Задание №21
Простое моделирование процессов

1. Общая постановка задачи

Выполните задание, соответствующее вашему варианту.

Наличие в решении указанных в задании классов, их элементов и методов обязательно. Решение может быть засчитано только если задание выполнено на 100%.

Кроме перечисленных в задании элементов разрешается вводить в решение любое количество дополнительных классов и методов на усмотрение студента, при условии, что все добавленные элементы служат для решения поставленной задачи, а не для расширения функциональности реализованного класса.

Все сформулированные подзадачи по накоплению статистики, сумм и т.п. должны быть реализованы в рамках описанных в задании классов.

Опишите набор примеров, демонстрирующих все аспекты работы вашей программы.

Код программы должен быть корректно отформатирован и прокомментирован. При наличии небрежностей форматирования и/или отсутствии комментариев преподаватель оставляет за собой право не проверять решение и оценить его в 0 баллов.

Файлу с программой дайте имя `task21-NN.rb`, где вместо `NN` — номер вашего варианта. Полученный файл загрузите на портал в качестве решения задания.

2. Ограничения

В решении не следует использовать конструкции цикла `while` и `until`.

Не следует делать предположений насчёт задания, не сформулированных явно в условии. Если возникают сомнения — задайте вопрос на форуме «Язык Ruby».

3. Пример выполнения задания

0. Музыкальный автомат содержит некоторое количество музыкальных треков, которые он может проигрывать. Среди музыкальных треков выделяются треки классической музыки.

Реализуйте классы `MusicTrack`, `ClassicalMusicTrack` и `MusicMachine`.

Объекты классов `MusicTrack` и `ClassicalMusicTrack` соответствуют музыкальным трекам вообще и музыкальным трекам классической музыки в частности. Создать музыкальный трек — определить для него исполнителя (строка), наименование (строка) и длительность (число), а для трека классической музыки указать ещё и композитора. Для объектов этих классов должна быть определена процедура инициализации от необходимых параметров. Классы должны реализовывать учёт статистики проигрывания созданных музыкальных треков для дальнейшего формирования хит-листа из 10 наиболее проигрываемых треков, а так же подсчитываться суммарное время проигрыша треков на всех автоматах.

Должны быть определены методы:

- `play` для моделирования проигрывания трека каким-то автоматом: в методе должна учитываться соответствующая статистика. Метод должен возвращать сам объект, для которого он вызван;
- `to_s` — строковое представление трека, содержащее полную информацию о нем;
- `time` — геттер для длительности трека.

Кроме того должны быть определены методы класса

- `reset` для обнуления статистики по проигранным трекам;
- `top10` — метод выводящий хит-лист из 10 наиболее проигрываемых треков в соответствии с накопленной статистикой;
- `time_stat` — геттер для суммарного времени проигранных треков.

Класс `MusicMachine` должен моделировать работу музыкального автомата. В классе должна быть определена константа `MaxTrackCount` — максимальное количество треков, которое может быть загружено в автомат. При инициализации автомат может получать массив, содержащий треки.

Должны быть реализованы следующие методы:

- `empty` — опустошить автомат: удалить все треки из него. Метод возвращает сам автомат.
- `add_track(track)` — добавить в автомат трек `track`. Метод возвращает сам автомат.
- `play numbers` — поставить в очередь для проигрывания список треков, определённый массивом порядковых номеров треков `numbers` в автомате (нумерация от нуля в том порядке, в котором треки добавлялись в автомат). Если указанный в списке номер отсутствует в автомате, то он игнорируется. Метод возвращает сам автомат.
- `pass` — пропустить первый трек в очереди для проигрывания (убрать его из головы очереди). Метод возвращает сам автомат.
- `pass` — проиграть первый трек в очереди для проигрывания (при этом удалить его из головы очереди). Если очередь пуста, то проиграть случайный трек из автомата, а если в автомат не загружено ни одного трека, то ничего не делать. Метод возвращает сам автомат.
- `count_time_to_play` — посчитать суммарное время треков, оставшихся в очереди для проигрывания.
- `all_print` — вывести список треков, загруженных в автомат с их порядковыми номерами (нумерация с 0). Результат выполнения метода — `nil`.
- `queue_print` — вывести список треков, находящихся в очереди для проигрывания с их порядковыми номерами (нумерация с 1). Результат выполнения метода — `nil`.

РЕШЕНИЕ:

Содержимое файла `task21-00.rb`:

```ruby
################################################################################
class MusicTrack
  # очистка статистики
  def self.reset
    @@track_stat = {} # хеш для статистики проигрывания треков
    @@time_stat = 0   # суммарное время проигранных треков
  end

  # геттер для хеша статистики треков
  def self.time_stat
    @@time_stat
  end

  # при инициализации (переопределнии) класса очищаем
  # (или инициализируем) статистику
  self.reset
```

```ruby
  # объявляем геттер для времени трека
  attr_reader :time

  # конструктор трека
  # исполнитель, название и время трека – обязательные параметры
  def initialize artist, title, time
    @artist = artist
    @title = title
    @time = time
    # ключ для хеша – строка из всех составляющих трека
    # это же представление будем выдавать в качестве строкового представления
    # в методе to_s
    @hash_key = "\"#{@title}\", #{@artist}, #{@time}"
  end

  # "проигрывание" трека
  # увеличивается соответствующая статистика
  def play
    if @@track_stat[@hash_key]
      @@track_stat[@hash_key] += 1
    else
      @@track_stat[@hash_key] = 1
    end
    @@time_stat += @time
    self
  end

  # вывод информации о 10 наиболее популярных треках
  def self.top10
    # сначала превращаем хеш со статистикой в массив, сортируем его
    # в порядке убывания количества раз проигрываний
    top = @@track_stat.to_a.sort {|x, y| y[1] <=> x[1]} [0..9]
    top.each_with_index do |track, i|
      # выводить будем порядковый номер и информацию о треке
      puts "#{i + 1}. #{track[0]}"
    end
  end

  # метод трансформации трека в строку
  def to_s
    @hash_key
  end

end
################################################################################

################################################################################
class ClassicalMusicTrack < MusicTrack
  # очистка статистики
  def self.reset
    @@classical_stat = {}
  end

  # при инициализации (переопределнии) класса очищаем
  # (или инициализируем) статистику
  self.reset

  # конструктор трека
  # композитор, исполнитель, название и время трека – обязательные параметры
  def initialize composer, artist, title, time
    # инициализируем трек как обычный
    super artist, title, time
    # затем инициализируем параметр для композитора
    @composer = composer
    # добавляем композитора к строковому представлению трека
    @hash_key = "#{@composer}, " + @hash_key
  end

  # "проигрывание" трека
  # увеличивается соответствующая статистика
  def play
    if @@classical_stat[@hash_key]
      @@classical_stat[@hash_key] += 1
    else
      @@classical_stat[@hash_key] = 1
    end
```

```ruby
    super
  end

  # вывод информации о 10 наиболее популярных треках
  def self.top10
    top = @@classical_stat.to_a.sort {|x, y| y[1] <=> x[1]} [0..9]
    top.each_with_index do |track, i|
      puts "#{i + 1}. #{track[0]}"
    end
  end

end
################################################################################

################################################################################
class MusicMachine
  def initialize(tracks = [])
    @track_count = 0  # количество треков в автомате
    @tracks = []      # массив треков в автомате
    @play_queue = []  # очередь номеров треков для проигрывания
    # инициализация массива треков
    tracks.each { |a| add_track(a) }
  end

  # максимальное количество треков в автомате
  MaxTrackCount = 15

  # метод добавления списка номеров треков в очередь для проигрывания
  def ask_to_play nums
    nums.each do |num|
      if @tracks[num]          # если есть в базе трек с номером
        @play_queue.push(num)  # то его можно проиграть
      end
    end
    self
  end

  # пропуск трека в очереди без его проигрывания
  def pass
    @play_queue.shift
    self
  end

  # проигрывание трека из очереди
  def play
    if @play_queue.empty? # если очередь пуста
      if @tracks.empty?   # и в автомате нет ни одного трека
        return self       # то ничего не делать
      else
        num = rand(@tracks.length) # иначе выбрать случайный трек в машине
      end
    else                      # если очередь не пуста
      num = @play_queue.shift # выбрать трек первый из очереди
    end
    puts @tracks[num]  # вывести наименование выбранного трека на экран
    @tracks[num].play  # проиграть выбранный трек
    self
  end

  # вычисление суммарного времени треков в текущей очереди
  def count_time_to_play
    # суммируем в аккумулятор все длительности треков в очереди
    @play_queue.inject(0) { |acc, num| acc + @tracks[num].time }
  end

  # добавить трек в машину
  def add_track(track)
    # если максимальное количество треков не превышено
    if @track_count < MaxTrackCount
        @tracks[@track_count] = track   # добавляем трек в массив
        @track_count += 1
    end
    self
  end

  # очистить "память" машины
```

```ruby
  def empty
    @track_count = 0
    @tracks = []
    @play_queue = []
    self
  end

  # вывести очередь треков
  def queue_print
    @play_queue.each_with_index do |num, i|
      puts "#{i + 1}, #{@tracks[num]}"
    end
    nil
  end

  # вывести список треков автомата
  def all_print
    @tracks.each_with_index do |track, i|
      puts "#{i}, #{track}"
    end
    nil
  end
end
################################################################################

################################################################################
##    ПРИМЕРЫ
################################################################################
# Создадим массив разных треков для того, чтобы использовать их впоследствии
@a = [ MusicTrack.new("Grimes", "We Appreciate Power", 5.3),
       MusicTrack.new("Azealia Banks", "212", 3.3),
       MusicTrack.new("The Chemical Brothers", "No Geography", 3.2),
       MusicTrack.new("The Chemical Brothers", "Free Yourself", 4.3),
       MusicTrack.new("Soulwax", "NY Lipps", 5.6),
       MusicTrack.new("Soulwax", "E Talking", 6.1),
       MusicTrack.new("Shit Robot", "I Got a Feeling", 8),
       MusicTrack.new("Shit Robot", "Losing My Patience", 4.3),
       MusicTrack.new("Shit Robot", "Grim Receiver", 8.5),
       MusicTrack.new("Pink Martini", "Una Notte a Napoli", 4.7),
       MusicTrack.new("Vitalic", "Poison Lips", 3.8),
       MusicTrack.new("Nils Frahm", "A Place", 7.1),
       MusicTrack.new("Nils Frahm", "All Melody", 9.5),
       MusicTrack.new("Nils Frahm", "Momentum", 5.3),
       MusicTrack.new("Dee-Lite", "Grove Is In The Heart", 3.9),
       MusicTrack.new("Leftfield", "Afro Left", 7.5),
       MusicTrack.new("Jimi Jules", "Too Far", 4.6),
       MusicTrack.new("Jimi Jules", "Clinomania", 4.5),
       MusicTrack.new("Jimi Jules", "Read", 4.8),
       ClassicalMusicTrack.new("Wagner", "Liselotte Rebmann et al", "Hojotoho! Heiaha!", 8.1),
       ClassicalMusicTrack.new("Wagner", "Sofia Radio Symphony Orchestra", "Siegfried's Death and Funeral March", 5.9),
       ClassicalMusicTrack.new("Beethoven", "Magyar Szimfonikus Zenekar Budapest Meisterwerke der Klassischen Music",
       ↪  "Fidelio. Act I, Scene 6: Marsch", 2.3),
       ClassicalMusicTrack.new("Beethoven", "Bernhard Jarvis", "Für Elise", 3.6),
       ClassicalMusicTrack.new("Schnittke", "Russian State Symphonic Cappella", "Concerto for Piano and Strings", 24.5),
       ClassicalMusicTrack.new("Strauss, R.", "Cologne Radio Symphony Orchestra", "Elektra's Dance", 1.2),
       ClassicalMusicTrack.new("Glass", "Stuttgart State Opera Orchestra and Chorus", "Akhnaten, Act II Scene 1:  Temple",
       ↪  12.9),
       ClassicalMusicTrack.new("Shostakovich", "Bavarian Radio Symphony Orchestra", "Cello Concerto No. 1, III. Cadenza",
       ↪  6.1),
       ClassicalMusicTrack.new("Berlioz", "Orchestre philharmonique de Strasbourg", "Nuit d'ivresse et d'extase infinie !",
       ↪  4.8),
       ClassicalMusicTrack.new("Wagner", "Berlin Philharmonic", "Siegfried / Erster Aufzug: Vorspiel", 4.8),
       ClassicalMusicTrack.new("Strauss, R.", "Vienna Philharmonic", "Salome's Dance of the Seven Veils", 8.6),
       ClassicalMusicTrack.new("Straus O.", "Wiener Staatsoper", "Ein Walzertraum", 38.7),
       ClassicalMusicTrack.new("Desyatnikov", "Orchestra of the Bolshoi Theatre", "Deti Rozentalya: Petrusha, ti?", 4.0) ]



# Создаем новые автоматы
# каждому из них передаем массив из 20 случайных элементов массива @a
@m1 = MusicMachine.new(@a.sample 20)
@m2 = MusicMachine.new(@a.sample 20)
@m3 = MusicMachine.new(@a.sample 20)
@m4 = MusicMachine.new(@a.sample 20)
@m5 = MusicMachine.new(@a.sample 20)
```

```ruby
# для примера выведем список всех треков в автомате M5
puts "================================="
puts "M5"
@m5.all_print

# создадим очередь в каждом автомате. Для этого будем генерировать
# массив из случайных чисел от 0 до 20 и будем пытаться сформировать очередь
# из треков с такими номерами в автомате
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))
@m1.ask_to_play((0..20).to_a.sample(10))

puts "================================="
puts "M1 queue"
@m1.queue_print
puts "Full time to play #{@m1.count_time_to_play}"

@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))
@m2.ask_to_play((0..20).to_a.sample(10))

puts "================================="
puts "M2 queue"
@m2.queue_print
puts "Full time to play #{@m2.count_time_to_play}"

@m3.ask_to_play((0..20).to_a.sample(10))
@m3.ask_to_play((0..20).to_a.sample(10))
@m3.ask_to_play((0..20).to_a.sample(10))
@m3.ask_to_play((0..20).to_a.sample(10))

puts "================================="
puts "M3 queue"
@m3.queue_print
puts "Full time to play #{@m3.count_time_to_play}"

@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))
@m4.ask_to_play((0..20).to_a.sample(10))

puts "================================="
puts "M4 queue"
@m4.queue_print
puts "Full time to play #{@m4.count_time_to_play}"

@m5.ask_to_play((0..20).to_a.sample(10))
@m5.ask_to_play((0..20).to_a.sample(10))
@m5.ask_to_play((0..20).to_a.sample(10))
@m5.ask_to_play((0..20).to_a.sample(10))

puts "================================="
puts "M5 queue"
@m5.queue_print
puts "Full time to play #{@m5.count_time_to_play}"

# имитируем проигрывание треков в автомате
puts "================================="
puts "M1"
```

```ruby
@m1.play.pass.play.play.play.play.pass.play.play.pass
@m1.play.pass.play.play.play.play.pass.play.play.pass
@m1.play.pass.play.play.play.play.pass.play.play.pass
@m1.play.pass.play.play.play.play.pass.play.play.pass
@m1.play.pass.play.play.play.play.pass.play.play.pass
@m1.play.pass.play.play.play.play.pass.play.play.pass

puts "================================="
puts "M2"
@m2.play.pass.play.play.play.play.pass.play.play.pass
@m2.play.pass.play.play.play.play.pass.play.play.pass
@m2.play.pass.play.play.play.play.pass.play.play.pass
@m2.play.pass.play.play.play.play.pass.play.play.pass
@m2.play.pass.play.play.play.play.pass.play.play.pass
@m2.play.pass.play.play.play.play.pass.play.play.pass

puts "================================="
puts "M3"
@m3.play.pass.play.play.play.play.pass.play.play.pass
@m3.play.pass.play.play.play.play.pass.play.play.pass
@m3.play.pass.play.play.play.play.pass.play.play.pass
@m3.play.pass.play.play.play.play.pass.play.play.pass
@m3.play.pass.play.play.play.play.pass.play.play.pass
@m3.play.pass.play.play.play.play.pass.play.play.pass

puts "================================="
puts "M4"
@m4.play.pass.play.play.play.play.pass.play.play.pass
@m4.play.pass.play.play.play.play.pass.play.play.pass
@m4.play.pass.play.play.play.play.pass.play.play.pass
@m4.play.pass.play.play.play.play.pass.play.play.pass
@m4.play.pass.play.play.play.play.pass.play.play.pass
@m4.play.pass.play.play.play.play.pass.play.play.pass

puts "================================="
puts "M5"
@m5.play.pass.play.play.play.play.pass.play.play.pass
@m5.play.pass.play.play.play.play.pass.play.play.pass
@m5.play.pass.play.play.play.play.pass.play.play.pass
@m5.play.pass.play.play.play.play.pass.play.play.pass
@m5.play.pass.play.play.play.play.pass.play.play.pass
@m5.play.pass.play.play.play.play.pass.play.play.pass

# Хит-парад
puts "================================="
puts "Top 10"
MusicTrack.top10

puts "================================="
puts "Top Classical 10"
ClassicalMusicTrack.top10

# Общее время проигрыша во всех автоматах
puts MusicTrack.time_stat
puts "M1 Full time to play #{@m1.count_time_to_play}"
puts "M2 Full time to play #{@m2.count_time_to_play}"
puts "M3 Full time to play #{@m3.count_time_to_play}"
puts "M4 Full time to play #{@m4.count_time_to_play}"
puts "M5 Full time to play #{@m5.count_time_to_play}"
```

Файл с примером можно загрузить с портала.

4. Варианты заданий

1. Пункт обмена валют обслуживает клиентов по разным схемам обмена. Обмен может быть прямой (по установленному курсу), с наценкой за обмен, с наценкой и с начислением налога.

При прямом обмене обмениваемая сумма просто меняется на сумму в заданной валюте по курсу. Процент наценки начисляется на обмениваемую сумму — обмену подлежит сумма за вычетом наценки, а сумма наценки идёт в доход обменного пункта. Налог начисляется на сумму выплаты клиенту (суммы, на которую происходит обмен), вычитается из неё, и накапливается для дальнейшей выплаты налоговым органам.

Считаем что обмен безналичный и может быть обменяна произвольная положительная сумма с точностью до любого знака.

Реализуйте классы `DirectExchange`, `ExchangeWithCharge`, `ExchangeWithChargeAndTax` и `Client`

Объекты классов `DirectExchange`, `ExchangeWithCharge` и `ExchangeWithChargeAndTax` являются вариантами схем обмена — прямого, с наценкой и с наценкой и налогом. Определить новую схему прямого обмена — указать код валюты, получаемой от клиента, код валюты, отдаваемой взамен, курс обмена (числовое значение, умножение которого на сумму, получаемую от клиента, приведёт к получению суммы, подлежащей выдаче клиенту взамен). Для определения схемы с наценкой необходимо передать конструктору ещё и процент наценки (значение от 0 до 1), а при схеме с налогом ещё и процент налога (значение от 0 до 1).

Классы должны накапливать статистику по обменным операциям, доходу пункта обмены и накопленной сумме налога.

Должны быть определены геттеры для параметров схемы обмена, а также методы:

- `change(s)` — обменять сумму `s` по данной схеме обмена. Результат метода — список из двух элементов, где первый элемент — код валюты на которую произведён обмен, а второй — сумма к выдаче после обмена.
- `to_s` — строковое представление, включающее все параметры схемы обмена.
- `stat` — для класса `DirectExchange` метод должен выводить информацию о всех произведённых обменных операциях; `ExchangeWithCharge` метод должен выводить информацию о всех произведённых обменных операциях, приводящих к начислению наценки. Результат метода — `nil`.

Должны быть определены методы класса:

- `stat` — для класса `DirectExchange` метод должен выводить информацию о всех произведённых обменных операциях; для класса `ExchangeWithCharge` метод должен выводить информацию о всех произведённых обменных операциях, приводящих к начислению наценки; для класса `ExchangeWithChargeAndTax` метод должен выводить информацию о всех произведённых обменных операциях, приводящих к начислению налога. Результат метода — `nil`;
- `income` — метод должен выдавать информацию о суммарном доходе обменного пункта по каждой из валют (предполагается что доход сохраняется в той же валюте, в которой начисляется). Результат метода — `nil`;
- `tax` — метод должен выдавать информацию о суммарном начисленном налоге по каждой из валют (предполагается что налоги платятся в той же валюте, в которой начисляются). Результат метода — `nil`.

Объекты класса `Client` соответствуют пользователям системы. При создании пользователя указывается только его имя.

Объект должен накапливать информацию, по обменным операциям, совершенных клиентом.

Должны быть определены геттеры для параметров пользователя, а также методы:

- `income(currency, s)` — метод, выделяющий клиенту сумму дохода `s` в валюте `currency`. Результат метода — сам клиент.
- `change(s, scheme)` — обменять сумму `s` по схеме обмена `scheme`. Метод должен осуществлять обмен при условии, что у клиента имеется заданная сумма в валюте, определённой в схеме обмена. В случае успешного обмена (у клиента достаточно средств) у клиента должна уменьшиться сумма средств в заданной валюте и увеличиться сумма средств в валюте, возвращаемой обменным пунктом. При этом на экране должно быть выведено сообщение об обмене, содержащее имя клиента, сумму и валюту обмена и сумму и валюту, на которую произведён обмен. Если у клиента средств недостаточно, то обмен не производится, а на экране выводится сообщение, содержащее имя клиента, сумму и валюту обмена и информацию, что обмен не состоялся. Результат метода — сам клиент.
- `to_s` — строковое представление имени клиента.
- `stat` — метод должен выводить всю информацию об успешно совершённых обменных операциях пользователя. Результат метода — `nil`.
- `status` — метод должен выдавать информацию о сумме средств клиента по каждой из валют. Результат метода — `nil`;

2 (Бонус 10%)**.** Отделения банка обслуживают счета клиентов. Клиент может обратиться в отделение банка для открытия счета: счёт может быть кредитным или депозитным. После открытия счёта клиент может работать с ним: он может снимать со счета и класть на счёт некоторые суммы. Для открытия депозитного счета клиенту на него следует положить начальную сумму. Для открытия кредитного счета клиенту нужно просто обратиться в отделение банка.

В банке индивидуальный подход к каждому клиенту, поэтому величина процента начисления на депозитном счёте и величина процента по кредиту определяется при открытии счета.

При окончании периода (условного года) происходят начисления процентов на депозитный счёт и начисление процентов по кредиту.

В отношении депозитного счета: клиент может класть на него произвольные суммы; снимать со счета клиент может только суммы, не превышающие общей суммы счета. Проценты по депозиту начисляются на сумму, находящуюся на счёте на момент окончания предыдущего периода, с добавленными к ней процентами, начисленными на момент окончания предыдущего периода, за вычетом суммы, снятой с момента окончания предыдущего периода.

Проценты по кредиту начисляются на общую сумму кредита на момент окончания периода. Проценты, начисленные по кредиту за период, должны быть погашены в течении следующего периода (до закрытия периода). Если до после начисления процентов они не были погашены до окончания следующего периода, то счёт начинает работать только на внесение денег до тех пор, пока не будут погашены все начисленные проценты по кредиту. Если на окончание периода остались непогашенные проценты, то они считаются суммой кредита и на них начисляются проценты.

Реализуйте классы `BankAccount`, `DepositAccount`, `CreditAccount` и `Client`.

Объекты классов `BankAccount`, `DepositAccount`, `CreditAccount` соответствуют банковскому счёту вообще, депозитному и кредитному счёту в частности. Открыть счёт — определить для него номер (число), владельца (ссылка на клиента — владельца счёта) и процент. Для объектов этих классов должна быть определена процедура инициализации от этих аргументов. Для объекта класса `DepositAccount` при инициализации должна быть ещё задана сумма первоначального взноса на счёт.

Объекты классов должны сохранять информацию о всех операциях по счёту.

Классы должны реализовывать учёт состояния всех счетов.

Должны быть определены методы:

- `deposit(s)` — метод для внесения суммы `s` на счёт. На счёт может быть внесена любая сумма. При внесении суммы на кредитный счёт сначала гасятся начисленные проценты по кредиту, а затем сама сумма кредита. После полного погашения процентов по кредиту, происходит разблокирование счета для снятия наличных. Клиент может положить деньги на счет при условии, что они у него имеются. Если операция прошла успешно, то сумма наличных денег у клиента должна уменьшиться. Результат метода — сам счёт;
- `withdraw(s)` — метод для снятия суммы `s` со счёта. Сумму с кредитного счёта можно снять только при условии отсутствия непогашенных процентов по кредиту, начисленных за позапрошлый период. Если операция снятия денег со счета прошла успешно, то у клиента должна увеличится наличность на соответствующую сумму. Результат метода — сам счёт;
- `close_period` — провести перерасчёты по счёту, соответствующие окончанию текущего периода;
- `to_s` — строковое представление счета. Должно включать в себя номер счета, имя владельца, общую сумму на счёте, а в случае блокировки кредитного счета на снятие наличных — отметку об этом. Результат метода — `nil`;
- `stat` — метод должен выводить информацию о движении средств (о всех операциях) по счёту. Результат метода — `nil`.

Кроме того должны быть определены методы класса

- `stat` — выдать информацию о всех счетах всех клиентов.
- `close_period` — закрыть период: провести процедуру закрытия периода по всем открытым счетам.

Класс `Client` должен имитировать клиента. Процедуре инициализации объекта данного класса передаётся имя клиента и, возможно, начальная сумма его наличности. Объект должен собирать информацию о всех открытых счетах клиента.

Должны быть определены следующие методы:

- `salary(s)` — метод для увеличения наличности клиента на сумму `s`. Результат метода — сам клиент.
- `name` — геттер для имени клиента.
- `to_s` — метод возвращающий строку — имя клиента с суммой его наличности.
- `stat` — выдать информацию о всех счетах клиента (в том же формате, что возвращается методом `to_s` для счета).

3 (Бонус 10%)**.** Зарегистрированные в системе обмена сообщениями пользователи могут приписать себя к одной или нескольким группам. Система обмена сообщениями позволяет пользователям посылать друг другу или целой группе пользователей короткие текстовые заметки. Сообщение характеризуется своим текстом, временем отправки, отправителем и получателем.

Реализуйте классы `Message`, `AlertMessage`, `User` и `Group`.

Объекты классов `Message` и `AlertMessage` являются сообщениями обычными и важными соответственно. Важные сообщения отличаются от обычных только способом их вывода на экране. Создать новое сообщение — сформулировать его текст (строковое значение) и назначить отправителя (отправителем является объект класса `User`) и адресата (объект класса `User` или `Group`). При создании сообщения фиксируется время его создания и оно сохраняется как параметр сообщения. Необходимо вести полный учёт созданных сообщений. Создание сообщения не эквивалентно его отправке: считаем, что создание сообщения инициируется пользователем, когда он его отправляет.

Должны быть определены геттеры для параметров сообщения, а также методы:

- `to_s` — строковое представление, включающее все параметры сообщения: время создания, от кого, кому (имя пользователя или название группы) и текст в двойных кавычках. Для важных сообщений строковое представление должно выводиться (при отображении на экране) в три строки: строка восклицательных знаков — параметры сообщения — строка восклицательных знаков.

- `stat` — метод должен выводить все созданные сообщения на экран.

Объекты класса `User` соответствуют пользователям системы. Создать пользователя — указать его имя.

Должны быть определены геттеры для параметров пользователя, а также методы:

- `add(group)` — зарегистрироваться в группе `group`. Результат метода — сам пользователь.
- `remove(group)` — удалить регистрацию в группе `group`. Результат метода — сам пользователь.
- `send_message(user_group, message)` — отправить пользователю или группе пользователей `user_group` сообщение `message`. Здесь `user_group` — объект класса `User` или `Group`, `message` — строка. Метод должен создавать объект-сообщение и организовать регистрацию данного сообщения у каждого адресата в качестве полученного и у отправителя в качестве отправленного. При выполнении метода на экране должно быть выведено данное сообщение. Результат метода — сам пользователь.
- `get_message(message)` — принять сообщение `message`. Здесь `message` — объект сообщения. Метод должен регистрировать данное сообщение у пользователя. Результат метода — сам пользователь.
- `to_s` — строковое представление имени пользователя.
- `stat` — метод должен выводить всю переписку пользователя (полученные и отправленные сообщения), отсортированную по времени создания сообщений. Результат метода — `nil`.
- `stat_in` — метод должен выводить все полученные пользователем сообщения, отсортированные по времени создания сообщений. Результат метода — `nil`.
- `stat_out` — метод должен выводить все отправленные пользователем сообщения, отсортированные по времени создания сообщений. Результат метода — `nil`.

Кроме того должны быть определены методы класса

- `stat` — метод должен выводить всю переписку всех пользователей (все полученные и отправленные сообщения), отсортированную по времени создания сообщений. Результат метода — `nil`.
- `stat_in` — метод должен выводить все полученные всеми пользователями сообщения, отсортированные по времени создания сообщений. Результат метода — `nil`.
- `stat_out` — метод должен выводить все отправленные всеми пользователями сообщения, отсортированные по времени создания сообщений. Результат метода — `nil`.

Класс `Group` моделирует группы пользователей системы. Процедуре инициализации объекта данного класса передаётся название группы.

Должны быть определены следующие методы:

- `add(user)` — зарегистрировать в группе пользователя `user`. Результат метода — сама группа.
- `remove(user)` — удалить из группы пользователя `user`. Результат метода — сама группа.
- `get_message(message)` — принять сообщение `message`. Здесь `message` — объект сообщения. Метод должен регистрировать данное сообщение у каждого пользователя группы и в самой группе. Результат метода — сама группа.
- `to_s` — строковое представление названия группы.
- `stat` — метод должен выводить все полученные группой сообщения, отсортированные по времени создания сообщений. Результат метода — `nil`.

Для написания тестовых вызовов может быть полезным последовательность вызов `Time.now` — текущий момент времени и `Time.now.to_s` — текущий момент времени, преобразованный в строку.

4 (Бонус 20%)**.** В кофейном аппарате несколько ёмкостей, например, для кофе, чая, какао, сливок, лимона, сахара. В автомате устанавливается максимальное число устанавливаемых ёмкостей. Объем каждой ёмкости задаётся порциями. Все ёмкости рассчитаны на одинаковое количество порций. Вода как ингредиент присутствует в любом автомате. Вода тоже отсчитывается порциями, но поступает отдельно (можно считать, что вода поступает из ёмкости бесконечного объёма). В каждом аппарате имеющиеся ингредиенты перечисляются по порядковым номерам (отдельно в каждом автомате).

В аппарат загружается набор рецептов, которые он может готовить. Рецепт имеет название и представляет собой набор номеров используемых ингредиентов из имеющихся (их порядковых номеров), с указанием количества порций каждого из них.

Пользователь аппарата может делать заказ напитка, указав название рецепта. При заказе пользователь может, при желании, увеличить или уменьшить количество каждого из ингредиентов рецепта на одну порцию, добавить одну порцию лимона и/или сливок (даже если они не входит в рецепт, но при условии, что ёмкость с соответствующим ингредиентом присутствует в аппарате), добавить от одной до девяти порций сахара (если ёмкость с сахаром присутствует в аппарате), добавить любое количество воды. Аппарат готовит напиток, только если все ингредиенты рецепта после уточнений пользователя есть в наличии.

Реализуйте классы `Container`, `SugarContainer`, `LemonContainer`, `WaterContainer` и `CoffeeMachine`.

Объекты классов `Container`, `SugarContainer`, `LemonContainer` и `WaterContainer` соответствуют ёмкостям с ингредиентами вообще и с сахаром, лимоном и водой, в частности. Создать ингредиент — определить для него

наименование (строка при необходимости). Для объектов этих классов должна быть определена процедура инициализации. Классы должны реализовывать учёт статистики использования ингредиентов на всех аппаратах.

Должна быть определена константа класса `MaxVolume` — максимальное количество порций, которое можно загрузить в ёмкость.

Должны быть определены методы:

- `use(n)` для учёта расходования `n` единиц ингредиента в статистике;
- `count(m, p, u)` — метод для вычисления возможного количества ингредиента для использования в рецепте, если в рецепте указано `m` порций, в ёмкости имеется `p` порций, а пользователь хочет увеличить количество в рецепте на `u` порций.
- `to_s` — строковое представление ингредиента.

Кроме того должны быть определены методы класса

- `reset` для обнуления статистики по использованию ингредиентов;
- `stat` — выдать статистику использования ингредиентов в порядке убывания количества.

Класс `CoffeeMachine` должен моделировать работу кофейного аппарата.

Должна быть определена константа класса `MaxVolume` — максимальное количество порций, которое можно загрузить в ёмкость.

При инициализации аппарата в качестве параметров могут быть заданы параметры: массив ёмкостей и хеш рецептов (хеш хешей). Ключом в хеше рецептов является строка — название напитка, а каждым значением — хеш в котором каждому ингредиенту напитка сопоставляется его количество.

Должны быть определены следующие методы:

- `load` — заполнить на 100% все емкости аппарата. Обнулить статистику изготовления напитков в автомате. Результат — загруженный аппарат.
- `order(rec, changes)` — заказать напиток, где `rec` — строка, наименование рецепта, а `changes` — хеш, содержащий необходимые изменения в рецепте. Для любого изменения задаётся номер ингредиента в аппарате и целое число — количество порций, на которое нужно увеличить или уменьшить вхождение ингредиента в рецепт. Если указанный ингредиент (не являющийся сахаром и лимоном) отсутствует в рецепте, то он игнорируется. Если для ингредиента, отличного от сахара и/или воды, указано изменение больше `1`, то оно принимается равным `1` (для сахара величина не может быть больше `9`, в противном случае принимается `9`). Если для ингредиента (отличного от сахара) указано изменение меньше чем `-1`, то оно принимается равным `-1`. Сахар в рецепте разрешено уменьшать до нуля. Метод должен изменять состояние аппарата и выдавать `true`, если напиток успешно приготовлен или `nil`, если напиток приготовить нельзя.
- `status` — определить состояние аппарата. Метод должен выводить информацию о наименованиях и количестве ингредиентов, оставшихся в ёмкостях автомата.
- `stat` — выдать статистику о количестве выполненных рецептов со времени последней загрузки аппарата. Метод должен выводить на экран количество заказов по каждому рецепту после последней заправки аппарата ингредиентами (для рецептов, по которым были заказы).

5 (Бонус 50%)**.** Мобильный оператор предлагает клиентам (абонентам) несколько тарифов. Тариф определяет стоимость времени разговора и принцип вычисления стоимости. При создании абонентского счета (при подключении к сети) клиент получает номер, выбирает тариф и пополняет свой счёт.

При положительном балансе пользователь может совершать звонки другим абонентам. При успешном соединении начинается тарификация разговора по тарифу вызывающего абонента, а по окончании разговора плата за него снимается с его счёта. Платит за разговор только вызывающий абонент.

Абоненты могут находиться в состоянии «Busy» (занято) или «Ready» (свободно, готов к разговору). По умолчанию все абоненты находятся в состоянии «Ready».

Если клиент находится в состоянии «Ready», то он может инициировать звонок другому абоненту. Если клиент начинает вызов другого абонента, его телефон становится недоступным для вызовов других абонентов (переходит в состояние «Busy»). Если, при этом, вызываемый абонент тоже находится в состоянии «Busy», то звонок завершается и вызывающий абонент переходит в состояние «Ready». Если же вызываемый абонент находится в состоянии «Ready», то он переводится в состояние «Busy» и ожидается его ответная реакция: он может принять звонок и начать разговор (в этом случае оба абонента остаются в состоянии «Busy») или может отклонить звонок (в этом случае оба абонента должны перейти в состояние «Ready»).

Реализуйте классы `Tariff`, `PerMinuteTariff`, `ShortTalkTariff` и `Phone`.

Объекты классов `Tariff`, `PerMinuteTariff` и `ShortTalkTariff` соответствуют тарифу мобильного оператора вообще и двум конкретным тарифам в частности. Создать вариант тарифа — определить для него название и установить стоимость единицы времени разговора. Стандартный тариф предполагает посекундную тарификацию разговора и при его инициализации задаётся стоимость одной секунды. Тариф класса `PerMinuteTariff` предполагает поминутную тарификацию (время разговора округляется в большую сторону до целой минуты) и при инициализации тарифа задаётся стоимость минуты. Тариф `ShortTalkTariff` также предлагает поминутную тарификацию и стоимость минуты, но инициализация тарифа должна иметь дополнительный параметр — нетари-

фицируемое количество минут разговора: если пользователь совершает звонок, то с него снимается стоимость времени разговора, превышающего нетарифицируемое количество минут.

Должны быть определены геттеры для параметров тарифа, а также методы:

- `count(ph, start, end)` — произвести начисление по тарифу для абонента `ph`, при условии, что разговор начался в момент времени `start` и закончился в момент времени `end`. Предполагается, что момент времени задаётся целым числом таким образом, что разность `end − start` доставляет количество секунд разговора. Результат метода — сумма начислений за разговор по данному тарифу.
- `to_s` — строковое представление, включающее все параметры тарифа.
- `stat` — метод должен выводить статистику по тарифу: выводить список абонентов, подключавшихся к тарифу с указанием суммы начислений по тарифу на каждого абонента.

Кроме того должны быть определены методы класса

- `stat` — вывести информацию о состоянии подключений всех пользователей к тарифам с указанием общей суммы начислений по всем тарифам на каждого абонента.

Класс `Phone` имитирует абонента. Процедуре инициализации объекта данного класса передаётся имя клиента, тариф (объект) и, возможно, начальная сумма, которая кладётся ему на счёт. При инициализации клиенту присваивается номер

Должны быть определены следующие методы:

- `top_up(s)` — пополнить баланс абонента на сумму `s`. Результат метода — сам абонент.
- `call(ph1)` — Инициировать звонок абоненту `ph1`. Вызов метода должен выводить на экран сообщение в котором указывается время начала вызова, чей звонок, кому звонок и, в случае если вызываемый или вызывающий абонент занят, то сообщение об этом. Пример сообщения «1649386911: Call Ivan to Vasiliy, Ivan is Busy».
- `answer` — Ответить вызывающему абоненту (при условии, что имеется входящий неотвеченный вызов). С момента вызова этого метода должно начать отсчитываться время для тарификации. При успешном ответе на звонок на экран следует вывести сообщение об этом. В сообщении нужно указать время начала разговора, кому звонок, от кого звонок. Пример сообщения «1649386921: Answered Call Ivan to Vasiliy»
- `hang_up` — Завершить звонок / Отклонить вызов / Отменить звонок. Если завершается принятый вызов, то с момента вызова этого метода завершается время тарификации звонка. В ходе выполнения метода следует вывести сообщение, в котором указать время завершения звонка и кто завершил / отклонил / отменил звонок. Пример сообщения «1649386971: Ivan ended the Call Ivan to Vasiliy» или «1649386971: Vasiliy rejected the Call Ivan to Vasiliy»
- `name` — геттер для имени клиента.
- `number` — геттер для номера абонента.
- `balance` — геттер для баланса абонента.
- `to_s` — метод возвращающий строку — имя клиента с указанием его тарифа и баланса.
- `stat` — выдать информацию о всех состоявшихся звонках абонента с указанием абонента вызова, времени начала звонка, длительности разговора в секундах, тарифа, по которому рассчитывался звонок, суммы, начисленной за звонок.

Возможно, необходимо продумать набор дополнительных методов для эффективного моделирования указанного взаимодействия абонентов.

Для написания тестовых вызовов может быть полезным последовательность вызов `Time.now.to_i` — текущий момент времени, заданный в секундах, а также вызов `sleep(n)`, задающий паузу в выполнении программы длительностью `n` секунд.