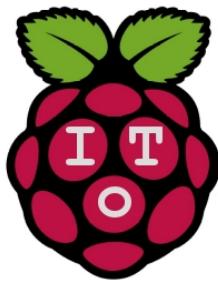


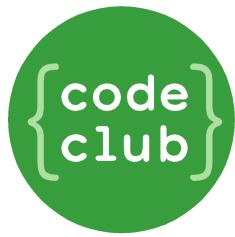
Internet of Things



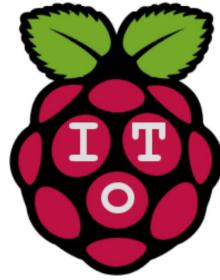
Объектно-ориентированное
программирование
на **Ruby**

Шадринск
2018-2019

M. V. Шохирев



Divide et impera !



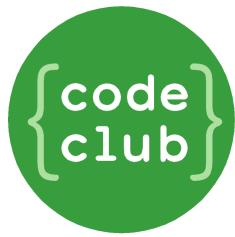
«Divide et impera!» (лат.) = «Разделяй и властвуй!»

Чем больше программа, тем труднее её изменять, дорабатывать, развивать и даже просто понимать, что она делает.

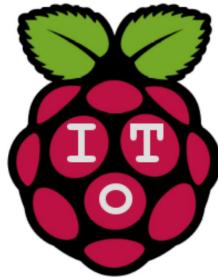
Поэтому уже давно программисты применяют *модульность* — разделение программ на небольшие части (программные модули = подпрограммы), каждая из которых выполняет одну хорошо понимаемую функцию. Проверенная подпрограмма может вызываться многократно для безошибочного выполнения своей работы.

Из логически взаимосвязанных модулей формируются *библиотеки программ* по разным видам решаемых задач.

В объектно-ориентированных языках программирования (**C++**, **Java**, **Kotlin**, **Ruby**, **Python** и т. п.) разрабатываются *библиотеки классов*.



Подпрограммы

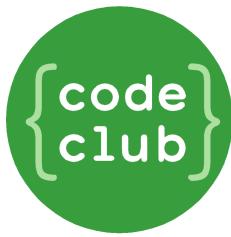


Подпрограммы — наборы команд, имеющие название (имя подпрограммы), которые можно многократно выполнять, вызывая их по имени.

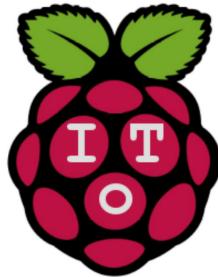
```
procedure() # вызов подпрограммы по имени
```

Подпрограммы обычно описываются в начале программы (или размещаются в отдельном исходном файле, который подключается к основной программе).

```
def procedure() # описание подпрограммы без параметров  
# ...  
end # полезные действия в подпрограмме  
# конец описания подпрограммы  
  
# значениями параметров
```



Подпрограммы: параметры



В описании подпрограммы можно предусмотреть *параметры* — переменные, значения которых можно определять при вызове подпрограммы для обработки в подпрограмме.

```
def procedure(prm1, prm2)
    print prm1.to_s + "+" + prm2
end
```

описание подпрограммы с параметрами
полезные действия в подпрограмме
конец описания подпрограммы

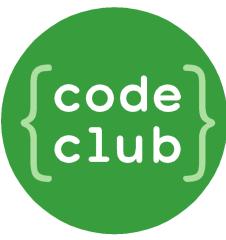
При вызове подпрограммы ей указываются *фактические значения* (аргументы) — литералы или переменные, доступные в подпрограмме через переменные параметров.

```
procedure(1.0, argument2)
```

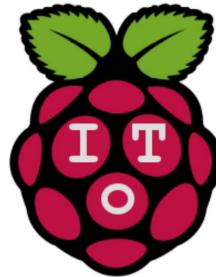
вызов подпрограммы с фактическими
значениями параметров

```
print "Начало работы \n"
```

скобки для списка аргументов
в языке Ruby можно не писать



Подпрограммы: функции и процедуры



Подпрограммы делятся на 2 вида: **функции**, которые возвращают вычисленный результат:

```
def function(prm1, prm2, prm3)          # описание функции с параметрами
    return 0                                # возврат результата функции
end                                         # конец описания функции

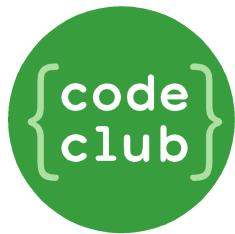
result = function(arg1, 2.0, true)         # вызов функции с фактическими
                                           # значениями параметров
print result                               # и присвоением результата
```

и **процедуры**, которые выполняют действия, но не возвращают результата.

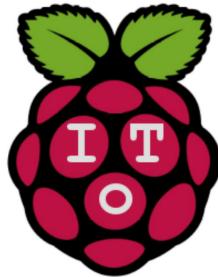
```
def procedure(prm1, prm2)          # описание процедуры с параметрами
    # ...
end                                         # полезные действия в подпрограмме
                                           # конец описания подпрограммы

procedure("1-й аргумент", argument2)    # вызов процедуры с фактическими
                                           # значениями параметров
```

В **Ruby** все методы возвращают значение явно по **return** или неявно: последнее выражение.



ООП

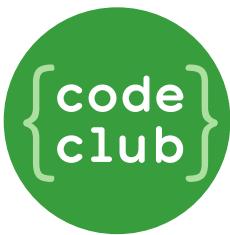


Объектно-ориентированным программированием (ООП) называется подход к разработке программ, при котором программно моделируются объекты реального мира и их взаимодействие.

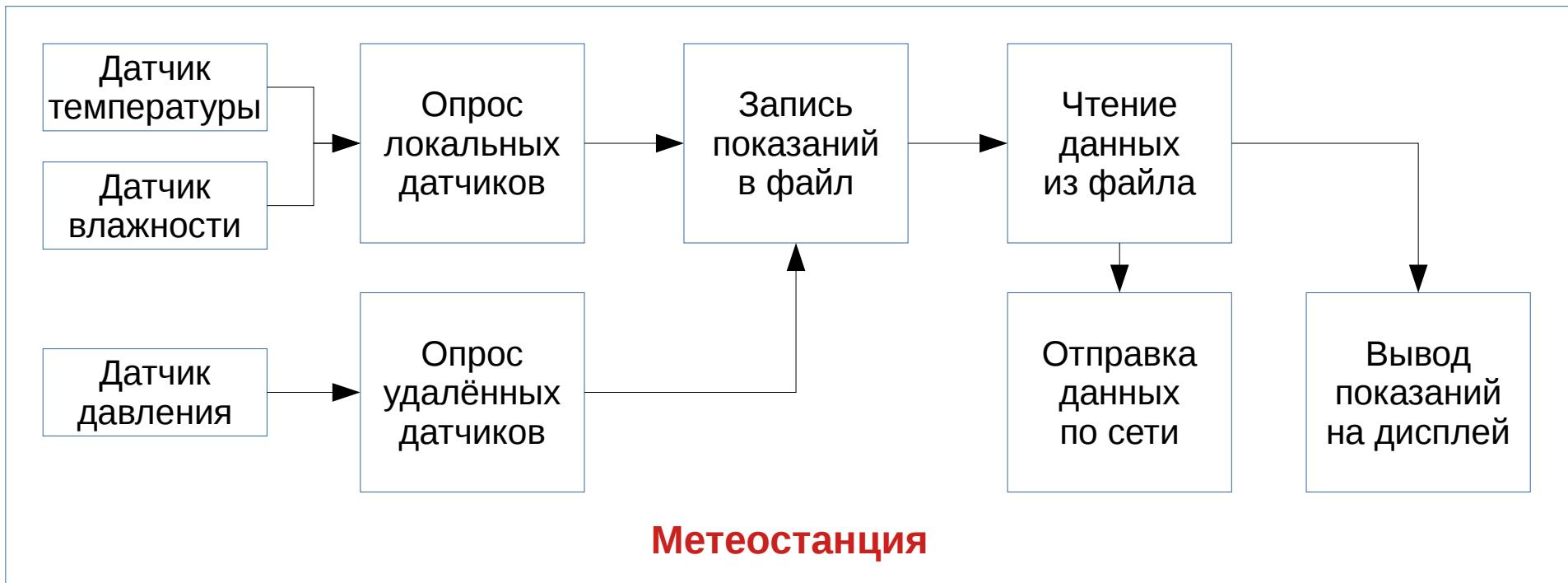
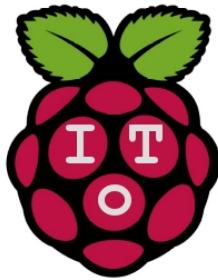
При разработке программы создаётся *программная модель*:

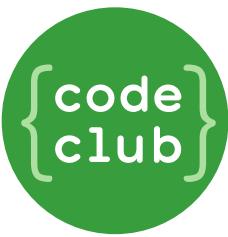
- предметы и понятия реальности определяются как **классы**, которые
- описывают **объекты** реального мира (светодиоды, кнопки, датчики, реле, моторы),
- их **свойства** (название, цвет, контакты, показания, состояние) и
- возможные **действия** с ними (включить, выключить, считать показания, изменить состояние, увеличить скорость).

Во время выполнения программы создаётся необходимое количество объектов разных классов, которые взаимодействуют между собой.



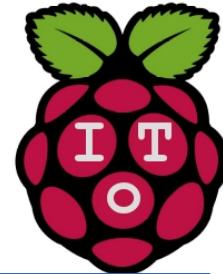
ООП: реальность





ООП:

программная модель



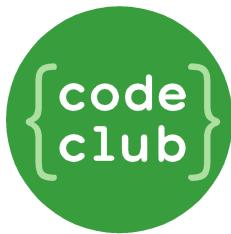
```
class TemperatureSensor  
end
```

```
class HumiditySensor  
end
```

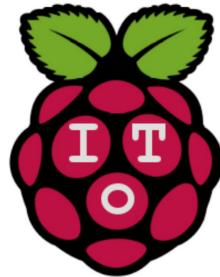
```
class PressureSensor  
end
```

```
class MeteoStation  
  def initialize  
    @t = TemperatureSensor.new  
    @h = HumiditySensor.new  
    @p = PressureSensor.new  
  end  
  
  def poll_local(sensor)  
  end  
  def poll_remote(sensor)  
  end  
  
  def write_to_file(data, file)  
  end  
  def read_from_file(file)  
  end  
  
  def show_on_display(data)  
  end  
end
```

```
m = MeteoStation.new  
  
t = m.poll_local(m.t)  
h = m.poll_local(m.h)  
p = m.poll_remote(m.p)  
  
f = File.new('F', 'w')  
  
m.write_to_file(t, f)  
m.write_to_file(h, f)  
m.write_to_file(p, f)  
  
d = m.read_from_file(f)  
  
m.show_on_display(d)
```



Подпрограммы: методы



Подпрограммы, описанные в классах для выполнения действий с объектами этих классов, называются *методами*.

```
class SomeClass
  def method_1()
    # ...
  end

  def method_2(p1)
    # ...
    return expression
  end
end

some_object = SomeClass.new
some_object.method_1()
result = some_object.method_2(25)
```

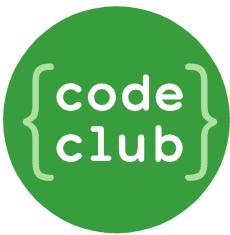
в описании класса
описание метода-процедуры без параметров
полезные действия
конец описания метода

описание метода-функции с параметром
полезные действия
возврат результата функции
конец описания метода

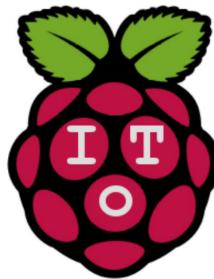
создать объект класса

выполнить метод объекта

выполнить другой метод объекта



Методы объектов класса

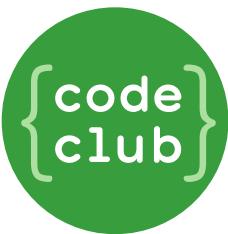


```
def method(p1, p2)      # описание метода с двумя параметрами (в классе Object)
    result = p1 + p2     # команды в теле метода
    return result         # вернуть результат действий
end                      # конец описания метода

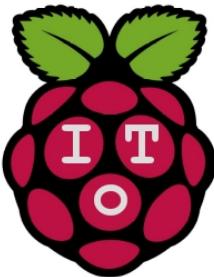
method(40, 2)            # вызвать метод (возвращаемое значение отбросить)
r = method(40, 2)         # вызвать метод (возвращаемое значение поместить в r)

class LED
    def dot              # описание метода без параметров для класса LED
        on_for(0.25)       # включить светодиод на 1/4 секунды
        off_for(0.25)      # выключить светодиод на 1/4 секунды
    end                    # конец описания метода
end

led = LED.new(18)          # создать объект led класса LED с параметром 18
3.times { led.dot }       # 3 раза вызвать для объекта led метод dot()
```



Выполнение метода



```
# Описание метода
def method(parameter)
    print(parameter)
    return 3.14
end
```

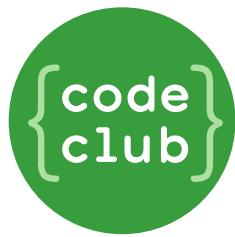
3

1 # команда перед вызовом метода
berry = "Raspberry"

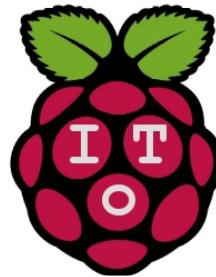
2 # вызов метода = его выполнение:
pi = method(berry)

4 # команда после вызова метода
print(pi) # 3.14

Выполнение метода



Классы: пример разработки



Требуется посчитать количество «счастливых» билетов в катушке с номерами от 0 до 999999 и напечатать номера этих билетов.

Для этого нужно: (1) описать класс с требуемыми характеристиками (диапазон номеров билетов) и (2) определить необходимые действия (сформировать список билетов со «счастливыми» номерами).

```
class RollOfTickets
# ...
end

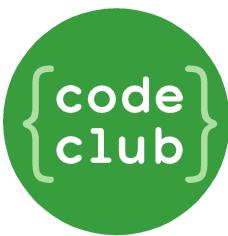
roll = RollOfTickets.new(0, 999999)

lucky_tickets = roll.lucky

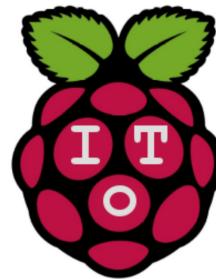
print lucky_tickets.size

print lucky_tickets
```

описание класса «КатушкаБилетов»
описание методов методов
действия по достижению цели:
новая катушка из 1_000_000 билетов
метод lucky находит счастливые билеты
вывести количество счастливых билетов
вывести список счастливых билетов



Классы: метод-инициализатор



В описании класса «КатушкаБилетов» предусмотреть требуемые атрибуты: нижнюю и верхнюю границы диапазона номеров для билетов в катушке.

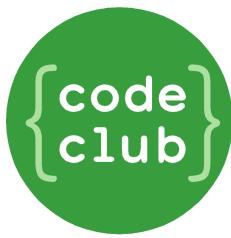
```
class RollOfTickets
  attr :lo
  attr :hi

  def initialize(lo=0, hi=999999)
    @lo = lo
    @hi = hi
  end
end

roll_1 = RollOfTickets.new(0, 999)           # новая катушка из 1_000 билетов

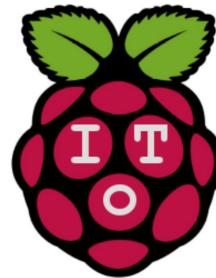
roll_2 = RollOfTickets.new                   # новая катушка из 1_000_000 билетов
                                              # (по умолчанию: lo=0, hi=999999)

roll_3 = RollOfTickets.new(0, 99999)         # новая катушка из 100_000 билетов
```



Классы:

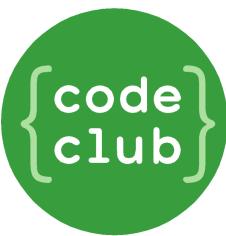
Открытые методы



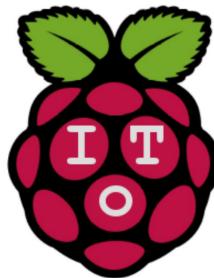
В описании класса «КатушкаБилетов» определить необходимый метод: сформировать список билетов со «счастливыми» номерами.

```
class RollOfTickets
public
  def lucky()                                # в описание класса добавить
    tickets = []                             # public - «открытые» методы для вызова без ограничений
    for i in (@lo..@hi)                      # метод поиска «счастливых билетов»
      if (self.lucky?(i))                    # массив для найденных билетов
        tickets << i                        # искать от нижней до верхней границы
      end                                     # метод проверки на «счастливость»
    end                                      # добавить найденный билет массив
    return tickets                            # вернуть список найденных билетов
  end
end

roll = RollOfTickets.new(0, 999)
lucky_tickets = roll.lucky                 # roll - новая катушка из 1000 билетов
                                            # метод lucky находит счастливые билеты
```



Классы: зашитённые методы



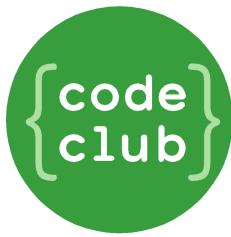
В классе «КатушкаБилетов» удобно определить вспомогательный метод, который будет проверять, «счастливый» ли номер у конкретного билета.

```
class RollOfTickets          # в описание класса добавить
  protected    # protected - «зашитённые» методы для вызова из объектов класса
  def lucky?(i)           # метод проверки на «счастливость»
    h1, h2, h3 = head(i)   # разбить 1-ю половину целого на 3 числа
    t1, t2, t3 = tail(i)   # разбить «хвост» целого на 3 числа
    return ((h1+h2+h3) == (t1+t2+t3))
  end
end
```

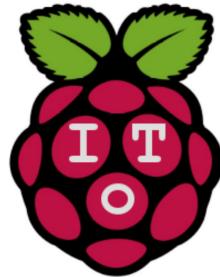
Для такой проверки можно разделить число на 2 половины, затем каждую из них разбить на отдельные цифры, чтобы посчитать их суммы для сравнения.

Можно реализовать другой подход — написать метод, сразу подсчитывающий суммы первой и второй половин числа:

```
(sum_left, sum_right) = calculate_sums(i)
```



Классы: закрытые методы

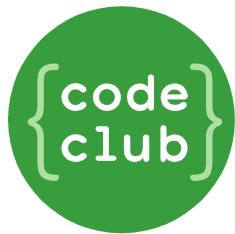


В классе «КатушкаБилетов» можно определить вспомогательные методы, которые будут разбивать число на отдельные цифры.

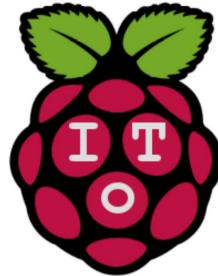
```
class RollOfTickets          # в описание класса добавить
  private      # private - «закрытые» методы для вызова только в этом объекте
    def head(i)
      h = (i/1000).to_i           # 123456
      return (h/100).to_i, ((h % 100)/10).to_i, ((h % 100)%10).to_i # 1,2,3
    end

    def tail(i)
      t = i % 1000               # 123456
      return (t/100).to_i, ((t % 100)/10).to_i, ((t % 100)%10).to_i # 4,5,6
    end
  end
```

Можно реализовать универсальный способ разбивки любого целого числа на 2 половины, а не только чисел, у которых по 3 цифры в левой и правой частях.



Классы: методы класса



В описании класса можно определять методы для управления поведением класса, а не отдельных объектов: например, добавочный конструктор объектов класса.

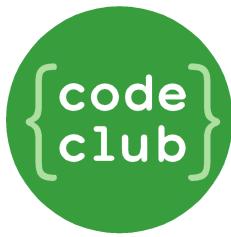
```
class RollOfTickets # в описание класса добавить

  def RollOfTickets.new_roll(n) # для создания катушки по количеству
    RollOfTickets.new(0, n-1) # вызываем конструктор с параметрами
  end

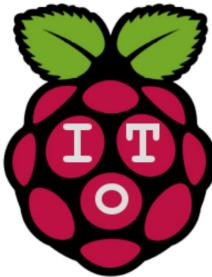
end

roll = RollOfTickets.new_roll(1000000) # новая катушка из 1000000 билетов

print roll.lucky().size() # найти количество счастливых билетов
```

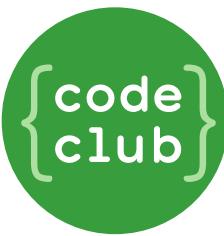


ООП: разработка «сверху вниз»

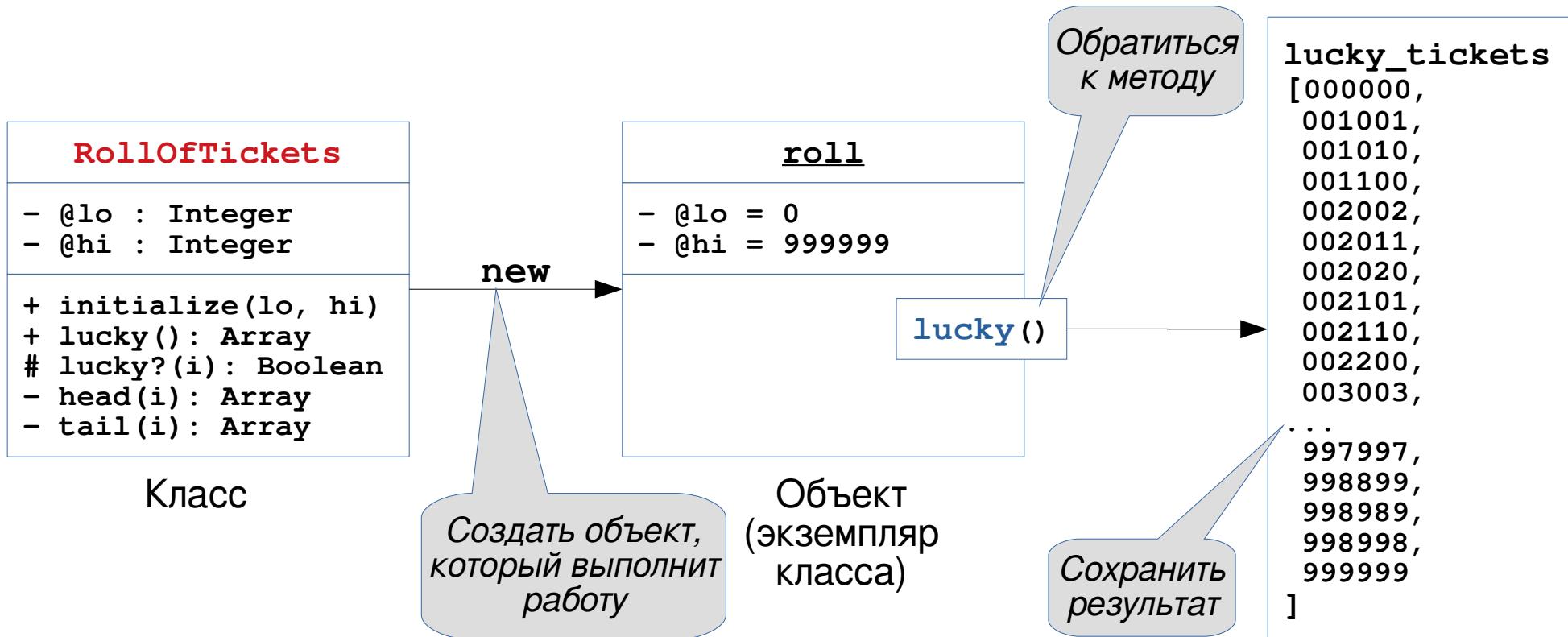
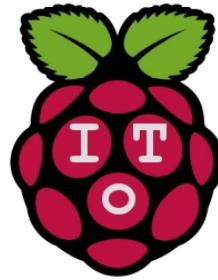


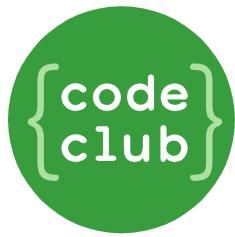
1. Придумать класс (с подходящим именем), в котором будут описаны требуемые характеристики и необходимые действия.
2. В классе описать атрибуты (переменные объектов), в которых будут храниться значения требуемых характеристик (свойств объектов).
3. Описать метод-инициализатор (`initialize`), который при создании каждого объекта класса будет задавать начальные значения атрибутов (и выполнять другие начальные действия).
4. Описать открытые методы, которые будет выполнять полезные действия с каждым из объектов, изменять значения его атрибутов (состояние этого объекта).
Набор открытых методов представляет из себя интерфейс объекта, через который можно управлять поведением каждого из объектов класса.
5. Описать защищённые и закрытые методы (при необходимости), которые будут выполнять скрытые действия по изменению состояния объекта или другие вспомогательные действия.
6. Создать нужное количество объектов этого класса с помощью конструктора new, при необходимости передавая ему параметры для инициализации объектов.
7. Управлять созданными объектами с помощью методов, чтобы получить требуемый результат.

Способ разработки программы от крупных частей к более мелким составляющим частям («сверху вниз») также называют методом пошагового уточнения (*stepwise refinement*).

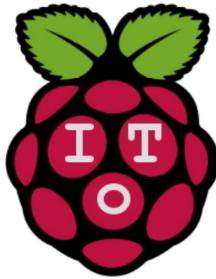


ООП: моделирование реальных объектов



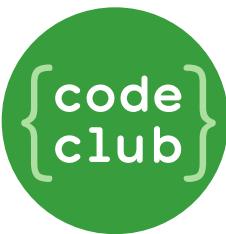


ООП : преимущества

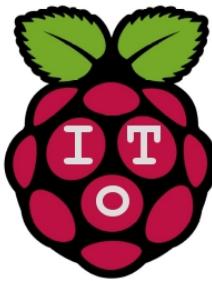


1. Объектно-ориентированный подход позволяет разработчику думать о решении задачи, применяя обычные понятия о разновидностях объектов (светодиоды, кнопки, датчики, реле, моторы), присущих им характерных свойствах (название, цвет, номера контактов, значения показаний), возможных действиях с ними (включить, выключить, считать показания, изменить состояние, увеличить скорость).
2. Разбивать задачу на обозримые части и постепенно разрабатывать решения для каждой из них.
3. Бороться со сложностью задачи, скрывая подробности в разных уровнях детализации при описании системы классов.
4. Создавать программную модель реальной системы, имитируя взаимодействие между её частями при помощи подходящих методов.

ООП даёт разработчику возможности писать программы быстрее, с меньшим количеством ошибок, большего размера.



Выполнение блока



Описание метода

```
def method
```

```
    yield # выполнение блока
```

```
end
```

Выполнение метода

2

1

3

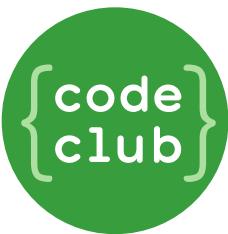
Выполнение блока

```
method
```

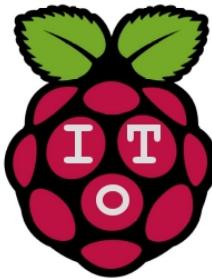
```
{ p "Блок 1.1" } # блок
```

вызов метода с параметром-блоком

команда после вызова метода



Блоки с параметрами



При вызове метода блоку можно передавать значения параметров (аргументы) так:

```
method { |parameter| do_something_with(parameter) }
```

или так:

```
do |parameter|
  do_something_with(parameter)
end
```

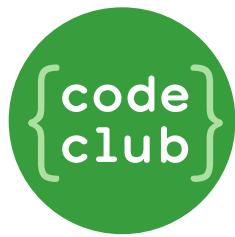
В методе параметр передаётся блоку так:

```
yield(parameter)
```

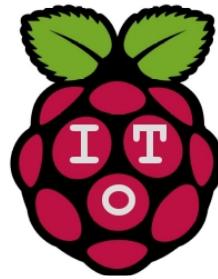
```
# метод с параметром
def method_2(parameter)          # начало описания метода
  print "> Начало работы метода 1\n" # команды в теле метода
  yield(parameter)                # выполнить блок с параметром
  print "< Конец работы метода 1\n" # команды в теле метода
end                                # конец описания метода
```

```
method_2("Блок 2.1") { |p1| print "---"+p1+"---" }
```

```
method_2("Блок 2.2") { |p2| print "===="+p2+"====" }
```



Справочник и учебник по Ruby

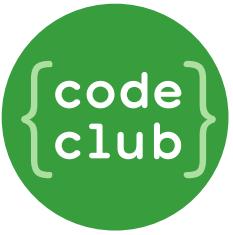


Краткий справочник по синтаксису языка — в файле:

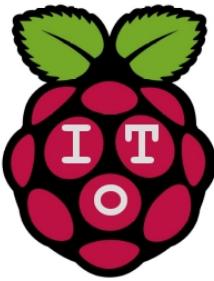
IoT-Ruby_syntax.pdf

Учебник для начинающих по языку — в файле:

~/Documents/books/Learn_To_Program-Ch.Pine-ru.pdf



ООП



*Как вы понимаете объектно-ориентированное
программирование?*

Почему сегодня объектный подход широко применяется?