From absolute beginner to advanced

Roman Pushkin

RUBY IS FOR FUN

Ruby Is For Fun

Self-Study Course, From Absolute Beginner to Advanced

Roman Pushkin

This book is for sale at http://leanpub.com/rubyisforfun

This version was published on 2022-05-26

Contents

Front Matter

In Lieu Of an Introduction

In the 21st century, programming has become one of the most important sciences in any economy. Processes which used to take place without the aid of computers have been partly or completely optimized. Businesses and private individuals have realized how useful electronic machines are, and the age of a flourishing IT industry has begun.

Certain specific trends have formed within all this variety of technologies. The most convenient tools for carrying out particular tasks have been determined. Programming languages have undergone significant changes. It is not as easy for the ordinary reader to understand all these languages and technologies as it might appear at first glance.

It became obvious at a certain point that “programmer” is one of the professions of the 21st century. But how do you become a programmer? In which direction should you apply your efforts? What needs to be studied, and what does not? What is the most efficient use of your time in mastering any technology?

Before answering these questions, the most important question of all must be answered. Why is it necessary to become a programmer? What’s the sense of it?

Some will want to become a programmer to develop micro-programs for inter-continental ballistic missiles and the space industry. Some will want to become a programmer to create their own games. Some will want to learn programming in electronic tables to calculate taxes more efficiently.

But the purpose of this book is more mundane. The author assumes that to the question “Why do I need to become a programmer?” you will give the answer “To earn money as a programmer.” Such an answer is usually given by people who have already tried some profession and want to use their time more efficiently to earn good money.

They could also be young people forced to keep up with the times and to learn technologies, and the quickest way of getting a result from such knowledge, as quickly as possible. Furthermore, this means a result not only in the form of the knowledge of how to write this or that program, but a result in terms of money.

Knowledge of any field in programming assumes a knowledge of the basics of the language, along with elementary theory (which is different for each field), basic concepts and definitions, and also a knowledge of tools other than the basic ones (such as an operating system, utilities and auxiliary programs).

There is a vast number of fields. They include game development, scientific research of various kinds, data processing and analysis, web programming, programming for mobile devices, and so on. It is not possible to be a specialist in all these at the same time.

So the person starting, or wishing to start, on programming faces a choice. Where to apply? What to study?

If you are a scientist at a research institute, your choice will most likely fall on the language “python” or “C++”, since a large number of libraries for data processing and analysis exists for these languages.

If, for example, you work as a watchman and are quite content with your work, you could study some exotic programming language that’s not high in demand, simply to avoid being bored.

If you live in a society in which ever-increasing accounts are paid every month, where you need to think not only of today, but of tomorrow, your choice will be different.

You will need to study something in high demand very quickly, to find work as soon as possible.

The language "Ruby", used in web programming, is something between "finding work quickly", "studying something simple and interesting" and "will come in useful in the future". Ruby not only enables you to compile boring programs while working in an office, but can also be useful at home, in everyday life. (One of my recent programs was about teaching how to play the guitar).

Furthermore, the philosophy of the language itself assumes that the study and use of programs will not be boring. For example, one of the principles of the language is the Principle of Least Surprise, which is explained as follows: "Whatever you do, you will most likely succeed". That is inspiring, you must agree!

Other programming languages also exist. I am not saying they are bad, not at all. Each language is good for a certain purpose. But let us remember our own task and make the comparison with some other languages.

Ruby vs. Ybur

The language "Ybur" is Ruby in reverse. It is an exotic programming language which no-one knows except me. I've only just thought it up and I don't know what it does. Let us compare Ybur and Ruby using the three parameters described above.

Finding work quickly

Ruby is a very popular language, it is easy to find work where it is used. Ybur? No-one's ever heard of it, finding work using it is impossible.

There is no need to compare the other parameters. In other words, if what is important to you is not programming in itself (though that's no bad thing either), but the possibility of earning money in the foreseeable future, Ruby is not a bad choice. It is quite a popular language. Of course, other popular programming languages exist too. We might say that JavaScript is the most popular. But let us compare JavaScript and Ruby.

Learning something simple and interesting

Ruby incorporates the Principle of Least Surprise, and that is not at all bad. JavaScript was not initially created in accordance with this principle. It is more complicated than Ruby because it is completely asynchronous (you'll have to take my word for that for the time being).

We can show that JavaScript is not as simple as it looks at first glance. Let us consider a Ruby program for sorting numbers:

Example: Simple program to sort four numbers in Ruby

```
[11, 3, 2, 1].sort()
```

The above program has to sort the numbers 11, 3, 2 and 1 into rising order (it is not important for now if you don't understand this syntax, we'll deal with that subject later). The result of the Ruby program's work is: 1, 2, 3, 11. No surprise there! But let us write the same program in JavaScript:

Example: Incorrect program to sort four numbers in JavaScript

```
[11, 3, 2, 1].sort();
```

In this case, the syntax is very similar, and differs only in the semicolon at the end. But what will the result be? Even experienced JavaScript programmers cannot always give the correct answer. The program's results are quite unexpected: 1, 11, 2, 3. Why this is so is a question of history. But to sort numbers in JavaScript, you have to write:

Example: Correct program to sort four numbers in JavaScript

```
[11, 3, 2, 1].sort((a, b) => a - b);
```

Once you understand it, it isn't difficult. But the question is something else. Do you want to waste time on such fine points in the initial stage? JavaScript is much in demand, and every Ruby programmer must know it at a minimal level. But I must say I would want a *very great deal of money* to be a full-time JavaScript developer.

Could come in handy in the future

JavaScript is developing very dynamically. Knowledge gained ten years ago is not always up-to-date (in this case I am speaking of popular frameworks - sets of tools). In Ruby's case, the Rails framework has existed for more than ten years. Knowledge gained ten years ago is still applicable.

Incidentally, it is worth making a separate comment about the applicability of knowledge. Knowledge of shell-scripting languages is still applicable. Little has changed in more than 30 years. Knowledge of the basics of computer science is still applicable in interviews and at work (this knowledge has hardly aged at all). So it is definitely something you want to learn at some point of time.

But no-one can make precise predictions about the applicability of a particular *programming language* in the future. However, one may look at the statistics of recent years. At the time this book was being written, Microsoft bought GitHub, written in Ruby, for 7.5 billion dollars. In other words, the language is in fine form today and widely used. Updates are being issued, the speed and syntax are being improved. And the number of available libraries makes a rapid solution of virtually any problem possible (within the framework of the field called web programming).

Ruby Is For Fun

In our opinion, programming language should not only solve certain business problems, but should also be easy enough to use every day without problems.

For example, Java is a fine tool for solving business problems. But it has to be treated with respect. The language is statically-typed (we'll come back to this topic). The type of data on which different operations are carried out must be specified. This takes time, and is fully justified in the business field, where it is better to spend several times as long on development, rather than having to pay for mistakes later.

In the case of Ruby, the program can be written quickly, in a simple way. It is not all that reliable (which can be a problem sometimes), but many companies, particularly startups, have come to the conclusion that it is *reliable enough*, and the relatively slow speed of program execution is not a problem either. After all, in today's world we often have to do something quickly to get quick investments, to attract the first customers, take advantage of momentum while competitors are still trying to find their way.

From the author's personal point of view, Ruby is a good tool for doing something of my own - some project of mine, a program which can be shared with others, attract attention or earn money.

In other words, Ruby is an efficient and interesting language, not only for work but for its own sake too.

What will we study?

As stated earlier, there are many trends in software development. Each trend is unique and requires its own know-how. The author believes that there are currently at least two "tried and tested" trends in programming which give the maximum result in the minimum time. "Results" here means both cash compensation and knowing how to do something with your own hands.

The first is mobile development – programs for mobile phones (Android, iPhone), tablets (iPad) and other devices.

The second is web programming (or web development).

However, mobile development itself often means optimizing the code for mobile devices in all sorts of ways. The programming language and SDK (software development kit) is very often bound up with a certain style of development. And this style is very different from the *classic* object-oriented programming, it is more procedural. With procedural programming, you can't always make full use of the language's capabilities, though this is not always important, particularly if your aim is to earn a salary.

A second aspect of program development for mobile devices is that there are two main mobile platforms at the present time. One belongs to Apple Corporation, the other to Google. How these platforms will develop in the future depends entirely on the policy of these companies.

In the case of web programming in Ruby, it all looks somewhat different. The language itself is being developed and supported by the programmers themselves. The web framework Rails – about which more later – is also supported exclusively by the community. This enables programmers from all over the world to create a convenient tool just as they want, without having to look over their shoulder at the policy of any company.

Furthermore, programming in Ruby is rarely used on mobile devices, therefore in practice, they hardly ever have to be "specially" optimized. But the main difference between Ruby and the mobile development languages is that Ruby is a dynamic language – not in the sense that it is developing dynamically (though it is), but that it includes what is called *dynamic typing* of data as mentioned earlier.

The main advantage of dynamic typing compared with static is that there are fewer rules and less strictness, which gives a higher rate of development of apps by the programmer (admittedly at the cost of slower performance of the written programs, and of "sufficient" reliability. But performance rate does not particularly interest us, since Ruby is not used for developing mobile apps, although it is more often than not a key link on the server and facilitates the functioning of mobile apps for iOS, Android, etc.).

No doubt other programming trends not checked by the authors of this book do exist. For example, the development of computer games. A lifetime would probably not be enough to "check" all the trends, so we will leave this task to more inquisitive minds, and restrict ourselves to what is actually in demand in the market, permits "rapid input" and is more or less interesting rather than boring.

Web Programming Or Something Else?

The book "Ruby is for fun" is divided into two parts. In Part One (which you are reading now), we consider the basis of the Ruby language and its use from the so-called command line. Part Two (upcoming) will include web programming and the Rails framework.

"*Wait!*" the observant reader will say. "*Surely we've just been talking about web programming? Yet it's going to be in Part Two as well?*"

Quite true. The point is that Ruby is quite a powerful tool in itself. Students of the online Ruby school have found work even without knowledge of web programming. The basics of the language and the ability to find and use the required libraries already make possible the creation of quite useful apps which can be used for data processing (for example, so-called "web-scraping") to create script configurations and to control an operating system (which will definitely come in handy for any system administrator), for working with files of different formats etc.

The ability to use a language for jobs of various kinds not connected with web programming gives an indisputable advantage before you start web programming. Web programming itself involves the knowledge of certain generally accepted concepts. And we'll now be able to solve these problems using a tool we are already learning how to handle.

...

The full version of the book is available at http://rubyisforfun.com[1]

...

Everything Is An Object (Proof)

We know that `123.class` returns *Integer*, and `"blabla".class` returns *String*. But any object also has `is_a?` method, which returns *true* or *false*, depending on parameter you're passing to this method:

```
$ irb
> 123.is_a?(Integer)
 => true
```

In example above we're calling `is_a?` method for `123` object and passing parameter `Integer`. Method returns true. In other words, `123` is a type of *Integer*. The similar way we can test if `123` is `String`:

[1]http://rubyisforfun.com

```
$ irb
> 123.is_a?(String)
 => false
```

Answer is false, `123` is not *String*, but `"123"` (in quotes) is *String*:

```
$ irb
> "123".is_a?(String)
 => true
> "blabla".is_a?(String)
 => true
```

By calling `is_a?` we're kinda asking question in plain English "**Is** this **a**… ?", "Is this object a string?"

We've confirmed that `123` is *Integer*, and `"blabla"` is *String*. Now let's make sure integers and strings are objects:

```
$ irb
> 123.is_a?(Object)
 => true
> "blabla".is_a?(Object)
 => true
```

Great, numbers and strings are objects in Ruby! In other words, `123` is *Integer* and *Object* at the same time. And `"blabla"` is *String* and *Object* at the same time. In other words, there can be multiple objects, and objects *can implement* multiple types.

We'll discuss later in this book what object really is. We don't need to remember `is_a?` method at the moment, how to call this method, and what it returns (in computer literature it is called "method signature", sometimes API - Application

Program Interface). But it's worth remembering `.class`, so you can any time check the type of any object (or the type of result of some operation).

...

The full version of the book is available at http://rubyisforfun.com[2]

...

Blocks With Parameters

Object that executes your block can pass a parameter. It depends on implementation, some objects do pass parameters to your blocks, others don't. It's up to a programmer if you want to use this parameter or want to ignore it. So far, we've been always ignoring this information, but technically parameter was always there. Let's see how we can use it.

In case below we have object of `24` with type *Integer*. Let's write a program called "Piggy Bank", this piggy bank will accept 500 dollars every month, and will save this money. What we want to know is how much money it has after 24 months.

Piggy Bank program, version 1

```
sum = 0

24.times do |n|
  sum = sum + 500
  puts "Month #{n}, piggy bank has #{sum} dollars"
end
```

Result:

[2] http://rubyisforfun.com

```
Month 0, piggy bank has 500 dollars
Month 1, piggy bank has 1000 dollars
Month 2, piggy bank has 1500 dollars
Month 3, piggy bank has 2000 dollars
...
Month 22, piggy bank has 11500 dollars
Month 23, piggy bank has 12000 dollars
```

Well, result was little bit unpredictable. For some reason the count starts from zero: "Month 0". Actually, it was expected, and the only confusing part is the variable naming. Usually "n" means "natural number", and natural numbers can be used for counting[3] (one apple, two apples, three apples, ...). But in our case *n* can be `0`, and it can confuse some programmers.

Usually one uses *n*, *m*, etc. for natural numbers. And if we're talking about *index*, it starts with zero, and the more appropriate name for variable will be *i*, *j*, and so on. It's not a mistake if you named variable incorrectly (and definition of correctness is only up to you), but the code has two readers: computer and human. And human is not only you, but also somebody else: your colleagues, friends, random people online who came across your published code, and also *the future you*. That's why you need to write the most predictable code you can. Moreover, Ruby is about least surprise, and community expects the same from other Ruby developers.

With this in mind we can rewrite the program:

[3]However, some definitions begin natural numbers with `0`. This situation is strange because mathematics is normally a very precise science and there is normally broad agreement about such definitions.

Piggy Bank program, version 2

```ruby
sum = 0

24.times do |i|
  sum = sum + 500
  puts "Month #{i}, piggy bank has #{sum} dollars"
end
```

But renaming variable is not enough, months still start from 0:

```
Month 0, piggy bank has 500 dollars
...
```

We don't count apples starting from zero, and practically speaking there is no too much sense in counting months starting from zero. So let's take advantage of string interpolation and replace expression `i` with `i + 1`:

Piggy Bank program, version 3

```ruby
sum = 0

24.times do |i|
  sum = sum + 500
  puts "Month #{i + 1}, piggy bank has #{sum} dollars"
end
```

Now it works as expected:

```
Month 1, piggy bank has 500 dollars
...
Month 24, piggy bank has 12000 dollars
```

For the sake of experiment imagine you have not just a piggy bank, but “Magic Piggy Bank, model 10”! It will generate the revenue of 10% every month for any money you put in:

Magic Piggy Bank program

```
1  sum = 0
2
3  24.times do |i|
4    sum = sum + 500 + sum * 0.1
5    puts "Month #{i + 1}, magic piggy bank has #{sum}"
6  end
```

So the improvement is only to add “`+ sum * 0.1`” on line 4, and here is result:

```
Month 1, magic piggy bank has 500.0
Month 2, magic piggy bank has 1050.0
Month 3, magic piggy bank has 1655.0
Month 4, magic piggy bank has 2320.5
...
Month 23, magic piggy bank has 39771.512162761865
Month 24, magic piggy bank has 44248.66337903805
```

In other words, Magic Piggy Bank will generate $44248 instead of just saved $12000.

Exercise 1

You're planning to buy a house for $500.000 and your employer credit union is offering zero percent interest rate for 30 years. To pay off this house you need to pay $16.666 every year (it can be calculated by dividing $500K by 30). Write a program which will show the amount left to pay for each year.

Exercise 2

Modify the program from exercise 1 to meet the following criteria. Your interest rate is 4% a year for *remaining* amount. For every year calculate how much *interest in dollars* you need to pay for using bank's money.

Methods Of Integer Class

There is no too much methods for *Integer* class, and it's worth looking at documentation to get a better understanding of what's there available for a programmer. However, we'll take a close look to some of them.

`even?` and `odd?`

`even?` or `odd?` – two methods of *Integer* class, method names "come" with question mark at the end.

We can check any *Integer* if it's even or odd using two methods above. Since question mark at the end of the method comes up for the first time in this book, let's see why it's there.

Question mark just indicates that method returns type *Boolean*. Well, technically there is no any *Boolean* in Ruby, we've just introduced it for the purpose of this book.

There are two actual types you will see: *TrueClass* and *FalseClass*. In other words, there is `true` and `false` keywords in Ruby language, and you can use these keywords (see examples below).

Something unambiguous in real life can be represented by true or false, usually these methods start with prefix "`is_`". For example:

```
girlfriend.is_pregnant?
```

There is no any other options possible when you're using boolean, it's either true or false. Question mark is optional in Ruby language, but expected by community. It's highly recommended to have "`?`" and the end of methods that return boolean value.

Let's look at example:

```
$ irb
> 1.even?
false
> 1.odd?
true
> 2.even?
true
> 2.odd?
false
> 10 % 2 == 0 # our own implementation of "even?" method
true
```

The last line from our REPL dialog has two parts:

- `10 % 2` - divide by two, and returns remainder (which will be `0` or `1`)
- `== 0` - comparison with zero, returns true or false

`upto` and `downto`

`upto` and `downto` methods of *Integer* class accept parameter and call provided block for certain amount of times. We're already familiar with `times` method, which starts counting from zero. You can do the same with `upto`. For example:

```
> 3.times { |i| puts "I'm robot #{i}" }
I'm robot 0
I'm robot 1
I'm robot 2
...
> 0.upto(2) { |i| puts "I'm robot #{i}" }
I'm robot 0
I'm robot 1
I'm robot 2
```

The output is the same, but `upto` is more flexible, because it accepts parameter (`2` in program above). With this parameter we can specify "from" and "to" values, like:

```
> 1000.upto(1002) { |i| puts "I'm robot #{i}" }
I'm robot 1000
I'm robot 1001
I'm robot 1002
```

`downto` is similar, but it counts backwards:

```
puts "Launching missiles..."
5.downto(1) { |i| puts "#{i} seconds left" }
puts "Boom!"
```

Result:

```
Launching missiles...
5 seconds left
4 seconds left
3 seconds left
2 seconds left
1 seconds left
Boom!
```

Well, of course you can do block with `do...end` and result will be the same:

```
puts "Launching missles..."
5.downto(0) do |i|
  puts "#{i} seconds left"
end
puts "Boom!"
```

Exercise 1

Put all numbers to the screen starting from 50 to 100.

Exercise 2

Put all numbers to the screen starting from 50 to 100 with parity next to the number. If number is even, print "true", if it's odd, print "false".

. . .

The full version of the book is available at http://rubyisforfun.com[4]

. . .

[4]http://rubyisforfun.com

Part III. Having Fun

Ternary Operator

Ternary operator is quite useful feature of any language, and it's implemented in Ruby the same way as it is implemented in C, Java, Python, JavaScript and so on. Some programmers use this operator for a long time, but don't familiar with its name. But we recommend to remember its name, because it's always nice to use correct naming and show your knowledge when speaking to a colleague:

> Dear colleague, why don't we just replace this beautiful branching with ternary operator?

Despite the scary name, syntax is pretty straightforward:

```
something_is_truthy ? do_this() : else_this()
```

For example:

```
is_it_raining? ? stay_home() : go_party()
```

Which is 100% the same as the following code, but with "if…else":

```
if is_it_raining?
  stay_home()
else
  go_party()
end
```

Note that you can always omit empty parentheses in Ruby in favor of code readability. Parentheses just indicate that we're calling a method with no arguments (parameters). But Ruby is smart enough to understand what we mean:

```
if is_it_raining?
  stay_home
else
  go_party
end
```

Or just:

```
is_it_raining? ? stay_home : go_party
```

Note double question mark in one-line above. We did in intentionally, and we just assume that there is a method with question mark defined somewhere, like this (we'll cover methods definition later in this book):

```
def is_it_raining?
  # ... do something, like call to external weather service ...
end
```

Method `is_it_raining?` always returns "true" or "false", in other words *Boolean* type (we invented this type in previous chapters). And Ruby's naming conventions say that if method returns *Boolean*, we should define this method with question mark at the end, so it looks like question in plain English: “Is it raining?”.

But if result depends on some variable, one-liner ternary operator may look like (note one question mark only, the mandatory question mark for ternary operator):

```
x ? stay_home() : go_party()
```

Or just:

```
x ? stay_home : go_party
```

As you can see from examples above, ternary operator has more compact syntax and can save us couple of lines on the screen – your program will look less lengthy, and (hopefully) more readable to you and your fellow colleagues. But the disadvantage of ternary operator is that it only looks neat when you need to perform one instruction. It’s better to use “if…else” when you need to do something complex.

Result of operation can be also saved to variable. In the example below we save result of ternary expression to `result` variable:

```
x = is_it_raining?
result = x ? stay_home : go_party
```

`result` will now keep the return value of `stay_home` or `go_party` method. In example above `result` could be the number of drinks one drunk[5]: “if I stay home, I will watch Netflix and will drink from 0 to 1 beer; but if I go party I will drink from 1 to 2 beers”. So `result` will keep just the number in our example, and `x` variable will hold “true” or “false” indicating if it is raining outside.

Two lines above can be rewritten with “if…else” clause:

[5]We hope you’re yoga-mantras-smoothie-falafel person and don’t drink alcohol?

```
x = is_it_raining?
result = if x
  stay_home
else
  go_party
end
```

Exercise

Rewrite the following examples with ternary operator:

Example 1:

```
if friends_are_also_coming?
  go_party
else
  stay_home
end
```

Example 2:

```
if friends_are_also_coming? && !is_it_raining
  go_party
else
  stay_home
end
```

...

The full version of the book is available at http://rubyisforfun.com[6]

...

[6]http://rubyisforfun.com

Judgement Day Emulator

Let's practice a bit to sum up everything we know about methods. The machines have taken over the world. The is a struggle for survival. Who will survive: humanity or machines? Now it's up to destiny to decide our future. Well, actually up to the random number generator.

Program will show the steam of upcoming events happening in the world. It would be much more interesting if we could do it by using some graphics, but in our case it will all depend on observer's imagination. One may like or program and even use it as screensaver.

Important note: this program can be done the different and better way. But we still have limited Ruby knowledge, so our goal is more to practice, rather than delivering a piece of art.

First, let's agree that there are only `10000` humans and the same number of machines left. With every iteration we'll have only one random event. And the number of humans and machines will be decreasing with the same probability. Victory is when there are no more machines (or humans) left. Let's proceed with a solution.

We'll start with definition of victory and the main loop with two variables:

```
humans = 10_000
machines = 10_000

loop do
  if check_victory?
    exit
  end
  ...
end
```

Variables `humans` and `machines` represent the information about survivors.

Method `check_victory?` returns *Boolean* type value if one side has won (it doesn't matter which one). If there is a victory, we just `exit` the program. If there is no victory, we continue iteration inside of infinite loop. Let's also print information about who actually won the battle inside of this method.

Now we need to define a couple of events that can happen. Let's call them `event1`, `event2`, and `event3`. We'll be calling this or another method depending on random number provided by `rand`. It's like throwing dice with only 3 distinct values from 1 to 3:

Sketch of the program we're going to make

```
def event1
  # ...
end

def event2
  # ...
end
```

```
def event3
  # ...
end

# ...

dice = rand(1..3)

if dice == 1
  event1
elsif dice == 2
  event2
elsif dice == 3
  event3
end
```

We used `elsif` keyword above (we're already familiar with `else`). `elsif` is one of the less intuitive abbreviations in Ruby, and it just means `else if`.

And we'll finish our loop with `sleep` statement, which will delay the execution of our program by certain amount of seconds (from `0.3` to `1.5`):

```
sleep rand(0.3..1.5)
```

Complete version of the program:

Judgement Day Emulator, version 1

```
########################################
# DEFINE VARIABLES
########################################

@humans = 10_000
@machines = 10_000

########################################
# AUXILIARY METHODS
########################################

# Method returns random value: true or false
def luck?
  rand(0..1) == 1
end

def boom
  diff = rand(1..5)
  if luck?
    @machines -= diff
    puts "#{diff} machines destroyed"
  else
    @humans -= diff
    puts "#{diff} humans killed"
  end
end

# Method returns random city name
def random_city
```

```ruby
  dice = rand(1..5)
  if dice == 1
    'San Francisco'
  elsif dice == 2
    'Moscow'
  elsif dice == 3
    'Beijing'
  elsif dice == 4
    'London'
  else
    'Seoul'
  end
end

def random_sleep
  sleep rand(0.3..1.5)
end

def stats
  puts "#{@humans} humans and #{@machines} machines left"
end

########################################
# EVENTS
########################################

def event1
  puts "Missile launched against #{random_city}"
  random_sleep
  boom
```

```
end

def event2
  puts "Nuclear weapon used against #{random_city}"
  random_sleep
  boom
end

def event3
  puts "Group of soldiers have breached defence in #{random_city}"
  random_sleep
  boom
end

########################################
# CHECKING VICTORY
########################################

def check_victory?
  false
end

########################################
# MAIN LOOP
########################################

loop do
  if check_victory?
    exit
  end
  dice = rand(1..3)

  if dice == 1
    event1
  elsif dice == 2
    event2
  elsif dice == 3
    event3
  end

  stats
  random_sleep
end
```

Sample result:

```
Nuclear weapon used against London
3 machines destroyed
10000 humans and 9997 machines left
Group of soldiers have breached defence in London
2 machines destroyed
10000 humans and 9995 machines left
Nuclear weapon used against Seoul
4 humans killed
...
```

Exercise 1

Implement `check_victory?` method (now it just returns `false`). In case of victory it should show information about who actually won the battle and who lost. Replace `10_000` with `10` so it will be easier to work on this program (you won't need to wait too long to see how your method works).

Exercise 2

Lookup documentation for "ruby case statements" and replace `if...else` with `case...when`.

Exercise 3

Improve the program, so every iteration we not only lose, but also get random number of humans and machines. Theoretically, this battle may never end, but with low initial number of humans and machines at the beginning (like 10 or 100 instead of 10000) there can be a case when one side randomly wins.

Exercise 4

Improve the program and add at least 3 events, so your output looks more interesting. Use your imagination.

. . .

The full version of the book is available at http://rubyisforfun.com[7]

. . .

[7]http://rubyisforfun.com

Programming Slot Machine

We'll practice a bit and sum up everything we know so far, and implement simple one-armed bandit game. We will put some money in the bank, pull the handle and see what happens.

Let's outline our plan. Variable `balance` will represent the amount of money we have in the bank. We'll have three reels (positions, or slots) and few images. Traditionally, slot machines have fruit images, cherry, bell, and number "7". To simplify our task we'll be just using numbers from 0 to 5 instead of mentioned images - we can always come back to it and make improvements later.

Variables `x`, `y`, `z` will represent the reels. Each reel can hold only one value (from 0 to 5) at a time. This value will be defined by random number generator.

Next step is to define what "win" and "lose" means: when all three variables `x`, `y`, `z` are equal, we'll be enforcing following rules:

- If all numbers are zeroes, we lose everything
- If all variables are ones, we add $10 to our balance
- If all variables are twos, we add $20 to our balance
- Otherwise we charge 50 cents for attempt

Program should work until we have money on our balance. Elementary age check will be helpful:

```
print "What's your age: "
age = gets.to_i
if age < 18
  puts 'Sorry, but you should be at least 18 to play'
  exit
end
```

`balance` variable keeps the balance (initialized with $20), also define the infinite loop:

```
balance = 20
loop do
  # ..
end
```

Wait for Enter key inside of the loop:

```
  puts 'Press Enter to pull the handle...'
  gets
```

Initialize variables `x`, `y`, `z`:

```
  x = rand(0..5)
  y = rand(0..5)
  z = rand(0..5)
```

Print result of a single draw:

```
  puts "Result: #{x} #{y} #{z}"
```

Check first condition "If all numbers are zeroes, we lose everything":

```
if x == 0 && y == 0 && z == 0
  balance = 0
  puts 'You lost your money'
end
```

Check second condition "If all variables are ones, we add $10 to our balance":

```
elsif x == 1 && y == 1 && z == 1
  balance += 10
  puts 'You won $10'
end
```

Add third condition "If all variables are twos, we add $20 to our balance" and also the last case "Otherwise we charge 50 cents for attempt". Here is the complete code for all conditions:

```
if x == 0 && y == 0 && z == 0
  balance = 0
  puts 'You lost your money'
elsif x == 1 && y == 1 && z == 1
  balance += 10
  puts 'You won $10'
elsif x == 2 && y == 2 && z == 2
  balance += 20
  puts 'You won $20'
else
  balance -= 0.5
  puts 'You lost 50 cents'
end
```

At the end show the final result:

```
  puts "Your balance is #{balance} dollars"
```

Complete program:

Slot Machine Program

```
print "What's your age: "
age = gets.to_i
if age < 18
  puts 'Sorry, but you should be at least 18 to play'
  exit
end

balance = 20
loop do
  puts 'Press Enter to pull the handle...'
  gets

  x = rand(0..5)
  y = rand(0..5)
  z = rand(0..5)

  puts "Result: #{x} #{y} #{z}"

  if x == 0 && y == 0 && z == 0
    balance = 0
    puts 'You lost your money'
  elsif x == 1 && y == 1 && z == 1
    balance += 10
    puts 'You won $10'
  elsif x == 2 && y == 2 && z == 2
```

```
    balance += 20
    puts 'You won $20'
  else
    balance -= 0.5
    puts 'You lost 50 cents'
  end

  puts "Your balance is #{balance} dollars"
end
```

Result:

```
What's your age: 20
Press Enter to pull the handle...

Result: 5 5 3
You lost 50 cents
Your balance is 19.5 dollars
Press Enter to pull the handle...

...

Result: 1 1 1
You won $10
Your balance is 29.5 dollars
Press Enter to pull the handle...
```

You must agree that there is nothing complex here and program works. With knowledge we gained in last chapters we can build simple text games, do some basic calculations, make other useful applications. Ruby programs are small, elegant, and very easy to read and understand. And this is exactly what makes programming fun!

In next chapters we'll cover essential data structures, will find out what classes and objects are, will get familiar with some tools, we'll have foundation for absolutely amazing things you can do with Ruby programming!

Exercise 1

Define method that will return random number. Make sure program works with this method. After that apply animation[8] to variables `x`, `y`, and `z`.

Exercise 2

Add more conditions to the game, like 333, 444, and so on. Use your imagination.

. . .

The full version of the book is available at http://rubyisforfun.com[9]

. . .

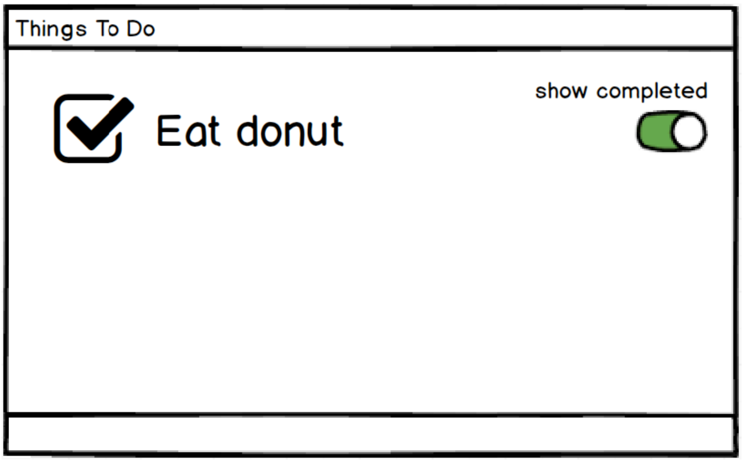
JSON-structure of a real-world application

JSON-structure is a universal way of describing almost any data. For example, the following hash describes the *state* of UI[10] of minimalistic TODO-list application:

[8] https://goo.gl/hpk49x
[9] http://rubyisforfun.com
[10] User Interface

```
{
  todos: [{
    text: 'Eat donut',
    completed: true
  }, {
    text: 'Go to gym',
    completed: false
  }],
  visibility_fiter: :show_completed
}
```

UI part of the app may look like this:



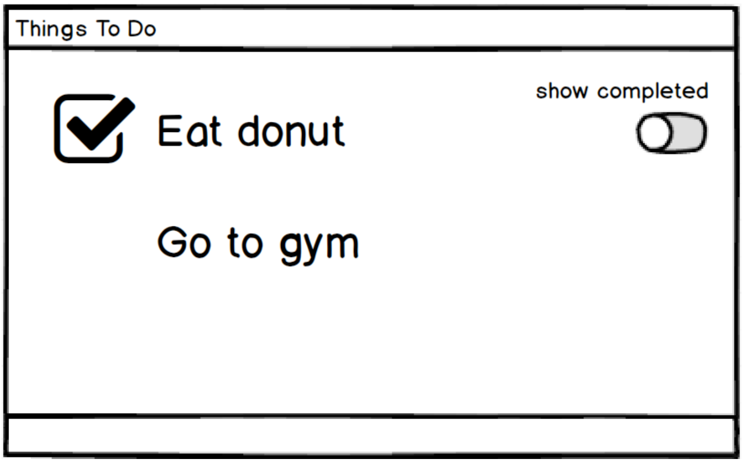
To-Do app, first item is visible, "Show completed" switch is on

If we break down this data structure, we'll see that there is no magic:

```
{
  todos: [ { ... }, { ... }, ... ],
  visibility_fiter: :show_completed
}
```

Value by the key of `:todos` is *Array*. Every element of the array is hash (object), which has two properties: 1) text 2) completion flag (*Boolean* type which is either `true` or `false`). Root level hash has also `:visibility_filter` key with corresponding value of `:show_completed`. It's up to developer to pick the right naming for hash keys or values. We just assume that somewhere some part of our program will be responsible for filtering out items based on the value of the filter. That's why we have two items in todos-array, but show only one on the mockup above.

When we turn off the switch, the UI will look different:



To-Do app, all items are visible, "Show completed" switch is off

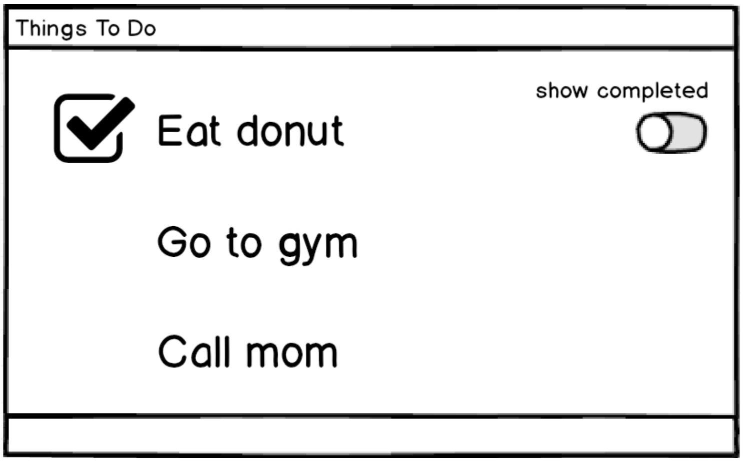
And the state of the program will be represented by slightly different hash:

```
{
  todos: [{
    text: 'Eat donut',
    completed: true
  }, {
    text: 'Go to gym',
    completed: false
  }],
  visibility_fiter: :show_all
}
```

When we want to add an item ("Call mom" below), we just push this element to "`todos`" array:

```
{
  todos: [{
    text: 'Eat donut',
    completed: true
  }, {
    text: 'Go to gym',
    completed: false
  }, {
    text: 'Call mom',
    completed: false
  }],
  visibility_fiter: :show_all
}
```

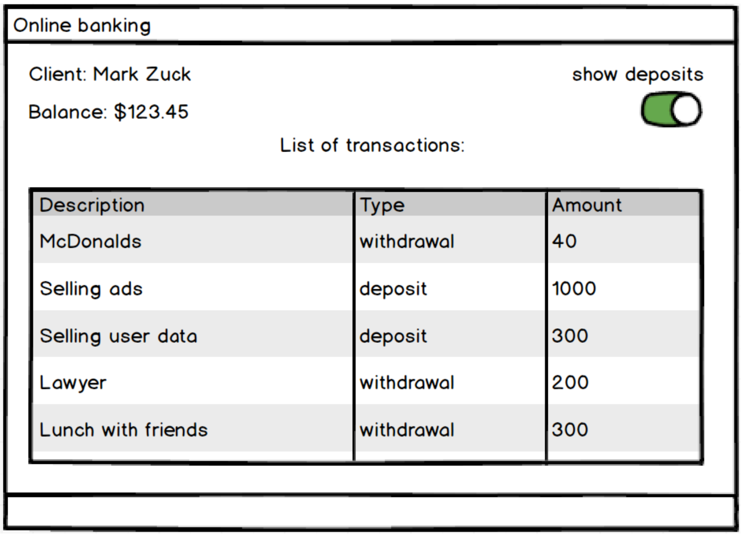
So the last item gets reflected in UI:



To-Do app with one extra item

Exercise 1

Create a JSON structure to represent the state of the following online-banking account:



Client: Mark Zuck

Balance: $123.45

List of transactions:

Description	Type	Amount
McDonalds	withdrawal	40
Selling ads	deposit	1000
Selling user data	deposit	300
Lawyer	withdrawal	200
Lunch with friends	withdrawal	300

UI of an online banking account

Exercise 2

Write a program that accepts hash you created in Exercise 1 and prints the result to the screen. Make sure that switch above works and the program doesn't show any deposits if the switch is off.