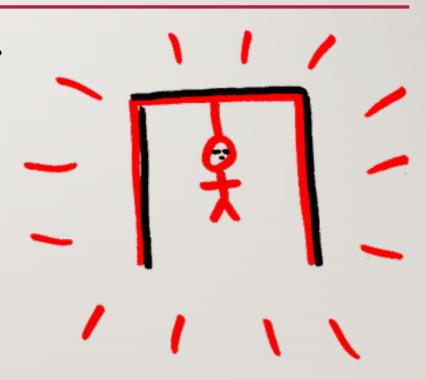
ЛЕКЦИЯ 12

ОБЪЕКТНАЯ «ВИСЕЛИЦА» V.2.0

ПЛАН ЗАНЯТИЯ

- Переделываем структуру программы «Виселица»
- Добавляем к выводу результата изображение виселиц
- Мы напишем вторую версию игры, разбив
 программу на два класса Game и ResultPrinter.
 Узнаем как работает оператор case в Ruby, как
 создавать поля класса (переменные экземпляра)
 и немного о спецсимволах и псевдографике.



• Удобство деления программы на классы легко продемонстрировать на таком примере: у топора есть древко, а есть металлическое лезвие. Если лезвие топора снять и закрепить на древке ударную часть молотка, получится молоток (замечание для плотников, которые будут читать этот текст: конечно, получится лишь жалкое подобие молотка, это просто для примера).



- Древко это экземпляр класса ручка, который может работать с разными экземплярами класса насадка: лезвие топора или ударная часть молотка.
- Вы, наверное, неоднократно видели отвёртки с набором насадок для различных шурупов и винтов.





- В такой ситуации можно сказать, что держатель отвёртки является экземпляром класса «основание», а каждая насадка экземпляром класса «насадка». Удобство в том, что класс «основание» ничего не знает о том, какие болты будут с его помощью закручивать, зато он может уметь, например, фиксировать вращение только в одну сторону, для удобства работы.
- Если вдруг изобретут насадку в форме звёздочки, вам не придётся выкидывать «основание» такой отвёртки. Вам просто надо будет найти соответствующую насадку.

- В то время класс «насадка» совершенно ничего не знает про то, с каким основанием он будет работать: оно может быть длинным, коротким, угловым, это вообще может быть дрель-шуруповёрт.
- Задумайтесь. Каждая деталь фактически становится самостоятельным инструментом, который хоть и бесполезен без своей «второй половинки», но является универсальным для всех таких половинок. Это удобно.

ДЕЛИМ «ВИСЕЛИЦУ» НА КЛАССЫ

- Грубо говоря, все методы нашей Виселицы v. I по факту занимаются двумя вещами:
- Меняют состояние игры
 - get_letters
 - get_user_input
 - check_input

- Выводят что-то на экран (или чистят его)
 - get_word_for_print
 - print_status
 - cls

• Это-то наталкивает нас на мысль, что игра по факту состоит из двух глобальных частей: «внутренность игры» и «интерфейс вывода». Отделим вывод информации для игрока от внутренней игровой логики. В действительности, методов окажется немного больше, но не пугайтесь, всё достаточно просто.

КЛАСС GAME

- В первой части (игра) у нас будет всё, что связано с состоянием игры: загаданное слово, отгаданные буквы, буквы, которых в слове не оказалось и количество ошибок будут полями (переменными экземпляров) класса Game.
- Для этого класса создадим отдельный файл: game.rb (как обычно, в новой папке урока c:\rubytut\lesson | 2).
- A методы get_user_input и check_input мы соберём в один метод ask_next_letter.

КЛАСС GAME

• Итак, поехали, повторяем логику программы.

```
class Game
  # тут будет описание класса Game
end
```

METOД INITIALIZE

• Конструктор класса Game у нас будет вызывать метод get_letters (так как игра начинается с загадывания слова и без этого слова мы не можем продолжать игру). Так как метод get_letters принадлежит тому же классу Game, в конструкторе нет необходимости писать Game.get_letters, достаточно написать просто get_letters. Это верно для всех методов класса Game, вызываемых из других методов класса Game.

METOД INITIALIZE

```
def initialize(slovo)
   @letters = get_letters(slovo)
   @errors = 0
   @good_letters = []
   @bad_letters = []
   @status = 0
end
```

METOД INITIALIZE

- Заметьте, мы ждём в конструкторе параметр slovo (да, конструктор это обычный метод, и в него тоже можно передать параметр).
- При вызове:

```
Game.new("жираф")
```

• нам нужно будет в скобках указать загаданное слово, чтобы игра началась, а слово сохранилось в переменной letters нового экземпляра класса Game. Обратите также внимание на новое поле @status, оно нам понадобится в дальнейшем.

METOД GET_LETTERS

```
def get_letters(slovo)
  if (slovo == nil || slovo == "")
    abort "Для игры введите загаданное слово в качестве аргумента при запуске программы"
  end
  return slovo.split("")
end
```

• Как и в старой версии, этот метод принимает на вход строчку с загаданным словом, проверяет, есть ли в этой строчке что-нибудь и если там пусто, заканчивает программу, сообщив об этом пользователю. Если же там что-то есть, он разбивает слово на буквы уже знакомым нам способом и возвращает получившийся массив конструктору, чтобы тот мог его записать в поле класса letters.

METOД ASK_NEXT_LETTER

```
def ask_next_letter
  puts "Введите следующую букву"
  letter = ""
  while letter == "" do
    letter = STDIN.gets.encode("UTF-8").chomp
  end
  next_step(letter)
end
```

• Мы немного переименовали метод get_user_input, чтобы было понятнее, что именно он делает.

METOД ASK_NEXT_LETTER

```
def ask_next_letter

рыть "Ввелите следующую букву"
```

- Метод ask_next_letter не просто берёт то, что ввёл пользователь, а именно спрашивает следующую букву. Всегда старайтесь называть ваши классы и методы максимально точно.
- В нём мы спрашиваем у пользователя букву и добиваемся, чтобы он её-таки ввёл (в цикле проверяя, не ввёл ли он пустоту), а потом вызываем метод next_step, который эту букву обработает, как надо. Обратите внимание, опять внутренний метод класса мы вызываем без упоминания самого класса (не пишем Game. перед названием метода).

МЕТОД NEXT_STEP

• Метод next_step по сути, самый важный, он передвигает состояние игры на следующий шаг, проверяя букву в слове.

```
def next_step(bukva)
  if @status == -1 || @status == 1
    return
  end

if @good_letters.include?(bukva) || @bad_letters.include?(bukva)
    return
  end

if @letters.include? bukva
```

METOД NEXT_STEP

```
II @LECCETS.INCLUME: DUKVA
    @good_letters << bukva</pre>
    if @good_letters.uniq.sort == @letters.uniq.sort
      @status = 1
    end
  else
    @bad_letters << bukva</pre>
    @errors += 1
    if @errors >= 7
      astatus = -1
    end
  end
end
```

METOД NEXT_STEP

- Этот метод очень похож на наш старый метод check_input с той лишь разницей, что все данные он берёт не из параметров, а из полей класса.
- Это ещё одно удобство классов: они хранят нужные для их методов данные в полях, рядом с этими же методами, не загромождая ими остальные части вашей программы.
- Обратите внимание, также, что этот метод ничего не возвращает, он просто меняет состояние поля @status (вот оно нам и пригодилось), это ещё одно удобство: методам класса Game не надо ничего возвращать, они просто меняют поля класса Game.

МЕТОДЫ-ГЕТТЕРЫ

• Ещё одна особенность классов заключается в том, что они «прячут» переменные своих экземпляров от чужих глаз. Если у нас в программе будет экземпляр класса Game:

```
game = Game.new('слово')
```

• То мы не сможем достать переменные экземпляра game (@status, например) просто так (нельзя просто взять и написать game.@status — будет ошибка). Зачем так придумали мы сейчас рассказывать не будем, важно лишь то, что нам нужно научиться доставать значения переменных для экземпляров класса Game.

МЕТОДЫ-ГЕТТЕРЫ

• Ещё одна особенность классов заключается в том, что они «прячут» переменные своих экземпляров от чужих глаз. Если у нас в программе будет экземпляр класса Game:

```
def status
  return @status
end

def errors
  return @errors
end

def letters
  return @letters
end
```

```
def good_letters
  return @good_letters
end

def bad_letters
  return @bad_letters
end
```

 Для каждого поля класса мы написали метод, который, будучи вызванным у экземпляра этого класса, возвращает значение одноимённой переменной класса.

КЛАСС RESULTPRINTER

- Теперь переходим ко второй части нашей программы: выводу информации на экран. Этим будет заниматься класс ResultPrinter, который как и полагается новому классу, мы будем описывать в отдельном файле result_printer.rb.
- Обратите внимание, что для названий классов мы используем <u>CamelCase</u> написание словосочетаний без пробелов с увеличенной большой буквой каждого слова, а названия файлов этих классов мы пишем маленькими буквами, заменяя пробелы нижним подчёркиванием это соглашение, принятое в сообществе Ruby, настоятельно советуем вам поступать также.

class ResultPrinter
end

METOД PRINT_STATUS

• Первый и самый важный метод класса ResultPrinter будет заниматься выводом состояния игры на экран.

```
def print_status(game)
    cls
    puts "Слово: #{get_word_for_print(game.letters, game.good_letters)}"
    puts "Ошибки: #{game.bad_letters.join(", ").to_s}"

if game.status == -1
    puts "Вы проиграли :("
    puts "Загаданное слово было: " + game.letters.join("")
    elsif game.status == 1
    puts "Поздравляем, вы выиграли!"
    else
        puts "У вас осталось ошибок: " + (7 - game.errors).to_s
    end
end
```

METOД PRINT_STATUS

- Этот метод вызывается каждый раз, когда нам нужно обновить картинку для игрока показать ему новую виселицу.
- Во-первых, этот метод чистит экран с помощью метода cls, который тоже, конечно же, логично сделать частью класса ResultPrinter (если вы забыли, как он работает вспомните).

```
def cls
  system "clear" || system "cls"
end
```

METOД PRINT_STATUS

- Во-вторых, он не может просто взять данные из полей объекта класса Game. Ему нужно передать этот объект, а уже состояние игры наш ResultPrinter сам возьмет из объекта Game с помощью геттеров.
- Ну и в-третьих, здесь описан вспомогательный метод get_word_for_print, чтобы напечатать слово с закрытыми неразгаданными буквами (как в «Поле чудес»):

```
def get_word_for_print(letters, good_letters)
  result = ""

for item in letters do
  if good_letters.include?(item)
    result += item + " "
  else
    result += "__ "
  end
  end
  return result
end
```

- Самое время написать нашу программу с использованием новых классов Game и ResultPrinter: пора взять основание отвёртки, насадить на него наконечник и закрутить парочку шурупов, Дамы и Господа!
- Сперва нам надо наши классы подключить:

```
require_relative "game"
require_relative "result_printer"
```

• Теперь создадим по экземпляру каждого класса. ResultPrinter создаём просто вызвав у него new (у него даже конструктора нет, ничего страшного, так можно), а вот для игры нам нужно получить слово.

• Обратите внимание, классу game абсолютно плевать, откуда мы возьмём это слово, главное чтобы мы передали его конструктору. А берём мы слово как обычно из строки запуска и передаём его в конструктор класса Game.

```
slovo = ARGV[0]
  if (Gem.win_platform? && ARGV[0])
    slovo = slovo.dup
        .force_encoding("IBM866")
        .encode("IBM866", "cp1251")
        .encode("UTF-8")
    end

game = Game.new(slovo)
printer = ResultPrinter.new
```

- Время запустить основной игровой цикл. Условием выхода из цикла будет смена статуса игры (game.status, не забываем, что у нас есть такой метод-геттер), который изначально равен 0 (прописали в конструкторе).
- Теперь мы знаем, что как только @status в нашем объекте game (не путать с классом Game) станет отличным от 0, мы выйдем из цикла и закончим работу программы. В цикле мы выводим текущее состояние игры на экран и спрашиваем новую букву у игрока:

```
while game.status == 0 do
  printer.print_status(game)
  game.ask_next_letter
end
```

- Обратите внимание, насколько ёмким теперь выглядит тело цикла. Вся сложность ушла туда, где ей и суждено быть во внутреннюю логику методов классов Game и ResultPrinter.
- После цикла нам только ещё раз нужно написать результат и вуаля! Программа готова! Дальше всё произойдёт само, программа стала простой и ясной. Не бойтесь ошибок и опечаток.

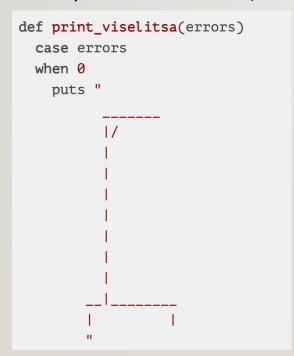
• Теперь программу можно запустить:

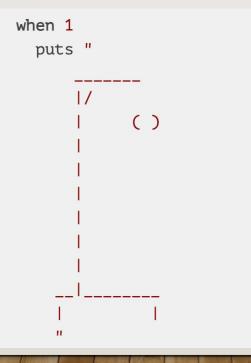
```
cd c:\rubytut\lesson12
ruby viselitsa "космонавт"
```

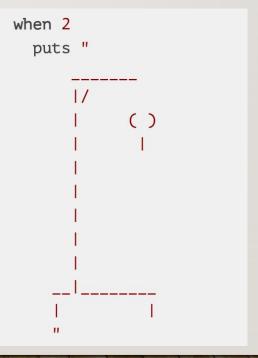
ВИЗУАЛИЗАЦИЯ РЕЗУЛЬТАТА

- Теперь продемонстрируем еще одну крутую вещь, которую нам помогут сделать классы.
- Мы поменяем метод print_status класса ResultPrinter, добавив туда псевдографику картинку, составленную из текстовых символов, тире, подчёркиваний, скобочек и прочего.
- Все остальные части нашей программы, а именно, код основной её части viselitsa.rb и код класса Game останется прежним.

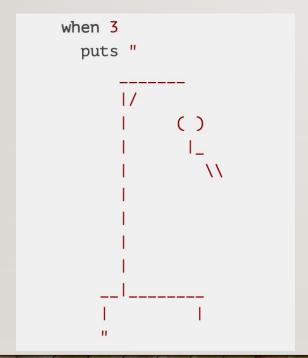
• А в класс ResultPrinter добавим новый достаточно громоздкий метод для отрисовки картинки виселицы в зависимости от количества ошибок:

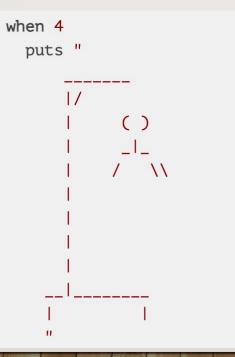


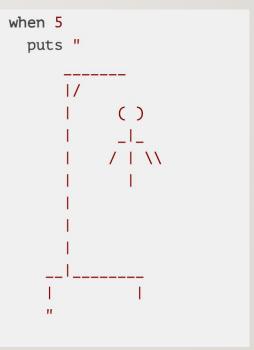




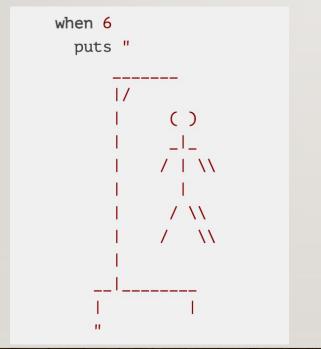
• А в класс ResultPrinter добавим новый достаточно громоздкий метод для отрисовки картинки виселицы в зависимости от количества ошибок:

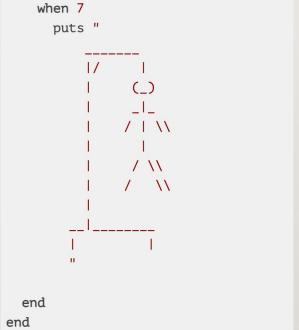






• А в класс ResultPrinter добавим новый достаточно громоздкий метод для отрисовки картинки виселицы в зависимости от количества ошибок:





- Meтoдy print_viselitsa нужно только передать в качестве параметра количество ошибок.
- Обратите внимание, что вместо одного обратного слеша \ мы пишем два \\, это так называемые спец-символы. Например, символ переноса строки тоже начинается со слеша: \n, так Ruby (и не только Ruby, это во многих языках верно) понимает, что эта n (что после слеша) не просто буква n, а именно перенос строки.
- Если же мы хотим напечатать просто обратный слеш \, как он есть, то Ruby может подумать, что мы хотим начать таким образом какой-то спец-символ, поэтому умные программисты добавили спец. символ, который просто выводит обратный слеш: \\. Просто запомните это. Пригодится.

ОПЕРАТОР CASE

В нашем методе print viselitsa мы использовали конструкцию casewhen, которая очень удобна, когда у нас есть переменная и много (больше 2-х) вариантов развития событий в зависимости от того, что в этой переменной находится. Если if-else это развилка, где всего два пути, то case — развилка где дорог может быть сколько угодно:

```
case fruit
when 'banana'

puts "Это банан"
when 'apple'

puts "Это яблоко"
when 'orange'

puts "Это апельсин"
else

puts "Это какой-то непонятный фрукт"
end
```

ОПЕРАТОР CASE

• Оператор case сравнивает значение выражения (в данном случае fruit) со всеми вариантами. Если в переменной fruit записана строка "banana", то на экран выведется строчка:

"Это банан"

• если в переменной fruit записана строка "apple", то на экране окажется:

"Это яблоко"

ОПЕРАТОР CASE

• Если же в переменной fruit — "вишня" или вообще цифра 5, то программа не выберет ни один из вариантов и пойдёт по варианту, который начинается со спец-слова default (по умолчанию, как аналог else):

"Это какой-то непонятный фрукт"

- Запустите вашу новую виселицу и поиграйтесь с ней, глядя на новую псевдографику.
- Если что-то не работает, то внимательно исправьте все ошибки.

ОПЕРАТОР CASE

- И помните, что с первого раза (и даже с десятого) может не получиться даже у гениальных программистов. Ваша настойчивость в борьбе с трудностями и ошибками это важнейшая часть вашего обучения. Чем труднее сейчас, тем проще будет потом.
- А в этом уроке мы попробовали классы в деле, разбив нашу программу Виселица на два класса Game и ResultPrinter, узнали, как работает оператор case, как пользоваться полями класса и немного узнали о спецсимволах и псевдографике.

ГЕРОИ И ЗЛОДЕИ

- Напишите с помощью case программу, которая отвечает на вопрос, кто был главным врагом указанного героя.
- Например:

Врага какого персонажа вы хотите узнать?
> Бэтмен
Джокер!
Врага какого персонажа вы хотите узнать?
> Шерлок Холмс
Профессор Мориарти

ГЕРОИ И ЗЛОДЕИ

- Конечно список возможных пар имен героев должен быть прописан в самой программе. Чтобы пользователь примерно знал, что спрашивать.
- Вот ещё пары для вдоховения: Буратино Карабас-Барабас, Фродо Бэггинс Саурон, Моцарт Сальери.
- Для особенно любознательных: сделайте так, чтобы имя персонажа можно было ввести маленькими буквами и по-английски, для этого по-лучше изучите особенности конструкции case в Ruby.
- Если враг персонажа не найден, программа должна отвечать: Не удалось найти врага.
- Помните также, что строки, написанные разными буквами (ЗАГЛАВНЫМИ и строчными, или сМешаНными), в Ruby считаются совершенно разными строчками.

ГЕРОИ И ЗЛОДЕИ. ПОДСКАЗКА

- Заведите переменную hero и сохраните в неё то, что введёт пользователь с помощью команды gets.
- А потом с помощью case выберите один из подходящих вариантов. Не забудьте написать else, для случая, когда hero не совпадёт ни с одним из написаний.
- Чтобы разрешить писать как английскими, так и русскими буквами, да ещё и независимо от регистра (большие/маленькие), нужно в проверке условия после when написать несколько строк, разделяя их запятой:

```
when "batman", "Ваtman", "Бэтмен", "бэтмен"
```

ФИЛЬМЫ С РЕЖИССЕРАМИ

- Напишите программу, помогающую выбрать какой фильм сегодня просмотреть.
- Создайте класс «Фильм». У него должно быть два свойства название фильма и фамилия режиссера. Оба этих значения должны передаваться как параметры в конструкторе.
- Напишите программу, которая спрашивает у пользователя фамилию любимого режиссера, а затем спрашивает в цикле три раза три любимых фильма этого режиссера.
- В этом же цикле программа создает массив из объектов класса «Фильм». После чего программа должа выбрать случайный элемент этого массива и выводить его на экран. То есть показать имя режиссера и название фильма.

ФИЛЬМЫ С РЕЖИССЕРАМИ

• Например:

```
Фильмы какого режиссера Вы хотите посмотреть?
```

> Роберт Земекис

Какой-нибудь его хороший фильм?

> Форрест Гамп

Какой-нибудь его хороший фильм?

> Назад в будущее

Какой-нибудь его хороший фильм?

> Экипаж

И сегодня вечером рекомендую посмотреть: Форрест Гамп

Режиссера: Роберт Земекис

ФИЛЬМЫ С РЕЖИССЕРАМИ. ПОДСКАЗКА

- Не забудьте конвертировать введенные пользователем строки в правильную кодировку (использовать gets.encode("UTF-8")).
- В массив можно добавлять любые объекты точно так же как мы делали со строками и числами ранее. Просто создавайте в цикле три разных объекта класса «фильм» и добавляйте их в массив.
- Случайный элемент из любого массива выбирается методом sample, как мы делали в уроке про волшебный шар.

люди и фильмы

- Объединяем людей и фильмы. Напишите программу, которая будет использовать одновременно два класса из предыдущих заданий.
- Добавьте в класс «Человек» поле, хранящее любимый фильм данного человека. В это поле будет записываться объект класса «Фильм».
- Также добавьте в класс «Человек» два метода: один будет записывать значение в это поле, другой будет возвращать текущее значение этого поля.
- Создайте трех людей, каждому из них назначьте (с использованием нового метода) по одному фильму и выведите всех трех людей и их фильмы на экран.

люди и фильмы

• К примеру:

Сергей

с любимым фильмом: Челюсти

Марина

с любимым фильмом: Список Шиндлера

Мадонна

с любимым фильмом: Парк Юрского периода

люди и фильмы

• Подключайте в программу два класса из прошлых заданий. А в класс Person добавьте новую переменную экземпляра «любимый фильм» (в конструкторе) и два метода: один с параметром, он задает новое значение этой переменной; другой без параметра, он возвращает значение переменной.

СПРАВОЧНАЯ ИНФОРМАЦИЯ

- Грэди Буча начали читать? Пора начать ;)
- Про объектное программирование на Wiki

СПАСИБО ЗА ВНИМАНИЕ!

Объектная «Виселица» v.2.0