1. Środowisko

Tworzymy katalog do zabawy

```
$ mkdir uczelnia
$ cd uczelnia
```

2. Podstawy GIT

Tworzymy repozytorium lokalne

```
$ git init
$ git status
$ git log
```

Tworzymy plik i zapisujemy go

```
$ touch test.txt

$ git status
$ git add test.txt       # git add . => żeby dodać wszystko co zmieniliśmy
$ git status             # teraz pokaże nam to co dodaliśmy
```

Commitujemy nasze zmiany do lokalnego repozytorium

```
$ git add .
$ git commit -m "commit message"  # message jest bardzo ważny

$ git status                      # nic nie pokaże
$ git log                         # pokaże nasz pierwszy commit
```

Message powinien opisywać co zmieniliśmy (w kilku słowach).
Często, zwłaszcza do większych lub ważniejszych commitów, warto dodać jeszcze opis. W opisie skupiamy się nad tym dlaczego wprowadziliśmy dane zmiany, tłumaczymy zawiłości. Możemy też dorzucić linki do źródeł, które były pomocne przy podejmowaniu decyzji oraz do miejsc, w których opisany jest rozwiązywany przez nas problem (program do zarządzania projektem).

Remote (zdalny serwer)

```
$ git remote add <name> <url>  # dodajemy zdalny serwer
$ git remote -v                # sprawdzamy jakie mamy serwery
$ git remote remove <name>     # usuwamy zdalny serwer
```

GitHub

```
$ git clone <url> [folder_name]  # opcjonalnie możemy dodać nazwę katalogu

$ git remote -v  # teraz zobaczymy zdalny serwer
```

Praca ze zdalnym repozytorium

```
$ git pull origin master
$ touch README.md
```

Dodajmy jakąś informację pliku `README.md`

```
$ git add .
$ git commit -m "readme added"
$ git push origin master
```

3. Ruby

Otwieramy irb:

```
$ irb
```

Podstawowe typy i zmienne

```
> 1234
 => 1234

> 1234.class
 => Integer
```

```
> 2.15.class
 => Float

> variable = 10
 => 10

> variable
 => 10

> _
 => 10

> second_variable = _
 => 10
```

Null

```
> nil
 => nil

> nil.class
 => NilClass
```

Wyrażenia logiczne

Mamy dwie możliwości zapisywania wyrażeń AND i OR

```
> false and true
 => false

> false && true
 => false

> false or true
 => true

> false || true
 => true
```

Jednak trzeba uważać bo druga forma ma pierwszeństwo, czyli jest ewaluowana jako pierwsza:

```
> false and false or true
 => true
```

```
>false and false || true
 => false
```

Co więcej na liście pierwszeństwa operatorów, operator przypisania `=` jest pomiędzy `&& ||`, a `and or`. Aby to zobrazować spójrzmy na poniższy przykład:

```
> variable = true and false   # przypisze true do variable i wykona dalej
 => false
> variable
 => true

> variable = true && false    # przypisze do variable wynik operacji &&
 => false
> variable
 => false
```

Stringi i symbole

```
> 'some characters'
 => "some characters"

> 'some characters'.class
 => String

> 'some chars'.length
 => 10
> 'some chars'.reverse
 => "srahc emos"
> 'some chars'.upcase
 => "SOME CHARS"
```

Konkatenacja i interpolacja

```
> 'two ' + 'parts'
 => "two parts"

> lines_count = 10
 => 10
> "This file has #{lines_count} lines."
 => "This file has 10 lines."
```

Symbol

```
> :counter
 => :counter

> :counter.class
 => Symbol

> "char".object_id
 => 70099361336540
> "char".object_id
 => 70099353185720
> :char.object_id
 => 1276508
> :char.object_id
 => 1276508
```

Tablice i Hashe

Tablica są kolekjcją różnych obiektów, indeksowanych kolejnymi liczbami całkowitymi (zaczynając od 0). Hashe mogą być indeksowane także stringami lub symbolami.

```
> Array.new
 => []

> arr = [1, 5, "cat", :counter]
 => [1, 5, "cat", :counter]

> arr[0]
 => 1
> arr[2]
 => "cat"

> arr.push("last element")
 => [1, 5, "cat", :counter, "last element"]
> arr.pop
 => "last element"
> arr
 => [1, 5, "cat", :counter]
> arr << "added once more"
 => [1, 5, "cat", :counter, "added once more"]
```

```
> {}
 => {}
> Hash.new
 => {}

> age_of_people = { bob: 20, jane: 18 }
 => { :bob=>20, :jane=>18 }
```

```
> age_of_people[:jane]
 => 18

> age_of_people[:mark] = 30
 => 30

> age_of_people
 => {:bob=>20, :jane=>18, :mark=>30}

> age_of_people.keys
 => [:bob, :jane, :mark]> age_of_people.values
 => [20, 18, 30]
```

Pytajniki i wykrzykniki

Pomagają nam zrozumieć kod.

Metoda zakończona wykrzyknikiem oznacza potencjalne niebezpieczeństwo. Najczęściej oznacza to, że metoda może zwracać wyjątki, albo zmienia obiekt, na którym jest wywoływana.

Metoda zakończona znakiem zapytania zwraca wartośc boolean (true lub false).

```
> arr = [1, 10, 7, 3]
 => [1, 10, 7, 3]

> arr.sort
 => [1, 3, 7, 10]     # sortuje, ale nie zmienia obiektu
> arr
 => [1, 10, 7, 3]

> arr.sort!
  => [1, 3, 7, 10]     # sortuje i zmienia obiekt
> arr
 => [1, 3, 7, 10]

> arr.empty?
 => false
```

```
> age_of_people = { bob: 20, jane: 18 }
 => {:bob=>20, :jane=>18}

> age_of_people.has_key?(:bob)
 => true

> age_of_people.has_value?(19)
 => false
```

Bloki

Bloki są dłuższymi fragmentami kodu ujętymi słowami kluczowymi `begin` i `end`

```
> begin
?> a = 1 + 2
?> b = a + 3
?> end
 => 7         # zwracana jest ostatnia instrukcja w bloku
```

Skrypt w ruby

```
$ touch script.rb
```

```
# test.rb

begin
  a = 10
  b = 12
  puts a + b      # zwróć wynik do konsoli
end
```

```
$ ruby script.rb
```

Wyrażenia warunkowe

```
if 2 + 2 == 4
  puts 'math is working'
elsif  true
  puts 'should not go here'
else
  puts 'something is wrong'
end

# ternary operator
12 > 14 ? puts('is greater') : puts('is not')

a = some_array.empty? ? 'empty' : 'full'

if !false
```

```
  puts 'test2'
end

unless 1 > 4      # unless = if not
  puts 'test3'
end

# modyfikator
puts "2" if my_array.empty?
```

Pętle

Ruby oferuje standardowe pętle `for`, `while`, `loop`. Nie ma sensu ich omawiać. Zamiast tego przyjrzyjmy się kilku pętlom, które przydadzą nam się później.

```
# Metoda times wywołuje blok określoną liczbę razy
10.times do |index|
  puts index
end

# Pętle wykonywane na kolekcjach
collection = [1, :two, 'three']

for element in collection
  puts element
end

for number in 1..10
  puts number
end
```

Metoda `each` iteruje po kolekcji, wykonuje dany blok kodu dla każdego elementu i zwraca pierwotną kolekcję.

```
collection.each do |member|
  puts member
end
```

Metoda `map` iteruje po kolekcji, wykonuje dany blok kodu dla każdego elementu i zwraca tablicę wyników operacji.

```
collection = [1, 2, 3]

collection.map do |member|
  member^2
```

```ruby
end

collection.map { |member| member^2 }
```

Klasy i atrybuty

```ruby
class Animal
  attr_accessor :name, :age
end

animal = Animal.new
puts animal.name
animal.name = 'Reksio'
puts animal.name
puts animal
```

Konstruktor i metody

```ruby
class Animal
  attr_accessor :name, :age

  def initialize(name, age)
    @name = name
    @age = age
  end
end

animal1 = Animal.new('Reksio', 6)
animal2 = Animal.new('Turing', 3)

puts animal1.name
puts animal2.age
```

```ruby
class Animal
  attr_accessor :name, :age

  def give_sound
    puts 'default sound'
  end

  def introduce
    puts @name
  end
end

animal = Animal.new
animal.give_sound
```

```
animal.name = 'Reksio'
animal.introduce
puts animal.name
```

Dziedziczenie

```
class Animal
  attr_accessor :name, :age

  def initialize(name, age)
    @name = name
    @age = age
  end

  def sound
    puts 'default sound'
  end

  def age_in_years(years)
    puts age + years
  end
end

class Snake < Animal
  def sound
    puts 'sssssssssssss!'
  end
end

class Cat < Animal
  def sound
    puts 'miau!'
  end
end
```

```
animal = Animal.new('Undefined', 99)
cat = Cat.new('Mruczek', 3)
snake = Snake.new('Henry', 6)

animal.sound
cat.sound
snake.sound

animal.age_in_years(5)
cat.age_in_years(5)
snake.age_in_years(5)
```

Kompozycja

```ruby
module LivesInSea
  def swim
    puts 'I am swimming'
  end
end

module Mammal
  def have_lungs
    puts 'I have lungs!'
  end
end

class Whale
  include LivesInSea
  include Mammal
end
```

```ruby
animal = Whale.new
animal.swim
animal.have_lungs
```

4. RVM / Gemy

Komendy dotyczące ruby:

```
$ rvm list            # listujemy zainstalowane wersje ruby
$ rvm use 2.3.0       # aktywujemy daną wersję ruby
$ rvm install 2.4.0   # instalujemy nową wersję ruby
$ rvm remove 2.4.0    # usuwamy daną wersję ruby
```

Komendy dotyczące gemsetów:

```
$ rvm gemset list               # listujemy gemsety dostępne dla danego ruby
$ rvm gemset create new_gemset  # tworzymy nowy gemset
$ rvm gemset use new_gemset     # aktywujemy dany gemset
$ rvm gemset delete new_gemset  # usuwamy dany gemset
```

Gdy chcemy przełączyć się na inną wersję ruby z konkretnym gemsetem (istniejącym lub nowym) możemy użyć komendy skróconej:

```
$ rvm use --create ruby_version@new_gemset
```

Automatyczna kontrola RVM

W katalogu projektowym tworzymy dwa pliki, w których umieszczamy odpowiednio numer wersji ruby i nazwę gemsetu.

```
$ echo "ruby-2.5.1" > .ruby-version
$ echo "uczelnia" > .ruby-gemset
```

Po wejściu w nasz katalog projektowy RVM automatycznie wczyta odpowiednią wersję ruby z danym gemsetem. Dzięki temu nie musimy pamiętać o zmienianiu wersji.

Instalowanie gemów

Gemy instalujemy bardzo prosto:

```
$ gem install bundler   # instalujemy gem w danym gemsecie!!!
```

Gemset `global` jest szczególny. Wszystko co tam zainstalujemy będzie dostępne także w innych gemsetach.