Exercise : Exercise - Create a calculator

Define a method, which would evaluate basic arithmetic operations.
Example -
calculate 3, :+, 2

```
[input]
"3,:+,2"
[/input]
[output]
5
[/output]
```

Exercise : Exercise - Program from command line

1. Define a class dynamically. Class name should be taken from user by standard input(command line)
2. Then prompt user for a method name and a single line of code. This method should be defined as instance method in the class above dynamically with the code entered by user.
3. Tell user that the class and method is defined.
4. Then call this instance method and display the result

Ex.

```
Please enter the class name: User
Please enter the method name you wish to define: greet
Please enter the method's code: "Welcome from 
#{self.class} class. I am #{self}"

--- Result ---
```

```
Hello, Your class User with method greet is ready.
Calling: User.new.greet:
"Welcome from User class. I am <User#123456>”
```

Code should be object oriented. Calling api should be like this:

```
my_class = DynamicClass.new(class_name)
my_class.def_method(method_name, method_body)
my_class.def_method(another_method_name, method_body)
...
...

my_class.call(method_name)
my_class.call(another_method_name)
```

Exercise : Exercise - Interactive program

Write a program interactively, through the command line
Enter a blank line to evaluate.
Enter 'q' to quit.
Note: use binding

Exercise : Exercise - Shopping list

Write a simple DSL for creating a shopping list. We should be able to specify the item name and quantity..
Something like.

```
sl = ShoppingList.new
```

```
sl.items do
  add("Toothpaste",2)
  add("Computer",1)
```

Exercise : Simple : Singleton Method

Create an instance of a class string.
Define a method on this instance in a way that it is available only on this instance. If you create another instance of the class and try to call this method, it should give a method not defined error.
Create this method using
def something.method_name and also class << self

Exercise : Exercise - Dynamic Method Calling

Derive a class from String which defines a few methods on Strings, eg exclude? (opp of include). Also few methods which accepts arguments(required + optional).
From the command line,
* Create an object by asking user an input string of the new class
* Display all the method to user you defined (not defined by string or superclass)
* Prompt user for a method name to call on the object
* Ask any required/optional argument required for this

method
* Once the user enters the method name and arguments, execute it and display the results on the command line.

Exercise : Exercise: Creating Classes from CSV

Read a csv format file and construct a new class with the name of the file dynamically. So if the csv is persons.csv, the ruby class should be person, if it's places.csv, the ruby class should be places
Also create methods for reading and displaying each value in "csv" file and values in first row of csv file will act as name of the function.
Construct an array of objects and associate each object with the row of a csv file.
For example the content of the csv file could be
name,age,city
gaurav,23,karnal
vilok,23,hissar

Exercise : Exercise - MyObjectStore

Create a module "MyObjectStore"

1) When included in a class, on being called save, should save the object(to any data structure) - maybe a hash or array (non-persistent, lives in memory for the life of the program)
2) If a validate function is defined in the class, the object should only be saved if the validate condition is satisfied, else appropriate error message should be printed and added to objects.
3) The object should also be able to call the validate function directly
4) Should define memory efficient dynamic finders (for e.g.,

find_by_name("abc"), find_by_age("23"))
5) The result of dynamic finders should behave like Enumerators.
For example:-

```
class Play
  include MyObjectStore

  attr_accessor :age :fname, :email

  validate_presence_of :fname, :mail

  def validate
    ....
  end
end
p2 = Play.new;
p2.fname = "abc" ;
p2.lname = "def" ;
p2.save ;

Play.find_by_fname("xyz") ;
Play.find_by_email("abcd") ;
Play.collect ;
Play.count
#These should return all the objects satisfying the
condition
```

Exercise : Exercise - aliasing

Create a module (lets say 'MyModule', which adds a method (lets say, 'chained_aliasing')) by including which we can mimic the following behavior -

To understand how it is useful, look at the following example:

Before including module

```ruby
class Hello
   def greet
     puts 'hello'
   end
end

say = Hello.new
say.greet # =>  hello
```

After including module -

Now suppose we want to wrap logging behavior around `Hello#greet()`. So define a module MyModule which we can include/extend in our class to provide a macro chained_aliasing and define a method greet_with_logger. Like:

```ruby
        # reopening Hello class
    class Hello
      include/extend MyModule

      def greet_with_logger
        puts '--logging start'
        greet_without_logger
        puts "--logging end"
      end

      chained_aliasing :greet, :logger
    end

    say = Hello.new
    say.greet

    # --logging start
    # hello
    # --logging end


    say.greet_with_logger
    # --logging start
    # hello
    # --logging end
```

```
    say.greet_without_logger
    # hello
```

Method names with exclamations should be reserved. Example - method name - foo?, aliased method name - foo_with_bar?

Method scopes should also be preserved for public, private and protected methods. ie, if greet is private, greet_with_logger, greet_without_logger and new greet should all be private.

Exercise : Class accessors

Write a macro `cattr_accessor` that defines both class and instance accessors for class attributes. It works similar to how attr_accessor works in ruby but for class methods.

For eg.

```
class Person
  cattr_accessor :hair_colors
end
```

```
Person.hair_colors = [:brown, :black, :blonde, :red]
Person.hair_colors   # => [:brown, :black, :blonde, :red]
```

You can pass following boolean options to cattr_accessor: instance_writer, instance_reader, instance_accessor
They tell whether the appropriate reader/writer method needs to be created or not. By default, all will be true.
If both instance_accessor and instance_writer/instance_reader options are passed, it throws an error.

for ex

```
cattr_accessor :hair_colors, instance_writer: false, 
instance_reader: true
# OR
cattr_accessor :hair_colors, :address, 
instance_accessors: false
```

Also keep in mind that if a subclass changes the value then that would also change the value for parent class. Similarly if parent class changes the value then that would change the value of subclasses too.

For eg:

```
class Male < Person
end

Male.hair_colors << :blue
Person.hair_colors # => 
[:brown, :black, :blonde, :red, :blue]
```

If I have specified instance_writer/reader true, then we should be able to access it on instance too.

```
Male.new.hair_colors # => […….]
```

Exercise : Implement Dirty objects

Define a module DirtyObject which when included in a class, provides following functionality:

```
class User
  include DirtyObject

  attr_accessor :name, :age, :email
  define_dirty_attributes :name, :age
```

```
end
```
Now this User class would behave like this:

```
u = User.new

u.name  = 'Akhil'
u.email = 'akhil@vinsol.com'
u.age   = 30

u.changed? #=> true
u.changes  #=> { name: [nil, 'Akhil], age: [nil, 30] }

u.name_was    #=> nil
u.email_was   #=> undefined method.....
u.age_was     #=> nil

u.save        #=> true


u.changed?    #=> false
u.changes     #=> {}

u.name = 'New name'
u.age  = 31
u.changes    #=> {name: ['Akhil', 'New name'], age: [30, 
31]}
u.name_was   #=> 'Akhil'

u.name = 'Akhil'
u.changes    #=> {age: [30, 31]}
u.changed?   #=> true

u.age = 30
u.changes    #=> {}
u.changed?   #=> false
```