal.pet.needNoseAntEater'); // undefined
```

28. Array Helpers

```
This kata is designed to test your ability to extend the functionality of built-
in classes. In this case, we want you to extend the built-in Array class with th
```

e following methods: square(), cube(), average(), sum(), even() and odd().
Explanation:

square() must return a copy of the array, containing all values squared
cube() must return a copy of the array, containing all values cubed
average() must return the average of all array values; on an empty array must return NaN (note: the empty array is not tested in Ruby!)
sum() must return the sum of all array values
even() must return an array of all even numbers
odd() must return an array of all odd numbers

Note: the original array must not be changed in any case!
Example

```
var numbers = [1, 2, 3, 4, 5];

numbers.square();  // must return [1, 4, 9, 16, 25]
numbers.cube();    // must return [1, 8, 27, 64, 125]
numbers.average(); // must return 3
numbers.sum();     // must return 15
numbers.even();    // must return [2, 4]
numbers.odd();     // must return [1, 3, 5]
numbers = [1, 2, 3, 4, 5]

numbers.square()  # must return [1, 4, 9, 16, 25]
numbers.cube()    # must return [1, 8, 27, 64, 125]
numbers.average() # must return 3
numbers.sum()     # must return 15
numbers.even()    # must return [2, 4]
numbers.odd()     # must return [1, 3, 5]
```

29. Replicate `new`

TL;DR: write a nouveau function that replicates all the behavior of the new operator.

Aside: Operators?
In JavaScript, perhaps no operator is as complicated as new. "Wait; new is an operator?" Yep; an operator is something that operates on one or more operands and evaluates to a result. Binary operators like + and !== operate on two operands:

```
5 + 5 evaluates to 10
{} !== [] evaluates to true
```

Whereas unary operators like + and typeof take one operand (hmm, + is both a unary and binary operator, how 'bout that!):

```
+'5' evaluates to 5
typeof '5' evaluates to 'string'
```

Ultimately operators are functions with different syntax. They take inputs/operands and return/evaluate to something. In fact, some JS operators can be re-written as functions.

New
So what about new? Well, the unary operator new is intended to create "instances" of a constructor function. To be more precise, the operation new Constructor(arg1, arg2, ...argX) does the following:

Creates an empty object (which we'll call instance) which prototypally inherits from Constructor.prototype
Binds Constructor to instance (meaning this is instance) and invokes Constructor with any arguments passed in
If the return value of Constructor is an object (including arrays, functions, da

tes, regexes, etc.) the operation evaluates to that object
Otherwise, the operation evaluates to instance

Let's see some examples:

```
function Person (name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.introduce = function(){
  return 'My name is ' + this.name + ' and I am ' + this.age;
};
var john = new Person('John', 30);
var jack = new Person('Jack', 40);
console.log( john.introduce() ); // My name is John and I am 30
console.log( jack.introduce() ); // My name is Jack and I am 40

function ReturnsArray (name) {
  this.name = name;
  return [1, 2, 3];
}
var arr = new ReturnsArray('arr?');
console.log( arr.name ); // undefined
console.log( arr ); // [1, 2, 3]
```

Oof! No wonder people get confused about new. The good news isÃƒÂ¢Ã‚ÂÃ‚Â¦ everything new can do, you can do too.

Exercise
Your mission: write a function nouveau (that's French for "new") which takes one function parameter (the constructor), plus an unknown number of additional parameters of any type (arguments for the constructor). When invoked, nouveau should do everything new does and return the same object new would evaluate to, as specified above.

```
var john = nouveau(Person, 'John', 30); // same result as above
```

Good luck!

30. Sum of Digits / Digital Root

In this kata, you must create a digital root function.
A digital root is the recursive sum of all the digits in a number. Given n, take the sum of the digits of n. If that value has more than one digit, continue reducing in this way until a single-digit number is produced. This is only applicable to the natural numbers.
Here's how it works:

```
digital_root(16)
=> 1 + 6
=> 7

digital_root(942)
=> 9 + 4 + 2
=> 15 ...
=> 1 + 5
=> 6

digital_root(132189)
=> 1 + 3 + 2 + 1 + 8 + 9
=> 24 ...
=> 2 + 4
=> 6

digital_root(493193)
=> 4 + 9 + 3 + 1 + 9 + 3
=> 29 ...
=> 2 + 9
=> 11 ...
=> 1 + 1
=> 2
digitalRoot(16)
=> 1 + 6
=> 7

digitalRoot(942)
=> 9 + 4 + 2
=> 15 ...
=> 1 + 5
=> 6

digitalRoot(132189)
=> 1 + 3 + 2 + 1 + 8 + 9
=> 24 ...
=> 2 + 4
=> 6

digitalRoot(493193)
=> 4 + 9 + 3 + 1 + 9 + 3
=> 29 ...
=> 2 + 9
=> 11 ...
=> 1 + 1
=> 2
digital_root 16
=> 1 + 6
=> 7

digital_root 942
=> 9 + 4 + 2
=> 15 ...
=> 1 + 5
=> 6

digital_root 132189
=> 1 + 3 + 2 + 1 + 8 + 9
=> 24 ...
=> 2 + 4
=> 6

digital_root 493193
=> 4 + 9 + 3 + 1 + 9 + 3
=> 29 ...
=> 2 + 9
=> 11 ...
=> 1 + 1
=> 2
DigitalRoot(16)
=> 1 + 6
=> 7

DigitalRoot(942)
=> 9 + 4 + 2
=> 15 ...
=> 1 + 5
=> 6

DigitalRoot(132189)
=> 1 + 3 + 2 + 1 + 8 + 9
=> 24 ...
=> 2 + 4
=> 6

DigitalRoot(493193)
```

```
=> 4 + 9 + 3 + 1 + 9 + 3
=> 29 ...
=> 2 + 9
=> 11 ...
=> 1 + 1
=> 2
```

31. Fun with ES6 Classes #2 - Animals and Inheritance

Fun with ES6 Classes #2 - Animals and Inheritance

Overview

Preloaded for you in this Kata is a class Animal:

```
class Animal {
  constructor(name, age, legs, species, status) {
    this.name = name;
    this.age = age;
    this.legs = legs;
    this.species = species;
    this.status = status;
  }
  introduce() {
    return `Hello, my name is ${this.name} and I am ${this.age} years old.`;
  }
}
public class Animal
{
  public int Age;
  public int Legs;
  public string Name;
  public string Species;
  public string Status;

  public Animal(string name, int age, int legs, string species, string status)
  {
    this.Name = name;
    this.Age = age;
    this.Legs = legs;
    this.Species = species;
    this.Status = status;
  }

  public virtual string Introduce()
  {
    return $"Hello, my name is {this.Name} and I am {this.Age} years old.";
  }
}
```

Task

Define the following classes that inherit from Animal.

I. Shark

The constructor function for Shark should accept 3 arguments in total in the following order: name, age, status. All sharks should have a leg count of **0 (since they obviously do not have any legs) and should have a species of "shark".**

II. Cat

The constructor function for Cat should accept the same 3 arguments as with Shark: name, age, status. Cats should always have a leg count of 4 and a species of "cat".

Furthermore, the introduce/Introduce method for Cat should be identical to the original except there should be exactly 2 spaces and the words "Meow meow!" after the phrase. For example:

```
var example = new Cat("Example", 10, "Happy");
example.introduce() === "Hello, my name is Example and I am 10 years old.  Meow meow!"; // Notice the TWO spaces - very important
Cat example = new Cat("Example", 10, "Happy);
example.Introduce() => "Hello, my name is Example and I am 10 years old.  Meow m
```

eow!"; // Notice the TWO spaces - very important

III. Dog

The Dog constructor should accept 4 arguments in the specified order: name, age, status, master. master is the name of the dog's master which will be a string. Furthermore, dogs should have 4 legs and a species of "dog".

Dogs have an identical introduce/Introduce method as any other animal, but they have their own method called greetMaster/GreetMaster which accepts no arguments and returns "Hello (insert_master_name_here)" (of course not the literal string but replace the (insert_master_name_here) with the name of the dog's master).

32. Fun with ES6 Classes #3 - Cuboids, Cubes and Getters

Fun with ES6 Classes #3 - Cuboids, Cubes and Getters

Task

Define the following classes.

I. Cuboid

The object constructor for the class Cuboid should receive exactly three arguments in the following order: length, width, height and store these three values in this.length, this.width and this.height respectively.

The class Cuboid should then have a getter surfaceArea which returns the surface area of the cuboid and a getter volume which returns the volume of the cuboid.

II. Cube

class Cube is a subclass of class Cuboid. The constructor function of Cube should receive one argument only, its length, and use that value passed in to set this.length, this.width and this.height.

Hint: Make a call to super, passing in the correct arguments, to make life easier ;)

Related Articles

Listed below are a few articles of interest that may help you complete this Kata:

Stack Overflow - What are getters and setters in ES6?
getter - Javascript | MDN

33. Lazy evaluation

Lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed.

Implement the Lazy function. This function has two methods:

add(fn[, arg1, arg2, ...]): adds the fn function to the lazy chain evaluation. This function could receive optional arguments.
invoke(target): performs the evaluation chain over the target array.

For example:
Given these functions:

```
function max() {
    return Math.max.apply(null, arguments);
}

function filterNumbers() {
  return Array.prototype.filter.call(arguments, function(value) {
    return isNumeric(value);
  });
}

function isNumeric(n) {
  return !isNaN(n) && Number(n) === n;
}

function filterRange(min, max) {
  var args = Array.prototype.slice.call(arguments, 2);
  return Array.prototype.filter.call(args, function(value) {
    return min <= value && value <= max;
  });
}
```

You could use it via composition:

```
max.apply(null, filterRange.apply(null, [1, 3].concat(filterNumbers(1, 2, "3", 7, 6, 5))));
```

But this solution is not reusable.
A better approach could be to use composition with lazy invocation:

```
new Lazy()
      .add(filterNumbers)
      .add(filterRange, 2, 7)
      .add(max)
      .invoke([1, 8, 6, [], "7", -1, {v: 5}, 4]); //6
```

Step by step, this is what should happen when invoke function is called:

```
filterNumbers(1, 8, 6, [], "7", -1, {v: 5}, 4) // == [1, 8, 6, -1, 4]
//            ^------------------------------- from invoke
filterRange(2, 7, 1, 8, 6, -1, 4) // == [6, 4]
// from add ---^  ^------------- from previous result
max(6, 4) // == 6
//  ^--- from previous result

Result from invoke: 6
//                  ^ from last result
```

34. Tail recursion with trampoline

Functional programming prefers recursion over iteration.
Recursive functions are often more readable than its iterative version.
Besides, functional programming avoids declaring variables, so functions do not have mutable state. Recursion can solve problems without mutable state.
Here's an example:
We want to create a function sum(number) that calculates the sum of numbers between 1 and the passed number.

```
sum(1); //1
sum(2); //1+2 = 3
sum(4); //10
sum(10); //55
```

The iterative versiÃƒÂƒÃ‚Â³n of sum(number) could be:

```
function iterativeSum(n) {
  var i;
  var sum = 0;
  for (i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}
```

The recursive implementation is more elegant and it has not mutable state:

```
function recursiveSum(n) {
  if (n === 0) {
    return 0;
  } else {
    return n + recursiveSum(n - 1);
  }
}
```

But it has a problem of memory consumption.

```
recursiveSum(10); //55
recursiveSum(99999); //RangeError: Maximum call stack size exceeded
```

Some languages can deal with this problem by using a technique known as tail recursion.

A recursive function is tail recursive if the final result of the recursive call is the final result of the function itself. If the result of the recursive call must be further processed (say, by adding 1 to it, or consing another element onto the beginning of it), it is not tail recursive.

The benefit of tail recursion is that tail calls can be implemented without adding a new stack frame to the call stack.
This could be the tail recursive solution of our example:

```
function tailRecursionSum(n) {
  function _sum(ac, n) {
    if (n === 0) {
      return ac;
    } else {
      return _sum(ac + n, n - 1);
    }
  }

  return _sum(0, n);
}
```

But JavaScript still does not support tail recursion:

```
tailRecursionSum(10); //55
tailRecursionSum(99999); //RangeError: Maximum call stack size exceeded
```

Trampolining is a technique that allows us to create functions with the elegance of the recursive solution but without its memory issue, because, although it does not seem, the solution is actually iterative.
This could be our solution:

```
function trampolineSum(n) {
  function _sum(n, ac) {
    if (n === 0) {
      return ac;
    } else {
      return thunk(_sum, n - 1, ac + n);
    }
  }

  return trampoline(thunk(_sum, n, 0));
}
```

Note that the solution has the same structure as tailRecursionSum(n), but there is no recursive calls. Instead two auxiliary functions appear: thunk(fn /*, args */) and trampoline(thunk).

thunk(fn /*, args */) is a function that receives a function and possibly some arguments to be passed to the function and returns a function. When this returned function is called, it returns the result of execute the fnfunction. In functional programming, a thunk is a deferred expression (function). Its evaluation is postponed until it's really needed.
trampoline(thunk) is a function that executes repeatedly the thunk argument until it returns a non function value. Then this last value is returned.

Here is an example:

```
function add(a, b) {
    return a + b;
}

thunk(add, 4, 5)(); //9

trampoline(thunk(add, 4, 5)); //9
```

Another example:

```
function add(x , y) {
  return function() {
    return x + y + 6;
  }
}
trampoline(thunk(add, 4, 5)); //15 <- 4 + 5 + 6
```

Your job is to implement thunk(fn /*, args */) and trampoline(thrunk) functions.

Also you have to refactor the implementation of isEven(number) and isOdd(number) functions to use the trampoline(thunk) function.

```
function isEven(n) {
  return (n === 0 ? true : isOdd(n - 1));
}

function isOdd(n) {
  return (n === 0 ? false : isEven(n - 1));
}
```

35. Functional SQL

In this Kata we are going to mimic the SQL syntax with JavaScript (or TypeScript).

To do this, you must implement the query() function. This function returns and object with the next methods:

```
{
  select: ...,
  from: ...,
  where: ...,
  orderBy: ...,
  groupBy: ...,
  having: ...,
  execute: ...
}
```

The methods are chainable and the query is executed by calling the execute() method.

```
SELECT * FROM numbers
var numbers = [1, 2, 3];
query().select().from(numbers).execute(); //[1, 2, 3]

//clauses order does not matter
query().from(numbers).select().execute(); //[1, 2, 3]
```

Of course, you can make queries over object collections:

```
var persons = [
  {name: 'Peter', profession: 'teacher', age: 20, maritalStatus: 'married'},
  {name: 'Michael', profession: 'teacher', age: 50, maritalStatus: 'single'},
  {name: 'Peter', profession: 'teacher', age: 20, maritalStatus: 'married'},
  {name: 'Anna', profession: 'scientific', age: 20, maritalStatus: 'married'},
  {name: 'Rose', profession: 'scientific', age: 50, maritalStatus: 'married'},
  {name: 'Anna', profession: 'scientific', age: 20, maritalStatus: 'single'},
  {name: 'Anna', profession: 'politician', age: 50, maritalStatus: 'married'}
];

//SELECT * FROM persons
query().select().from(persons).execute(); // [{name: 'Peter',...}, {name: 'Michael', ...}]
```

You can select some fields:

```
function profession(person) {
  return person.profession;
}

//SELECT profession FROM persons
query().select(profession).from(persons).execute(); //select receives a function that will be called with the values of the array //["teacher","teacher","teacher","scientific","scientific","scientific","politician"]
```

If you repeat a SQL clause (except where() or having()), an exception will be thrown

```
query().select().select().execute(); //Error('Duplicate SELECT');
query().select().from([]).select().execute(); //Error('Duplicate SELECT');
query().select().from([]).from([]).execute(); //Error('Duplicate FROM');
query().select().from([]).where([]).where([]) //This is an AND filter (see below)
```

You can omit any SQLclause:

```
var numbers = [1, 2, 3];
```

```
query().select().execute(); //[]
query().from(numbers).execute(); // [1, 2, 3]
query().execute(); // []
You can apply filters:
function isTeacher(person) {
  return person.profession === 'teacher';
}

//SELECT profession FROM persons WHERE profession="teacher"
query().select(profession).from(persons).where(isTeacher).execute(); //["teacher
", "teacher", "teacher"]

//SELECT * FROM persons WHERE profession="teacher"
query().select().from(persons).where(isTeacher).execute(); //[{person: 'Peter',
profession: 'teacher', ...}, ...]

function name(person) {
  return person.name;
}

//SELECT name FROM persons WHERE profession="teacher"
query().select(name).from(persons).where(isTeacher).execute();//["Peter", "Micha
el", "Peter"]
Agrupations are also possible:
//SELECT * FROM persons GROUP BY profession <- Bad in SQL but possible in this k
ata
query().select().from(persons).groupBy(profession).execute();
[
  ["teacher",
     [
       {
        name: "Peter",
        profession: "teacher"
        ...
      },
      {
        name: "Michael",
        profession: "teacher"
        ...
      }
    ]
  ],
  ["scientific",
    [
       {
          name: "Anna",
          profession: "scientific"
        },
     ...
   ]
  ]
  ...
]
You can mix where() with groupBy():
//SELECT * FROM persons WHERE profession='teacher' GROUP BY profession
query().select().from(persons).where(isTeacher).groupBy(profession).execute();
Or with select():
function professionGroup(group) {
  return group[0];
}

//SELECT profession FROM persons GROUP BY profession
query().select(professionGroup).from(persons).groupBy(profession).execute(); //[
"teacher","scientific","politician"]
```

```
Another example:
function isEven(number) {
  return number % 2 === 0;
}

function parity(number) {
  return isEven(number) ? 'even' : 'odd';
}

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

//SELECT * FROM numbers
query().select().from(numbers).execute(); //[1, 2, 3, 4, 5, 6, 7, 8, 9]

//SELECT * FROM numbers GROUP BY parity
query().select().from(numbers).groupBy(parity).execute(); //[["odd",[1,3,5,7,9]]
,["even",[2,4,6,8]]]
Multilevel grouping:
function isPrime(number) {
  if (number < 2) {
    return false;
  }
  var divisor = 2;
  for(; number % divisor !== 0; divisor++);
  return divisor === number;
}

function prime(number) {
  return isPrime(number) ? 'prime' : 'divisible';
}

//SELECT * FROM numbers GROUP BY parity, isPrime
query().select().from(numbers).groupBy(parity, prime).execute(); // [["odd",[["d
ivisible",[1,9]],["prime",[3,5,7]]]],["even",[["prime",[2]],["divisible",[4,6,8]
]]]]
orderBy should be called after groupBy, so the values passed to orderBy function
 are the grouped results by the groupBy function.
Filter groups with having():
function odd(group) {
  return group[0] === 'odd';
}

//SELECT * FROM numbers GROUP BY parity HAVING odd(number) = true <- I know, thi
s is not a valid SQL statement, but you can understand what I am doing
query().select().from(numbers).groupBy(parity).having(odd).execute(); //[["odd",
[1,3,5,7,9]]]
You can order the results:
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

function descendentCompare(number1, number2) {
  return number2 - number1;
}

//SELECT * FROM numbers ORDER BY value DESC
 query().select().from(numbers).orderBy(descendentCompare).execute(); //[9,8,7,6
,5,4,3,2,1]
from() supports multiple collections:
var teachers = [
  {
    teacherId: '1',
    teacherName: 'Peter'
  },
  {
    teacherId: '2',
    teacherName: 'Anna'
  }
];


var students = [
  {
    studentName: 'Michael',
    tutor: '1'
  },
  {
    studentName: 'Rose',
    tutor: '2'
  }
];

function teacherJoin(join) {
  return join[0].teacherId === join[1].tutor;
}

function student(join) {
  return {studentName: join[1].studentName, teacherName: join[0].teacherName};
}

//SELECT studentName, teacherName FROM teachers, students WHERE teachers.teacher
Id = students.tutor
query().select(student).from(teachers, students).where(teacherJoin).execute(); /
/[{"studentName":"Michael","teacherName":"Peter"},{"studentName":"Rose","teacher
Name":"Anna"}]
Finally, where() and having() admit multiple AND and OR filters:
function tutor1(join) {
  return join[1].tutor === "1";
}

//SELECT studentName, teacherName FROM teachers, students WHERE teachers.teacher
Id = students.tutor AND tutor = 1
query().select(student).from(teachers, students).where(teacherJoin).where(tutor1
).execute(); //[{"studentName":"Michael","teacherName":"Peter"}] <- AND filter

var numbers = [1, 2, 3, 4, 5, 7];

function lessThan3(number) {
  return number < 3;
}

function greaterThan4(number) {
  return number > 4;
}

//SELECT * FROM number WHERE number < 3 OR number > 4
query().select().from(numbers).where(lessThan3, greaterThan4).execute(); //[1, 2
, 5, 7] <- OR filter

var numbers = [1, 2, 1, 3, 5, 6, 1, 2, 5, 6];

function greatThan1(group) {
  return group[1].length > 1;
}

function isPair(group) {
  return group[0] % 2 === 0;
}

function id(value) {
  return value;
}
```

```
function frequency(group) {
  return { value: group[0], frequency: group[1].length };
}

//SELECT number, count(number) FROM numbers GROUP BY number HAVING count(number)
 > 1 AND isPair(number)
query().select(frequency).from(numbers).groupBy(id).having(greatThan1).having(is
Pair).execute(); // [{"value":2,"frequency":2},{"value":6,"frequency":2}])
```

36. Can you get the loop ?

You are given a node that is the beginning of a linked list. This list always contains a tail and a loop.
Your objective is to determine the length of the loop.
For example in the following picture the tail's size is 3 and the loop size is 11.

```
# Use the `next' method to get the following node.

node.next
// Use the `getNext' method or 'next' property to get the following node.

node.getNext()
node.next
# Use the `next' attribute to get the following node

node.next
// Use the `getNext()` method to get the following node.

node.getNext()
-- use the `next :: Node a -> Node a` function to get the following node
# Use the `next' method to get the following node.

node.next
```

Note: do NOT mutate the nodes!

Thanks to shadchnev, I broke all of the methods from the Hash class.

Don't miss dmitry's article in the discussion after you pass the Kata !!