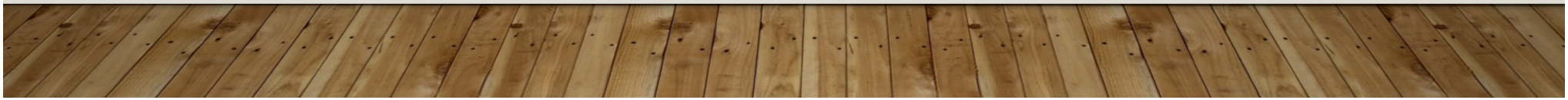
ЛЕКЦИЯ 26

КЛАССЫ, АБСТРАКТНЫЕ И СТАТИЧЕСКИЕ МЕТОДЫ



ПЛАН ЗАНЯТИЯ

1. Статические методы
2. Абстрактные методы
3. Блокнот v. 1.0
4. Первая полезная программа на github!

- На прошлом уроке мы научились наследовать классы друг от друга. Сегодня вы ещё немного нового узнаете об объектно-ориентированном подходе. В уроке расскажем о статических и абстрактных методах и научим вас расширять поведение базового (родительского) класса.

СТАТИЧЕСКИЕ МЕТОДЫ КЛАССА

- Давайте вспомним нашу программу с мостами.
- Обратите внимание на два метода bridge.open мы вызываем у экземпляра класса Bridge, а Bridge.new мы вызываем у самого класса Bridge.
- Методы, которые вызываются у класса, без создания экземпляра, например вышеупомянутый new — это статические методы.

```
puts "Река, нужно перекинуть мост"
sleep 1

puts "Создаём мост"
bridge = Bridge.new
sleep 1

puts "Открываем мост"
bridge.open
sleep 1

puts "Проехали мост, едем дальше..."
sleep 1
```

ЗАЧЕМ НУЖНЫ СТАТИЧЕСКИЕ МЕТОДЫ?

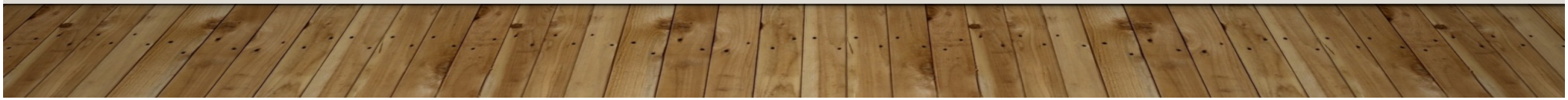
- У нас есть класс, по которому, как по чертежу, мы можем создавать экземпляры, то есть объекты. А что если какая-то функция общая, то есть не зависит от конкретного объекта.

ЗАЧЕМ НУЖНЫ СТАТИЧЕСКИЕ МЕТОДЫ?

- Например у нас есть класс Человек, можем создать экземпляр человек и у конкретного человека можно спросить, какой у него цвет глаз: человек.цвет_глаз. А что, если нам нужно узнать, сколько всего людей на планете Земля или, например, какой человек самый высокий? Никакой конкретный человек этого не знает (без ограничения общности считаем, что у нас нет переменной, которая указывает на человека с именем Анатолий Вассерман). Тогда нам нужно спросить об этом у самого класса, который заведует всеми людьми: Человек.сколько или Человек.самый_высокий.

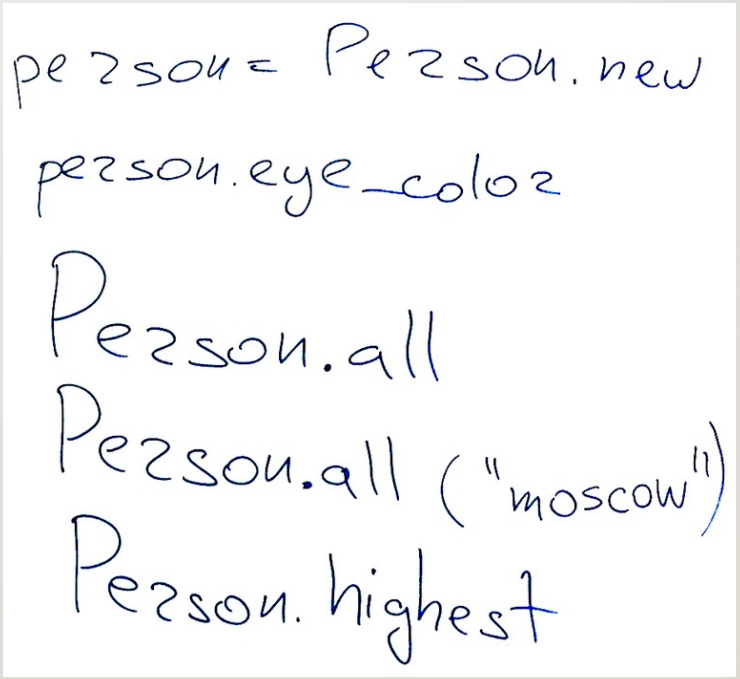
ЗАЧЕМ НУЖНЫ СТАТИЧЕСКИЕ МЕТОДЫ?

- А, например, чтобы узнать, сколько людей в конкретном городе, можно было бы методу сколько передать параметр: Человек.сколько("Москва").
- Статический метод, также, как и обычный, может что-то возвращать. Например Bridge.new — статический метод, встроенный в руби, возвращающий экземпляр класса Bridge.



ЗАЧЕМ НУЖНЫ СТАТИЧЕСКИЕ МЕТОДЫ?

- Также вам, наверное, знакомы статические методы Time.now и Date.parse.



КАК СОЗДАВАТЬ СТАТИЧЕСКИЕ МЕТОДЫ

- Определить статический метод у класса очень просто. Достаточно просто написать перед названием определяемого метода кодовое слово self:

```
class MyClass
  def self.static_method
    puts "Я статический метод класса MyClass"
  end
end

MyClass.static_method
```

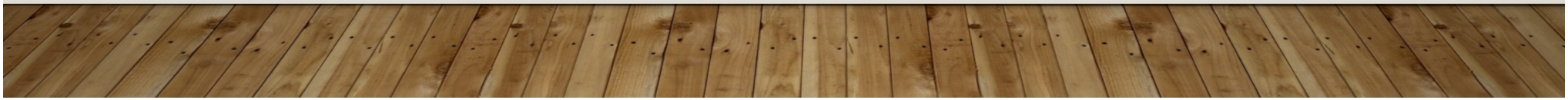
- Выведет на экран:

```
Я статический метод класса MyClass
```

- Обратите внимание ещё раз — мы не создавали экземпляр класса MyClass, мы вызвали метод самого класса.

ПЕРЕГРУЗКА МЕТОДОВ

- Из названия более-менее ясно, что это такое. Методы детей могут отличаться от методов родителей с таким же названием. Когда класс наследует метод, он может изменить реализацию на свою, если это нужно.
- Почти все рестораны подают бизнес-ланч, но в каждом ресторане бизнес-ланч свой. То, что в него будет входить — определяется для каждой кухни отдельно. Для итальянской одно, для китайской — совсем другое.

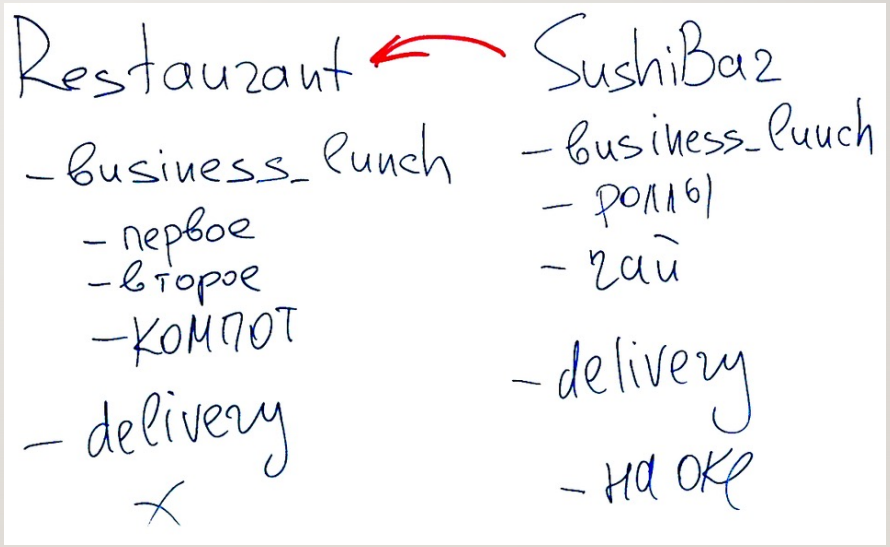


ПЕРЕГРУЗКА МЕТОДОВ

- Пусть у нас есть класс Ресторан и у него определен метод бизнес_ланч: в него входит *первое*, *второе* и *десерт*.
- Допустим, у нас также есть ребенок СушиБар < Ресторан. У него тогда автоматически тоже будет метод бизнес_ланч. Однако, если понадобится, в бизнес_ланч от СушиБара могут входить совсем другие блюда: например, *роллы* и *мисо суп*.

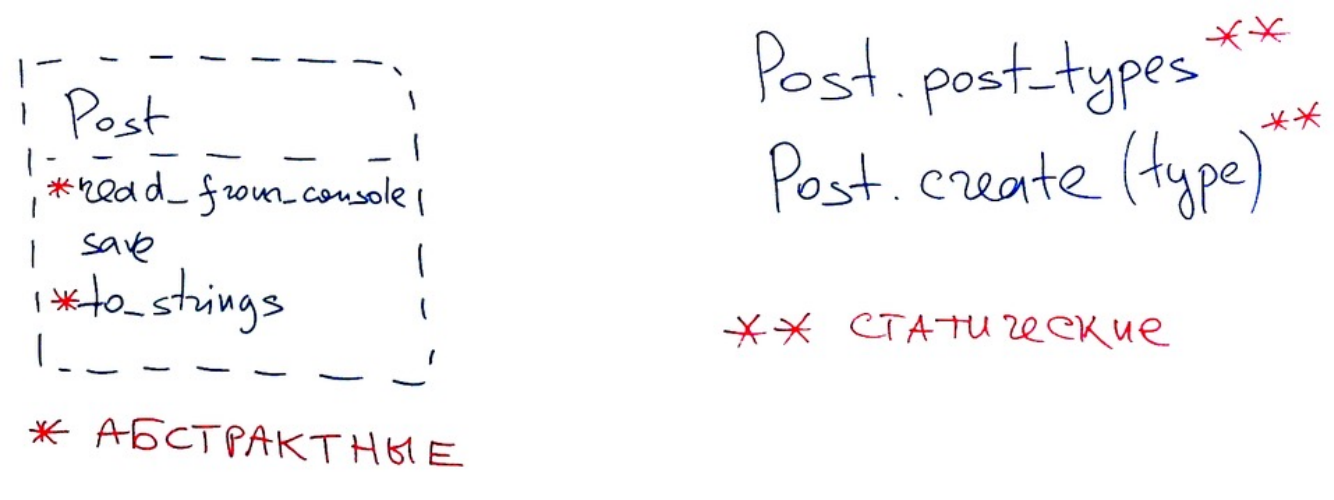
АБСТРАКТНЫЕ МЕТОДЫ

- Если реализация метода у родительского класса нас не интересует вовсе, то мы можем не писать её, а просто обозначить, что такой метод вообще есть. Тогда у каждого ребенка уже должна быть своя конкретная реализация.
- В некоторых языках программирования, например, в Java, такие методы называют **абстрактными**.



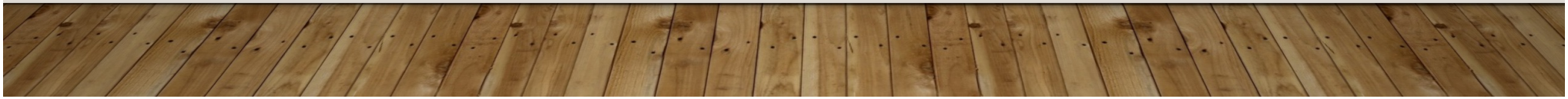
БЛОКНОТ ВЕРСИЯ V.1.0

- Давайте спроектируем наш блокнот из предыдущего урока и напишем недостающие методы.



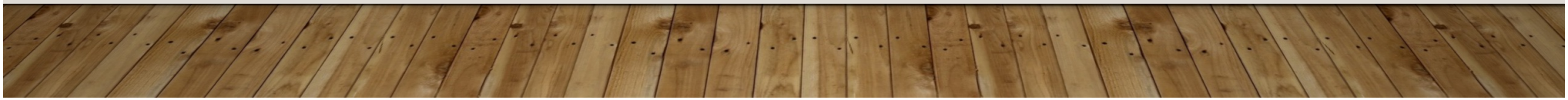
БЛОКНОТ ВЕРСИЯ V.1.0

- У класса Post будут:
  - статический метод post_types, который будет возвращать ассоциативный массив всех возможных детей этого класса (чтобы можно было спросить у пользователя, что он хочет создать);
  - метод create, который по переданному значению будет создавать нужного ребенка;
  - два абстрактных метода read_from_console и to_strings, которые будут реализованы у каждого ребенка;
  - и метод save, который будет только у родителя, и который будет использовать метод to_strings;
  - а также служебный метод file_path, который просто будет использоваться для определения, куда сохранять заметку.



POST.RB

```ruby
# Базовый класс "Запись"
# Задает основные методы и свойства, присущие всем разновидностям Записи
class Post
  # Конструктор
  def initialize
    @created_at = Time.now # дата создания записи
    @text = [] # массив строк записи — пока пустой
  end
```

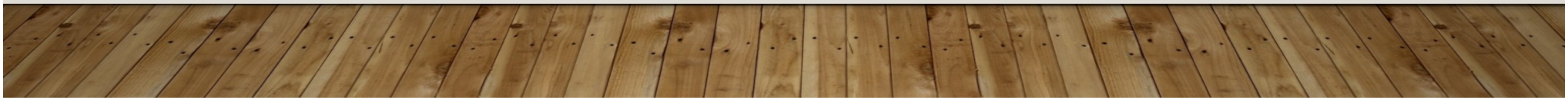


POST.RB

```
# Набор известных детей класса Запись в виде массива классов
def self.post_types
  [Memo, Task, Link]
end
# XXX/ Строго говоря этот метод self.types нехорош – родительский
  # класс в идеале в коде
# не должен никак зависеть от своих конкретных детей.
  # Мы его использовали для простоты
# (он адекватен поставленной задаче).
#
# В сложных приложениях это делается немного иначе: например
  # отдельный класс владеет всей информацией,
# и умеет создавать нужные объекты (т. н. шаблон проектирования
  # "Фабрика").
# Или каждый дочерний класс динамически регистрируется в подобном
  # массиве сам во время загрузки программы.
# См. подробнее книги о шаблонах проектирования в доп. материалах.
```

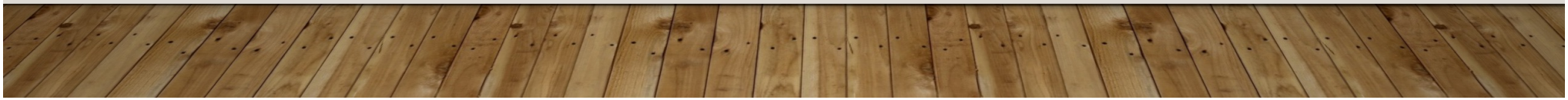
POST.RB

```
# Динамическое создание объекта нужного класса из набора возможных детей
def self.create(type_index)
  return post_types[type_index].new
end
```



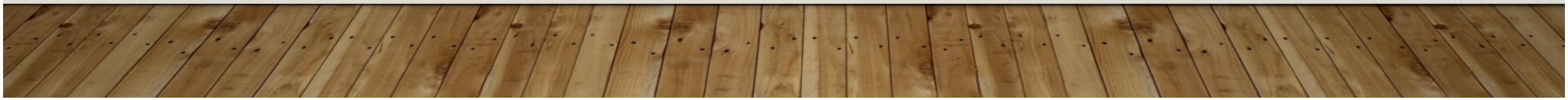
POST.RB

```
# Вызываться в программе когда нужно считать ввод пользователя и
  # записать его в нужные поля объекта
def read_from_console
  # todo: должен реализовываться детьми, которые знают как именно
  # считывать свои данные из консоли
end
```



POST.RB

```
# Возвращает состояние объекта в виде массива строк,
  # готовых к записи в файл
def to_strings
  # todo: должен реализовываться детьми, которые знают как именно
  # хранить себя в файле
end
```



POST.RB

```ruby
# Записывает текущее состояние объекта в файл
def save
  file = File.new(file_path, "w:UTF-8") # открываем файл на запись

  for item in to_strings do # идем по массиву строк, полученных
  # из метода to_strings
    file.puts(item)
  end

  file.close # закрываем
end
```

POST.RB

```
  # Метод, возвращающий путь к файлу, куда записывать этот объект
  def file_path
    # Сохраним в переменной current_path место, откуда запустили программу
    current_path = File.dirname(__FILE__)

    # Получим имя файла из даты создания поста метод strftime формирует
    # строку типа "2014-12-27_12-08-31.txt"
    # набор возможных ключей см. в документации Руби
    file_name = @created_at.strftime("#{self.class.name}_%Y-%m-%d_%H-%M-%S.txt")
    # Обратите внимание, мы добавили в название файла даже секунды (%S) – это
    # обеспечит уникальность имени файла

    return current_path + "/" + file_name
  end
end
```

POST.RB

```
# PS: Весь набор методов, объявленных в родительском классе называется
    # интерфейсом класса
# Дети могут по-разному реализовывать методы, но они должны подчиняться общей идее
# и набору функций, которые заявлены в базовом (родительском классе)

# PPS: в других языках (например Java) методы, объявленные в классе, но пустые
# называются абстрактными (здесь это методы to_strings и read_from_console).
#
# Смысл абстрактных методов в том, что можно писать базовый класс и пользоваться
# этими методами уже как будто они реализованы, не задумываясь о деталях.
# С деталями реализации методов уже заморачиваются дочерние классы.
#
```

- Пора реализовать абстрактные методы у детей: статьи, ссылки и задачи.

LINK.RB

```
class Link < Post
  def initialize
    # Вызовем одноимённый метод (initialize) родителя (Post) методом super
    super

    # А потом добавим то, что будет отличаться у ребёнка — поле @url
    @url = ''
  end
```

LINK.RB

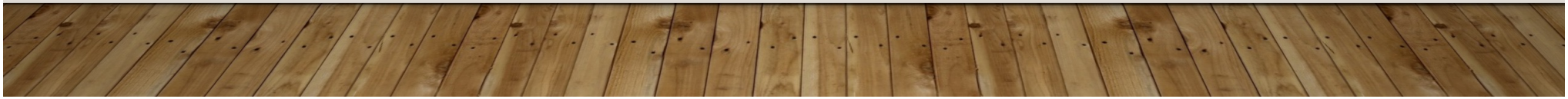
```
def read_from_console
  # Мы полностью переопределяем метод read_from_console родителя Post

  # Попросим у пользователя адрес ссылки
  puts "Введите адрес ссылки"
  @url = STDIN.gets.chomp

  # И описание ссылки (одной строчки будет достаточно)
  puts "Напишите пару слов о том, куда ведёт ссылка"
  @text = STDIN.gets.chomp
end
```

LINK.RB

```
  def save
    # Метод save во многом повторяет метод родителя, но отличия существенны

    file = File.new(file_path, "w:UTF-8")
    time_string = @created_at.strftime("%Y.%m.%d, %H:%M")
    file.puts(time_string + "\n\r")

    # Помимо текста мы ещё сохраняем в файл адрес ссылки
    file.puts(@url)
    file.puts(@text)

    file.close

    # Напишем пользователю, что запись добавлена
    puts "Ваша ссылка сохранена"
  end
end
```

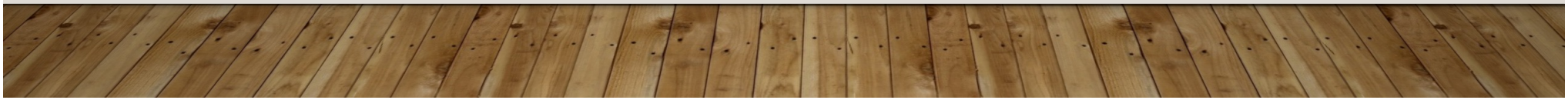
MEMO.RB

```
class Memo < Post
  def read_from_console
    # Метод, который спрашивает у пользователя, что он хочет написать в дневнике
    puts "Я сохраню всё, что ты напишешь до строчки \"end\" в файл."

    # Объявим переменную, которая будет содержать текущую введенную строку
    line = nil

    # Запустим цикл, пока не дошли до строчки "end",
    while line != "end" do
      # Читаем очередную строку и записываем в массив @text
      line = STDIN.gets.chomp
      @text << line
    end

    # Теперь удалим последний элемент из массива @text - там служебное слово "end"
    @text.pop
  end
```



MEMO.RB

```
def save
  # Откроем файл для записи в режиме записи (write)
  # Файл не существует и будет создан
  file = File.new(file_path, "w:UTF-8")

  # Обратите внимание, что мы вызвали метод file_name, который определили выше
  # save и file_name — методы одного класса и поэтому могут использовать
  # друг друга

  # Сперва запишем в блокнот дату и время записи и сделаем отступ
  # \r - специальный дополнительный символ конца строки для Windows
  time_string = @created_at.strftime("%Y.%m.%d, %H:%M")
  file.puts(time_string + "\n\r")
```



MEMO.RB

```
    # Затем в цикле запишем в файл строчку за строчкой массив @text
    for item in @text do
      # Метод puts добавляет перевод строки в конце, что нам и надо
      file.puts(item)
    end

    # Обязательно закрыть файл, чтобы сохранить все изменения
    file.close

    # Напишем пользователю, что запись добавлена
    puts "Ваша запись сохранена"
  end
end
```

TASK.RB

```
# Подключим встроенный в руби класс Date для работы с датами
require 'date'

class Task < Post
  def initialize
    # Вызовем одноимённый метод (initialize) родителя (Post) методом super
    super

    # А потом добавим то, что будет отличаться у ребёнка — поле @due_date
    @due_date = ''
  end
```

TASK.RB

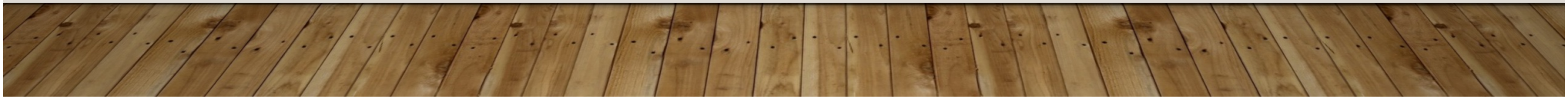
```
def read_from_console
  # Мы полностью переопределяем метод read_from_console родителя Post

  # Спросим у пользователя, что за задачу ему нужно сделать
  # Одной строчки будет достаточно
  puts "Что вам необходимо сделать?"
  @text = STDIN.gets.chomp

  # А теперь спросим у пользователя, до какого числа ему нужно это сделать
  # И подскажем формат, в котором нужно вводить дату
  puts "До какого числа вам нужно это сделать?"
  puts "Укажите дату в формате ДД.ММ.ГГГГ, например 12.05.2003"
  input = STDIN.gets.chomp

  # Для того, чтобы запись дату в удобном формате, воспользуемся
  # методом parse класса Time
  @due_date = Date.parse(input)
end
```

TASK.RB

```
  def save
    file = File.new(file_path, "w:UTF-8")
    time_string = @created_at.strftime("%Y.%m.%d, %H:%M")
    file.puts(time_string + "\n\r")

    # Так как поле @due_date указывает на объект класса Date, мы можем
    # вызвать у него метод strftime
    # Подробнее о классе Date читайте по ссылкам в материалах
    file.puts("Сделать до #{@due_date.strftime("%Y.%m.%d")}")
    file.puts(@text)

    file.close

    # Напишем пользователю, что задача добавлена
    puts "Ваша задача сохранена"
  end
end
```

NOTEPAD.RB

- Теперь есть чертежи, по которым можно строить любую запись. В основной программе — спросим пользователя, что он хочет создать, получив список возможных типов статическим методом Post.post_types.
- Создаём объект нужного класса, основываясь на ответе с помощью метода create, а потом просто вызываем у созданного объекта его методы read_from_console и save (обратите внимание, нам совершенно не важно, что за класс у нас получился, т.к. мы используем абстрактные методы).



NOTEPAD.RB

```
# Подключаем класс Post и его детей
require_relative 'post.rb'
require_relative 'memo.rb'
require_relative 'link.rb'
require_relative 'task.rb'

# Как обычно, при использовании классов программа выглядит очень лаконично
puts "Привет, я твой блокнот!"

# Теперь надо спросить у пользователя, что он хочет создать
puts "Что хотите записать в блокнот?"

# массив возможных видов Записи (поста)
choices = Post.post_types

choice = -1
```

NOTEPAD.RB

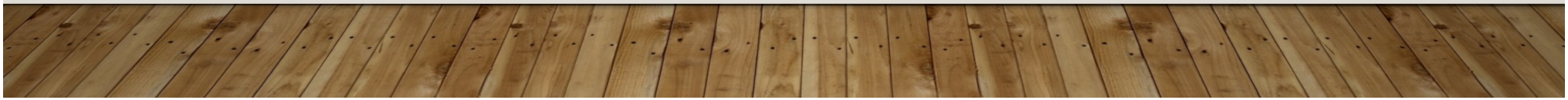
```
until choice >= 0 && choice < choices.size # пока юзер не выбрал правильно
  # выводим заново массив возможных типов поста
  choices.each_with_index do |type, index|
    puts "\t#{index}. #{type}"
  end
  choice = gets.chomp.to_i
end
```

NOTEPAD.RB

```
# сейчас в переменной entry лежит один из детей класса Post, какой именно,
# определилось выбором пользователя, переменной choice.
# Но мы не знаем какой, и обращаемся с entry как с объектом класса Post,
    # этого, оказывается, достаточно.

# Просим пользователя ввести пост (каким бы он ни был)
entry.read_from_console

# Сохраняем пост в файл
entry.save

puts "Ваша запись сохранена!"
```

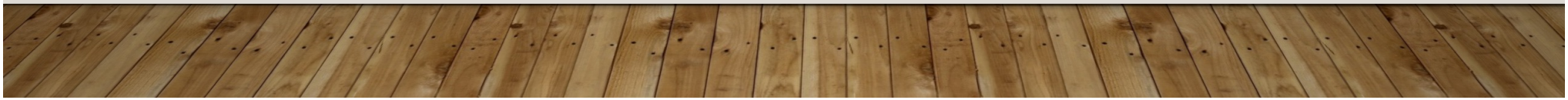
NOTEPAD.RB

- Итак, мы научились управлять классами как взрослые дядьки (и тётьки :-D). В следующих уроках мы начинаем новую тему — хранение данных. Конечно же эта тема тоже важна для уважающего себя программиста, так как все *не очень важные* темы мы просто заранее не стали разбирать.



МАГАЗИН С ВИТРИНОЙ И КОНСТРУКТОРАМИ

- Продолжаем развивать наш «Магазин», который мы создали в уроке про наследование классов.
- Сделайте так, чтобы из основной программы можно было создать объект класса Book или Film, передав ему кроме цены также другие параметры. Для книги — название, жанр и автора; для фильма — название, год и режиссера.
- Напишите также метод to_s для экземпляра класса Film и Book, который возвращает информацию об этом экземпляре одной строкой.



МАГАЗИН С ВИТРИНОЙ И КОНСТРУКТОРАМИ

- В основной программе создайте пару книжек и фильмов и выведите их в цикле на экран.
- **Например:**

```
Вот какие товары у нас есть:

Фильм «Леон», 1994, реж. Люк Бессон, 990 руб. (осталось 5)
Фильм «Дурак», 2014, реж. Юрий Быков, 390 руб. (осталось 1)
Книга «Идиот», роман, автор — Федор Достоевский, 1500 руб. (осталось 10)
```

МАГАЗИН С ЧТЕНИЕМ ФАЙЛОВ

- Продолжаем развивать наш «Магазин»: реализуйте функционал считывания продуктов из папки data. Пусть в папке, например, data/films лежат текстовые файлы в формате, который вам уже знаком (добавляется цена и остаток на складе):

```
Название фильма
Фамилия и имя режиссера
Год выхода
Цена
Остаток
```

МАГАЗИН С ЧТЕНИЕМ ФАЙЛОВ

- А в паке data/books — файлы в таком формате:

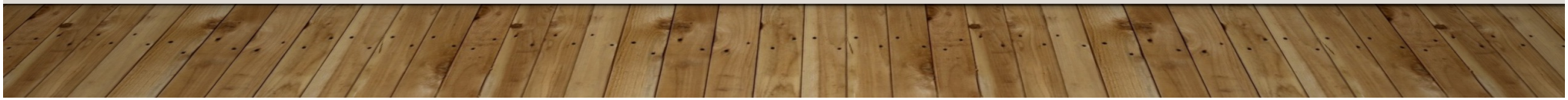
```
Название книги
Жанр
Фамилия и имя автора
Цена
Остаток
```

МАГАЗИН С ЧТЕНИЕМ ФАЙЛОВ

- Напишите для каждого класса-ребенка метод класса (статический метод) from_file, который создает новый экземпляр класса, заполняя его данными из файла, чтобы можно было написать вот так:

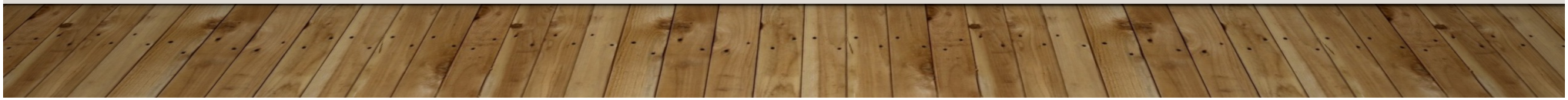
```
film = Film.from_file('./data/films/01.txt')
book = Book.from_file('./data/books/01.txt')
```

- Сделайте также, чтобы метод родителя возвращал ошибку NotImplementedError, на случай, если какой-то ребенок попытается создать себя используя статический метод родителя.



МАГАЗИН С ЧТЕНИЕМ ФАЙЛОВ. ПОДСКАЗКА

- Вспоминайте, как мы делали программу, которая читала фильмы из каталога, чтобы выбрать фильм на вечер.
- Напишите метод from_file у класса Film так, чтобы читал данные из файла и передавал их в конструктор подобно тому, как мы создавали фильмы из основной программы.
- Не забудьте у класса Product сделать так, чтобы метод from_file возвращал ошибку NotImplementedError.



МАГАЗИН С ProductCollection

- Продолжаем развивать наш «Магазин»: реализуйте класс ProductCollection, который может хранить в себе любые товары (фильмы или книги) и у которого есть:
- Метод класса (статический метод) from_dir, который считывает продукты из папки data, сам понимая, какие товары в какой папке лежат.
- Метод экземпляра to_a, который возвращает массив товаров.
- Метод экземпляра sort, который сортирует товары по цене, остатку на складе или по названию (как по возрастанию, так и по убыванию):
- Создайте в основной программе коллекцию товаров, прочитав её из директории и выведите все товары на экран.

МАГАЗИН С ProductCollection. ПОДСКАЗКА

- Для сортировки массива используйте метода sort_by!, который принимает на вход блок, по результату возврата которого будет идти сортировка:
- https://ruby-doc.org/core-2.4.0/Enumerable.html#method-i-sort_by

```
[1, 2, 4, 5, 10].sort_by! { |i| i * -1 }
# [10, 5, 4, 2, 1]
```

СПРАВОЧНАЯ ИНФОРМАЦИЯ

- Класс Date, работа с датами
- Хитрости устройства переменных класса в Ruby
- Что такое «Шаблоны проектирования»
- Шаблоны проектирования в Ruby — 1
- Шаблоны проектирования в Ruby — 2
- Что такое «переопределение методов»
- Что такое «абстрактный метод»

СПАСИБО ЗА ВНИМАНИЕ!

Лекция 26. Классы, абстрактные и статические методы