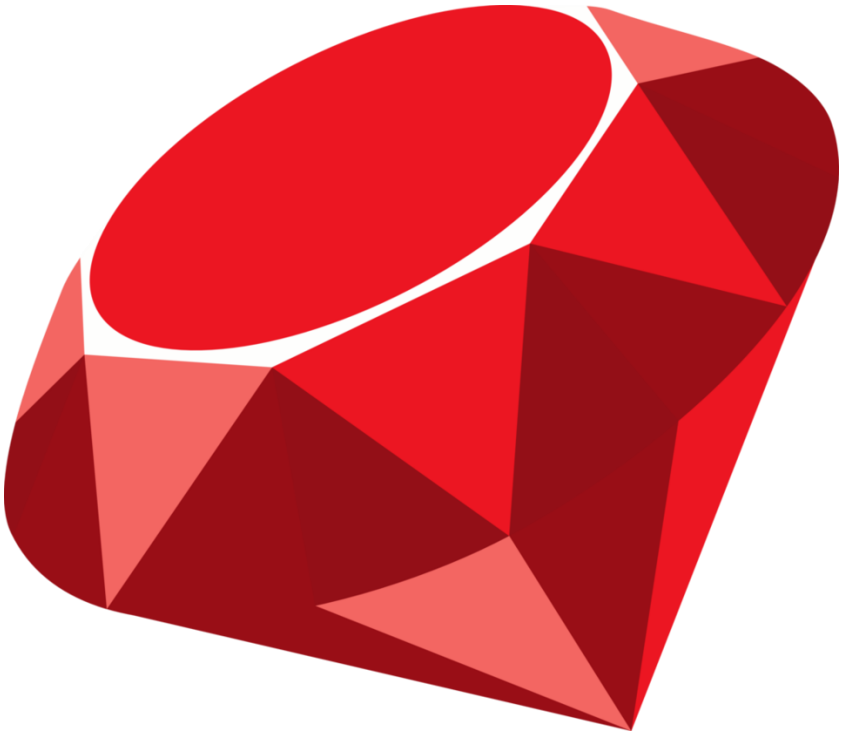


Ruby

به زبان ساده

تقدیم به

همه جویندگان علم

راه‌های ارتباط با نویسنده

وب سایت: www.w3-farsi.com

لینک تلگرام: https://telegram.me/ebrahimi_younes

ID تلگرام: @ebrahimi_younes

پست الکترونیکی: younes.ebrahimi.1391@gmail.com

Ruby چیست

Ruby (روبی)، یک زبان برنامه‌نویسی انعطاف‌پذیر، وب، تست نفوذ، پویا و شیءگرا است. Ruby ویژگی‌های نگارشی Perl و شی گرایی Smarttalk را با هم در خود دارد. زبان روبی در سال‌های میانی دهه ۱۹۹۰ توسط یوکیهیرو ماتسوموتو در ژاپن اختراع شد.

زبان روبی رسماً در روز ۲۴ فوریه ۱۹۹۳ توسط یوکیهیرو ماتسوموتو معرفی شد. او دنبال ساخت زبانی بود که امکانات متعادلی برای برنامه‌نویسی تابعی و برنامه‌نویسی دستوری برای برنامه‌نویس فراهم آورد. ماتسوموتو دربارهٔ انگیزه‌اش برای ساخت روبی می‌گوید: در جستجوی زبانی بودم که از پایتون شیءگراتر و از پرل قدرتمندتر باشد. برای همین تصمیم گرفتم خودم آن را بسازم.

یوکیهیرو ماتسوموتو و همکارش دو نام «روبی» و «کورال» را برای این زبان جدید برگزیده بودند. از آنجاییکه نام کورال پیش از آن برای یکی از زبان‌های برنامه‌نویسی بریتانیایی انتخاب شده بود، نام «روبی» به عنوان نام نهایی برگزیده شد. ماتسوموتو گفته که یکی از دلایل انتخاب نام «روبی» این بود که یاقوت (Ruby) نشان ماه تولد یکی از همکاران وی بوده‌است.

نخستین ویرایش Ruby با عنوان Ruby 0.95 در ۲۱ دسامبر ۱۹۹۵ میلادی روی یکی از شبکه‌های تخصصی اینترنتی در ژاپن منتشر شد. پس از آن، سه ویرایش دیگر Ruby در ظرف دو روز انتشار یافتند. در همین دوره نخستین لیست پست الکترونیک برای روبی در ژاپن براه افتاد.

نخستین نسخه اصلی روبی با عنوان Ruby 1.0 در ۲۵ دسامبر ۱۹۹۶ منتشر شد. پس از انتشار Ruby 1.3 در سال ۱۹۹۹، نخستین لیست پست الکترونیک به زبان انگلیسی آغاز بکار کرد. در سپتامبر سال ۲۰۰۰ نخستین کتاب راهنمای برنامه‌نویسی به زبان روبی به انگلیسی به چاپ رسید که به افزایش محبوبیت این زبان در کشورهای مختلف کمک کرد. در جدول زیر لیست نسخه‌های مختلف این زبان آمده است:

نسخه	آخرین نسخه کوچک	تاریخ انتشار اولیه	پایان مرحله پشتیبانی	پایان فاز نگهداری امنیتی
1.0	نامعلوم	1996-12-25	نامعلوم	نامعلوم
1.8	1.8.7-p375	2003-08-04	2012-06	2014-07-01
1.9	1.9.3-p551	2007-12-25	2014-02-23	2015-02-23
2.0	2.0.0-p648	2013-02-24	2015-02-24	2016-02-24
2.1	2.1.10	2013-12-25	2016-03-30	2017-03-31
2.2	2.2.10	2014-12-25	2017-03-28	2018-03-31
2.3	2.3.8	2015-12-25	2018-06-20	2019-03-31
2.4	2.4.10	2016-12-25	2019-04-01	2020-04-01
2.5	2.5.8	2017-12-25	نامعلوم	نامعلوم
2.6	2.6.6	2018-12-25	نامعلوم	نامعلوم

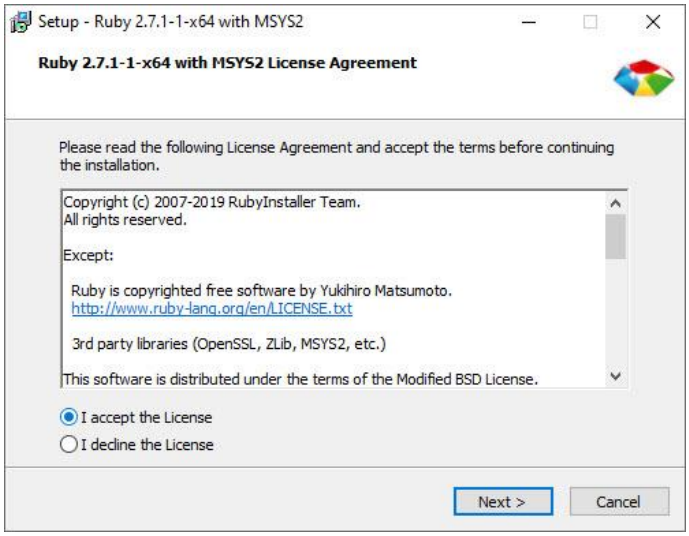
2.7	2.7.1	2019-12-25	نامعلوم	نامعلوم
3.0		2020	نامعلوم	نامعلوم

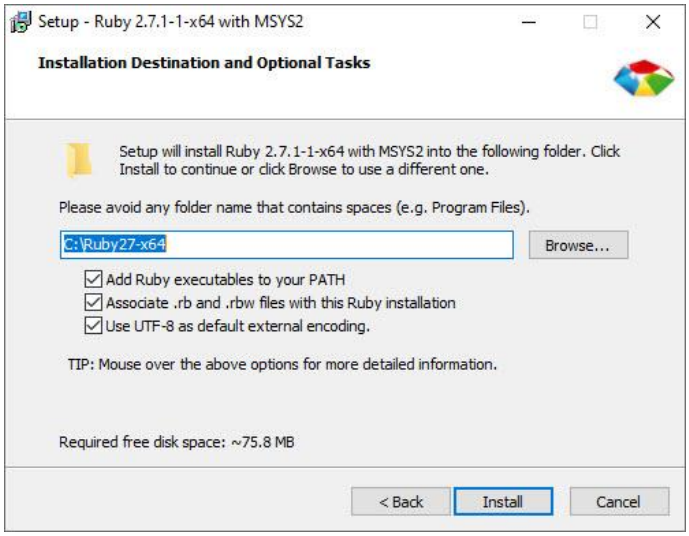
نصب و راه اندازی Ruby

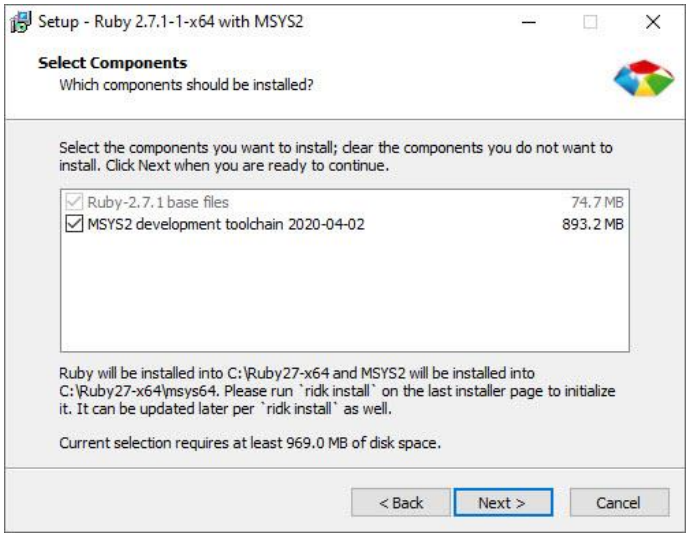
برای کدنویسی به زبان Ruby و همچنین اجرای کدهای این زبان به دو ابزار احتیاج دارید. یکی از آنها، یک ویرایشگر متن ساده مانند Ruby خود ویندوز و دیگری مفسر زبان Ruby می‌باشد. این کامپایلر را می‌توانید از لینک زیر دانلود کنید:

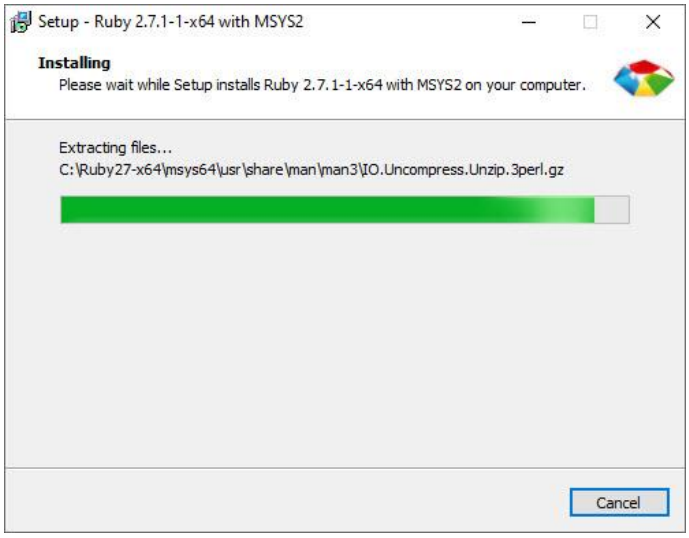
```
http://dl.w3-farsi.com/Software/Ruby/rubyinstaller-devkit-2.7.1-1-x64.exe
```

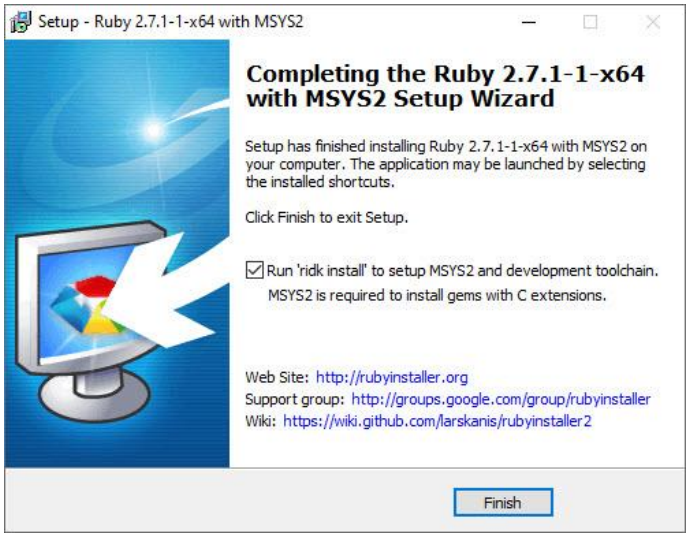
بعد از دانلود فایل بالا، با دوبار کلیک بر روی آن و زدن چند دکمه Next یا Yes، که مراحل آن در زیر نمایش داده شده است، مفسر Ruby نصب می‌شود:






Setup - Ruby 2.7.1-1-x64 with MSYS2
Select Components
Which components should be installed?
Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.
Ruby-2.7.1 base files 74.7 MB
MSYS2 development toolchain 2020-04-02 893.2 MB
Ruby will be installed into C:\Ruby27-x64 and MSYS2 will be installed into C:\Ruby27-x64\msys64. Please run `ridk install` on the last installer page to initialize it. It can be updated later per `ridk install` as well.
Current selection requires at least 969.0 MB of disk space.
< Back
Next >
Cancel


Setup - Ruby 2.7.1-1-x64 with MSYS2
Installing
Please wait while Setup installs Ruby 2.7.1-1-x64 with MSYS2 on your computer.
Extracting files...
C:\Ruby27-x64\msys64\usr\share\man\man3\IO.Uncompress.Unzip.3perl.gz
Cancel


Setup - Ruby 2.7.1-1-x64 with MSYS2
Completing the Ruby 2.7.1-1-x64 with MSYS2 Setup Wizard
Setup has finished installing Ruby 2.7.1-1-x64 with MSYS2 on your computer. The application may be launched by selecting the installed shortcuts.
Click Finish to exit Setup.
Run 'ridk install' to setup MSYS2 and development toolchain.
MSYS2 is required to install gems with C extensions.
Web Site: http://rubyinstaller.org
Support group: http://groups.google.com/group/rubyinstaller
Wiki: https://wiki.github.com/larskanis/rubyinstaller2
Finish

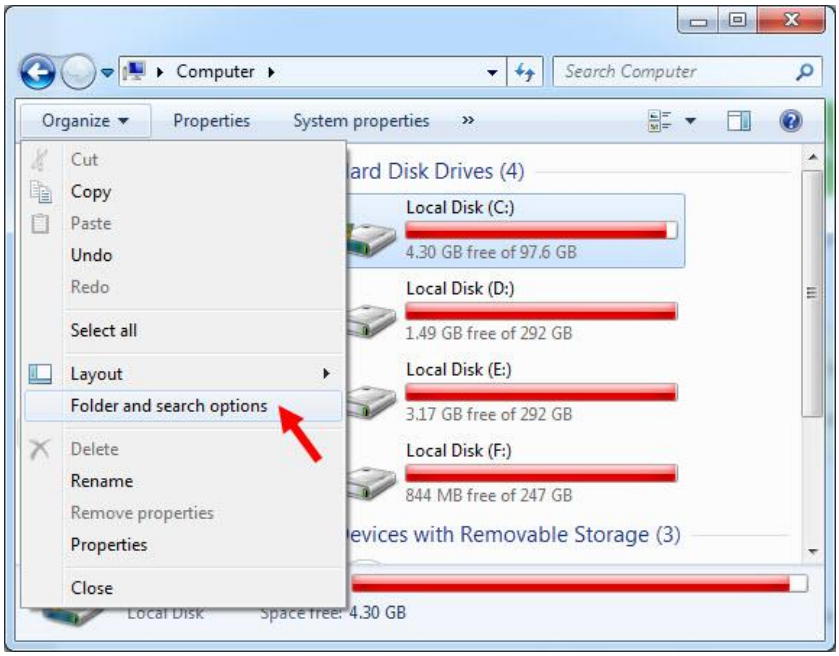
بعد از طی مراحل بالا، شما به راحتی می‌توانید، کدنویسی خود را شروع کنید. در درس بعد شما را با نحوه اجرای کدهای زبان Ruby آشنا می‌کنم.

ساخت یک برنامه ساده با Ruby

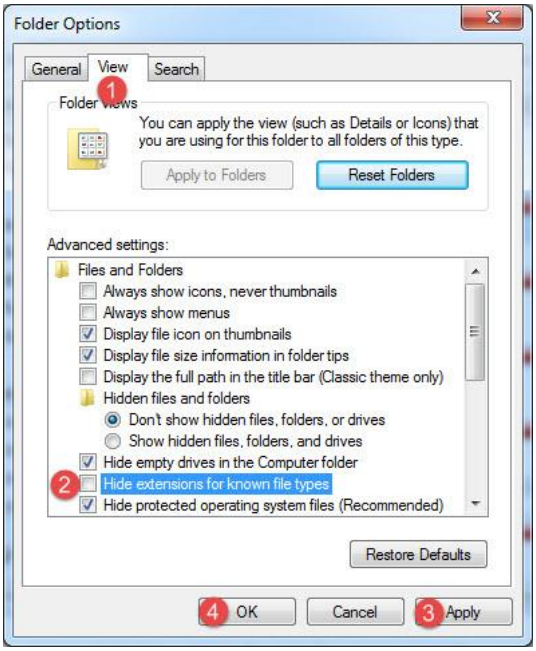
اجازه بدهید یک برنامه بسیار ساده به زبان Ruby بنویسیم. این برنامه یک پیغام را نمایش می‌دهد. در این درس می‌خواهم ساختار و دستور زبان یک برنامه ساده Ruby را توضیح دهم. قبل از ایجاد برنامه به این نکته خیلی مهم توجه کنید:

> در نوشتن این برنامه و برنامه‌های آتی، به حروف بزرگ و کوچک، توجه کنید. چون Ruby به بزرگ و کوچک بودن حروف حساس است.

یکی از تنظیماتی که قبل از شروع این درس توصیه می‌کنیم که اعمال کنید این است که پسوند فایل‌ها داخل ویندوز را قابل مشاهده کنید. برای این کار به My Computer رفته و به صورت زیر از منوی Organize گزینه Folder and search options را بزنید:



از پنجره باز شده به صورت زیر به سربرگ View رفته و تیک کنار گزینه Hide Extension for khnown file types را بردارید:



در ادامه شما را نحوه ایجاد اولین برنامه در Ruby را توضیح می‌دهیم. همانطور که گفته شد، شما برای کامپایل و اجرای برنامه‌های Ruby به مفسر این زبان نیاز دارید، که آن را در درس قبل نصب کردیم و الان فرض می‌کنیم که شما هیچ IDE یا محیط کدنویسی در اختیار ندارید و می‌خواهید یک برنامه Ruby بنویسید. در این برنامه می‌خواهیم پیغام `Welcome to Ruby Tutorials` چاپ شود.

استفاده از CMD یا Command Prompt

در این روش، شما به منوی Start ویندوز رفته و cmd را اجرا می کنید. بعد از اجرای cmd دستور `ruby -h` را بنویسید:

```
Microsoft Windows [Version 10.0.18363.959]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Siavash>ruby -h
Usage: ruby [switches] [--] [programfile] [arguments]
  -0[octal]       specify record separator (\0, if no argument)
  -a              autosplit mode with -n or -p (splits $_ into $F)
  -c              check syntax only
  -Cdirectory     cd to directory before executing your script
  -d              set debugging flags (set $DEBUG to true)
  -e 'command'    one line of script. Several -e's allowed. Omit [programfile]
  -Eex[:in]       specify the default external and internal character encodings
  -Fpattern       split() pattern for autosplit (-a)
  -i[extension]   edit ARGV files in place (make backup if extension supplied)
  -Idirectory     specify $LOAD_PATH directory (may be used more than once)
  -l              enable line ending processing
  -n              assume 'while gets(); ... end' loop around your script
  -p              assume loop like -n but print line also like sed
  -rlibrary       require the library before executing your script
  -s              enable some switch parsing for switches after script name
  -S              look for the script using PATH environment variable
  -v              print the version number, then turn on verbose mode
  -w              turn warnings on for your script
  -W[level=2|:category]     set warning level; 0=silence, 1=medium, 2=verbose
  -x[directory]   strip off text before #!ruby line and perhaps cd to directory
```

```
  --jit           enable JIT with default options (experimental)
  --jit-[option]  enable JIT with an option (experimental)
  -h              show this message, --help for more info
```

دستور ruby -h، لیست تمام دستوراتی که می توانیم از آنها برای تعامل با مفسر Ruby استفاده کنیم را در اختیار ما می گذارد. مثلا در بین دستورات بالا e- باعث اجرای یک دستور در cmd می شود. cmd را بسته و یکبار دیگر آنرا اجرا کرده و کدهای زیر را بنویسید:

```
Microsoft Windows [Version 10.0.18363.959]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Siavash>ruby -e 'puts "Welcome to Ruby tutorials!"'
Welcome to Ruby tutorials!

C:\Users\Siavash>
```

استفاده از محیط تعاملی یا Interactive

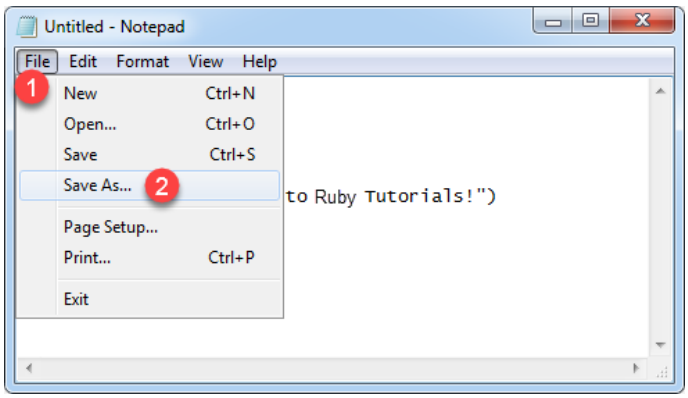
هنگام نصب مفسر Ruby در ویندوز یک محیط کدنویسی و تعاملی هم همراه آن نصب می شود که به شما اجازه کدنویسی با Ruby را می دهد. این محیط تعاملی در منوی Start ویندوز قرار دارد و نام Interactive Ruby می باشد. بر روی این نام در منوی Start کلیک کنید. این محیط همانند cmd مشکی می باشد. بعد از اجرا، دستورات زیر را در آن نوشته و دکمه Enter را بزنید:

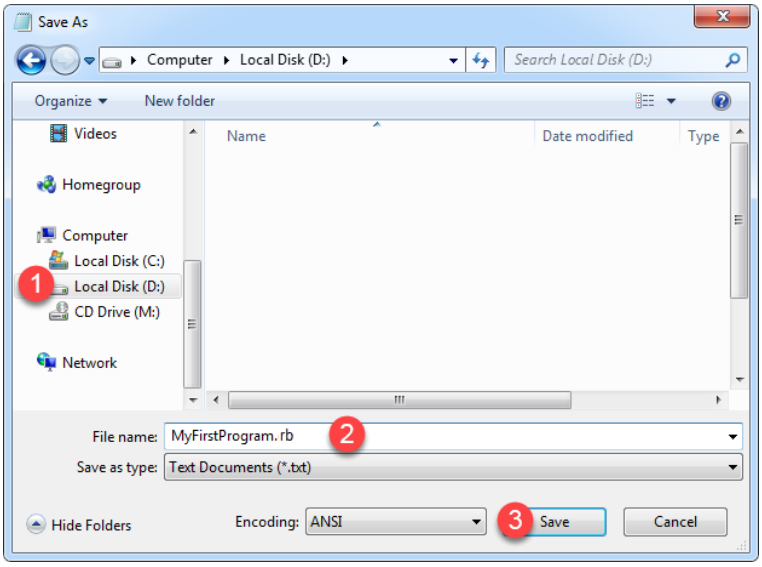
```
irb(main):001:0> puts "Welcome to Ruby Tutorials"
Welcome to Ruby Tutorials
=> nil
(reverse-i-search)
```

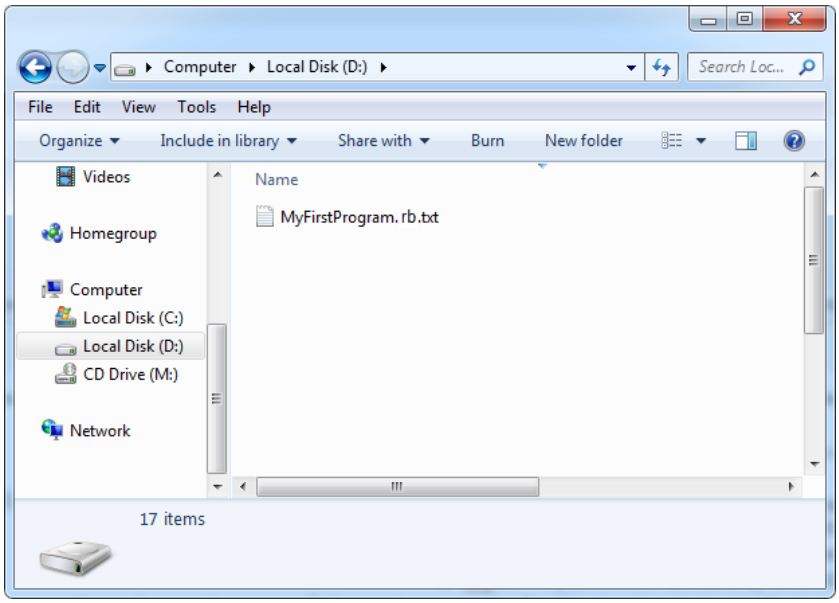
استفاده از ویرایشگر متن و محیط cmd

در این روش که پیشنهاد می دهیم در آموزش های بعدی هم از آن استفاده کنید، ابتدا یک ویرایشگر متن مانند Notepad را باز کرده و کدهای زیر را در داخل آن نوشته (حروف بزرگ و کوچک را رعایت کنید) و با پسوند rb ذخیره کنید :

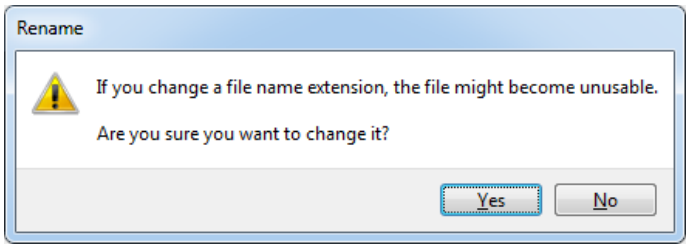
```
puts "Welcome to Ruby Tutorials!"
```



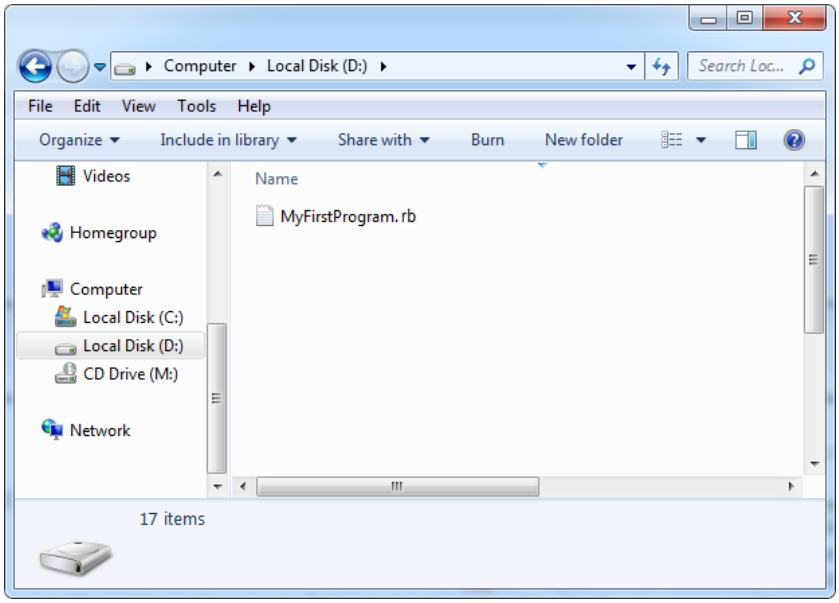




همانطور که مشاهده می‌کنید، بعد از ذخیره، فایل با پسوند `MyFirstProgram.rb.txt` ذخیره می‌شود که شما باید پسوند `.txt` آن را حذف کنید. هنگام پاک کردن پسوند، پیغامی به صورت زیر ظاهر می‌شود که شما باید بر روی گزینه `Yes` کلیک کنید:



تا شکل نهایی فایل به صورت زیر در آید:



حال نوبت به اجرای برنامه می‌رسد. فایل ما در درایو D قرار دارد. ابتدا cmd را باز کرده و کد زیر را در داخل آن نوشته و دکمه Enter را می‌زنید :

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\siavash>d:

D:\>ruby MyFirstProgram.rb
Welcome to Ruby Tutorials!

D:\>
```

همانطور که در کد بالا مشاهده می‌کنید برای اجرای کدهای Ruby ابتدا کلمه ruby و سپس نام پروژه به همراه پسوند آن را می‌نویسیم (مثل MyFirstProgram.rb).

مثال بالا ساده‌ترین برنامه‌ای است که شما می‌توانید در Ruby بنویسید. هدف در مثال بالا نمایش یک پیغام در صفحه نمایش است. هر زبان برنامه نویسی دارای قواعدی برای کدنویسی است. Ruby دارای توابع از پیش تعریف شده‌ای است که هر کدام برای مقاصد خاصی به کار می‌روند. هر چند که در آینده در مورد توابع بیشتر توضیح می‌دهیم، ولی در همین حد به توضیح تابع بسنده می‌کنیم که توابع مجموعه‌ای از کدها هستند که وقتی اسم آنها را صدا می زنیم، یک کار خاص را انجام می دهند. یکی از این توابع، تابع puts است. از تابع puts برای چاپ یک رشته استفاده می‌شود. یک رشته گروهی از کاراکترها است، که به وسیله دابل کوتیشن (") محصور شده است. مانند: "Welcome to Ruby Tutorials!" . یک کاراکتر می‌تواند یک حرف، عدد، علامت یا ... باشد. در کل مثال بالا نحوه استفاده از تابع puts است. توضیحات بیشتر در درس‌های آینده آمده است. Ruby فضاهای خالی را نادیده می‌گیرد. مثلاً از کد زیر اشکال نمی‌گیرد:

```
puts       "Welcome to Ruby Tutorials!"
```

همیشه به یاد داشته باشید که Ruby به بزرگی و کوچکی حروف حساس است. یعنی به طور مثال MAN و man در Ruby با هم فرق دارند. رشته‌ها و توضیحات از این قاعده مستثنی هستند که در درس‌های آینده توضیح خواهیم داد. مثلاً کدهای زیر با خطا مواجه می‌شوند و اجرا نمی‌شوند :

```
PUTs "Welcome to Ruby Tutorials!"
PuTs "Welcome to Ruby Tutorials!"
pUts "Welcome to Ruby Tutorials!"
```

تغییر در بزرگی و کوچکی حروف از اجرای کدها جلوگیری می‌کند. اما کد زیر کاملاً بدون خطا است :

```
puts "Welcome to Ruby Tutorials!"
```

در زبان Ruby نیازی به سمیکالن (;) ندارید. همین که بعد از هر خط یک بار Enter بزنید به منزله این است که آن خط به پایان رسیده است. مثلاً دو خط زیر دو دستور جدا هستند:

```
puts "Welcome to Ruby Tutorials!"
puts "Welcome to Ruby Tutorials!"
```

به جای استفاده از چند تابع puts برای چاپ چند دستور می توان از یک تابع puts برای این منظور استفاده کرد. به کد زیر توجه کنید:

```
puts "Welcome to Ruby Tutorials!" , "Welcome to Ruby Tutorials!" , "Welcome to Ruby Tutorials!"
```

```
Welcome to Ruby Tutorials!
Welcome to Ruby Tutorials!
Welcome to Ruby Tutorials!
```

در کد بالا ما سه بار رشته Welcome to Ruby Tutorials! را با استفاده از یک تابع puts چاپ کرده ایم. کافیست بین دستورات علامت کاما (,) قرار دهید. در Ruby یک تابع دیگر برای چاپ در خروجی وجود دارد که نام ان print است. تفاوت این تابع با تابع puts این است که تابع puts هر دستور را در یک خط جدا چاپ و تابع print همه دستورات را در یک خط و پشت سر هم چاپ می کند. برای درک بهتر به کد زیر توجه کنید:

```
puts "Hello "
puts "World!"
```

```
Hello
World!
```

مشاهده می کنید که دستورات در دو خط جدا از هم چاپ شدند. حال همین کد را با استفاده از تابع print می نویسیم :

```
print "Hello "
print "World!"
```

```
Hello World!
```

گاهی اوقات در برنامه نویسی می خواهیم که برخی از کد در ابتدا و برخی در انتهای برنامه اجرا شوند. برای این منظور می توانیم از دو دستور یا بلوک کد BEGIN و END به صورت زیر استفاده کنیم :

```
puts "This is a statement"

END {
  puts "End statement"
}
```

```
BEGIN {
  puts "BEGIN statement"
}
```

ممکن است که شما انتظار داشته باشید که کدها به ترتیب از بالا به پایین اجرا و خروجی به صورت زیر باشد:

```
This is a statement
END statement

BEGIN statement
```

ولی خروجی کد بالا به صورت زیر خواهد بود:

```
BEGIN statement
This is a statement
END statement
```

یعنی ابتدا دستورات داخل بلوک BEGIN، سپس `This is a statement` و در نهایت دستورات بلوک END اجرا می شوند.

توضیحات

وقتی که کدی تایپ می کنید شاید بخواهید که متنی جهت یادآوری وظیفه آن کد به آن اضافه کنید. در Ruby و بیشتر زبانهای برنامه نویسی می توان این کار را با استفاده از توضیحات انجام داد. توضیحات متونی هستند که توسط مفسر نادیده گرفته می شوند و به عنوان بخشی از کد محسوب نمی شوند.

هدف اصلی از ایجاد توضیحات، بالا بردن خوانایی و تشخیص نقش کدهای نوشته شده توسط شما، برای دیگران است. فرض کنید که می خواهید در مورد یک کد خاص، توضیح بدهید، می توانید توضیحات را در بالای کد یا کنار آن بنویسید. از توضیحات برای مستند سازی برنامه هم استفاده می شود. در برنامه زیر نقش توضیحات نشان داده شده است :

```
#This line will puts the message hello world
puts "Hello World!"
```

در کد بالا، خط اول کد بالا یک توضیح درباره خط دوم است که به کاربر اعلام می کند که وظیفه خط دوم چیست ؟ با اجرای کد بالا فقط جمله Hello World چاپ شده و خط اول در خروجی نمایش داده نمی شود چون مفسر توضیحات را نادیده می گیرد. همانطور که مشاهده می کنید برای درج توضیحات در Ruby از علامت # استفاده می شود. برای توضیحات طولانی هم باید در ابتدای هر خط از توضیح این علامت درج شود :

```
#This line will puts
#the message hello world
puts "Hello World!"
```

یکی روش دیگر برای ایجاد توضیحات چند خطی به صورت زیر است:

```
=begin
This is a multiline
Ruby comment example.
=end
puts "Hello World!"
```

همانطور که مشاهده می کنید، =begin را قبل و =end را بعد از توضیحات قرار داده ایم.

کاراکترهای کنترلی

کاراکترهای کنترلی، کاراکترهای ترکیبی هستند که با یک بک اسلش \ شروع می‌شوند و به دنبال آنها یک حرف یا عدد می‌آید و یک رشته را با فرمت خاص نمایش می‌دهند. برای مثال برای ایجاد یک خط جدید و قرار دادن رشته در آن می‌توان از کاراکتر کنترلی \n استفاده کرد :

```
puts "Hello\nWorld!"
```

```
Hello
World
```

مشاهده کردید که مفسر بعد از مواجهه با کاراکتر کنترلی \n نشانگر ماوس را به خط بعد برده و بقیه رشته را در خط بعد نمایش می‌دهد. جدول زیر لیست کاراکترهای کنترلی و کارکرد آنها را نشان می‌دهد :

عملکرد	کاراکتر کنترلی	عملکرد	کاراکتر کنترلی
Form Feed	\f	چاپ کوتیشن	\’
خط جدید	\n	چاپ دابل کوتیشن	\”
سر سطر رفتن	\r	چاپ بک اسلش	\\
حرکت به صورت افقی	\t	چاپ فضای خالی	\0
حرکت به صورت عمودی	\v	صدای بیپ	\a
چاپ کاراکتر یونیکد	\u	حرکت به عقب	\b

ما برای استفاده از کاراکترهای کنترلی، از بک اسلش \ استفاده می‌کنیم. از آنجاییکه \ معنای خاصی به رشته‌ها می‌دهد برای چاپ بک اسلش \ باید از \\ استفاده کنیم :

```
puts "We can puts a \\ by using the \\\\ escape sequence."
```

```
We can puts a \ by using the \\ escape sequence.
```

یکی از موارد استفاده از \\، نشان دادن مسیر یک فایل در ویندوز است :

```
puts "C:\\Program Files\\Some Directory\\SomeFile.txt"
```

```
C:\Program Files\Some Directory\SomeFile.txt
```

از آنجاییکه از دابل کوتیشن ” برای نشان دادن رشته‌ها استفاده می‌کنیم برای چاپ آن از \” استفاده می‌کنیم :

```
puts "I said, \"Motivate yourself!\"."
```

```
I said, "Motivate yourself!".
```

همچنین برای چاپ کوتیشن ' از \' استفاده می‌کنیم :

```
puts "The programmer\'s heaven."
```

```
The programmer's heaven.
```

برای ایجاد فاصله بین حروف یا کلمات از \t استفاده می‌شود :

```
puts "Left\tRight"
```

```
Left   Right
```

برای چاپ کاراکترهای یونیکد می‌توان از \u استفاده کرد. برای استفاده از \u، مقدار در مبنای ۱۶ کاراکتر را درست بعد از علامت \u قرار می‌دهیم.

برای مثال اگر بخواهیم علامت کپی رایت © را چاپ کنیم، باید بعد از علامت \u مقدار A9۰۰ را قرار دهیم مانند :

```
puts "\u00A9"
```

```
©
```

برای مشاهده لیست مقادیر مبنای ۱۶ برای کاراکترهای یونیکد به لینک زیر مراجعه نمایید :

http://www.ascii.cl/htmlcodes.htm

اگر مفسر به یک کاراکتر کنترلی غیر مجاز برخورد کند، برنامه پیغام خطا می‌دهد. بیشترین خطا زمانی اتفاق می افتد که برنامه نویس برای چاپ اسلش \ از \\ استفاده می‌کند.

متغیر

متغیر مکانی از حافظه است که شما می‌توانید مقادیری را در آن ذخیره کنید. می‌توان آن را به عنوان یک ظرف تصور کرد که داده‌های خود را در آن قرار داده‌اید. محتویات این ظرف می‌تواند پاک شود یا تغییر کند. هر متغیر دارای یک نام نیز هست. که از طریق آن می‌توان متغیر را از دیگر متغیرها تشخیص داد و به مقدار آن دسترسی پیدا کرد. همچنین دارای یک مقدار می‌باشد که می‌تواند توسط کاربر انتخاب شده باشد یا نتیجه یک محاسبه باشد. مقدار متغیر می‌تواند تهی نیز باشد. متغیر دارای نوع نیز هست بدین معنی که نوع آن با نوع داده‌ای که در آن ذخیره می‌شود یکی است.

متغیر دارای عمر نیز هست که از روی آن می‌توان تشخیص داد که متغیر باید چقدر در طول برنامه مورد استفاده قرار گیرد. و در نهایت متغیر دارای محدوده استفاده نیز هست که به شما می‌گوید که متغیر در چه جای برنامه برای شما قابل دسترسی است. ما از متغیرها به عنوان یک انبار موقتی برای ذخیره داده استفاده می‌کنیم. هنگامی که یک برنامه ایجاد می‌کنیم احتیاج به یک مکان برای ذخیره داده، مقادیر یا داده‌هایی که توسط کاربر وارد می‌شوند، داریم. این مکان، همان متغیر است.

برای این از کلمه متغیر استفاده می‌شود چون ما می‌توانیم بسته به نوع شرایط هر جا که لازم باشد، مقدار آن را تغییر دهیم. متغیرها موقتی هستند و فقط موقعی مورد استفاده قرار می‌گیرند که برنامه در حال اجراست و وقتی شما برنامه را می‌بندید محتویات متغیرها نیز پاک می‌شود. قبلاً ذکر شد که به وسیله نام متغیر می‌توان به آن دسترسی پیدا کرد. برای نامگذاری متغیرها باید قوانین زیر را رعایت کرد :

- نام متغیر باید با یکی از حروف کوچک الفبا (a-z) یا علامت _ شروع شود.
- نمی‌تواند شامل کاراکترهای غیرمجاز مانند . ,$,^ ,? ,# باشد.
- نمی‌توان از کلمات رزرو شده در Ruby برای نام متغیر استفاده کرد.
- نام متغیر نباید دارای فضای خالی (spaces) باشد.

اسامی متغیرها نسبت به بزرگی و کوچکی حروف حساس هستند. در Ruby دو حرف مانند a و A دو کاراکتر مختلف به حساب می‌آیند. دو متغیر با نامهای myNumber و MyNumber دو متغیر مختلف محسوب می‌شوند چون یکی از آنها با حرف کوچک m و دیگری با حرف بزرگ M شروع می‌شود. متغیر دارای نوع هست که نوع داده‌ای را که در خود ذخیره می‌کند را نشان می‌دهد. در درس بعد در مورد انواع داده ها در Ruby توضیح می دهیم. لیست کلمات کلیدی Ruby، که نباید از آنها در نامگذاری متغیرها استفاده کرد در زیر آمده است:

```
BEGIN   END     alias   and     begin   break   case      class     def      __ENCODING__
module  next    nil     not     or      redo    rescue    retry     return   __LINE__
elsif   end     false   ensure  for     if      true      undef     unless   __FILE__
do      else    super   then    until   when    while     defined?  self     yield
```

انواع داده

انواع داده هایی که در Ruby وجود دارند عبارتند از :

داده	توضیح	
عددی (Numeric)	integer	شامل اعداد مثبت و منفی صحیح می باشد.
	float	شامل اعداد اعشاری می باشد.
	complex	شامل اعداد مختلط می باشد. نوع داده مختلط نوع غیر قابل تغییری است که یک جفت float را نگهداری می کند که یک بخش آن نشان دهنده قسمت حقیقی و یک بخش آن نشان دهنده قسمت موهومی عدد مختلط است. مانند (7.1j + 3)
رشته ای (String)	به مجموعه ای از کاراکترها که بین دو علامت کوتیشن یا دابل کوتیشن قرار گرفته باشند، اطلاق می شود.	
آرایه (List)	مجموعه ای از آیتم ها هستند که بین دو علامت [] قرار گرفته و با علامت کاما (,) از هم جدا شده اند.	
هش (Hash)	مجموعه ای از آیتم ها هستند که به صورت کلید و مقدار بوده، بین دو علامت {} قرار گرفته، و با علامت کاما (,) از هم جدا شده اند.	
سمبل (Symbol)	همانند رشته ها هستند و با علامت دو نقطه (:) شروع می شوند و در Hash ها کاربرد دارند.	
boolean	شامل دو مقدار true یا false می باشد.	

در مورد انواع داده های بالا و نحوه استفاده از آنها در متغیرها، در درس بعد توضیح می دهیم.

استفاده از متغیرها

بر خلاف زبان هایی مثل جاوا و سی شارپ، که هنگام تعریف متغیر باید نوع متغیر را هم مشخص می کردیم، در Ruby کافیست که فقط نام متغیر را نوشته و به وسیله علامت مساوی یک مقدار به آن اختصاص دهیم :

```
variableName = Value
```

در مثال زیر نحوه تعریف و مقداردهی متغیرها نمایش داده شده است :

```ruby
intVar     = 10
floatVar   = 12.5
boolVar    = true
stringVar  = 'Hello World!'
arrayVar   = [1, 5, 8]
symbolVar  = :abc
hashVar    = {'Name' => 'jack', 'family' => 'Scalia', 'Age' => 7}

puts "intVar    = #{intVar}"
```

```
10  puts "floatVar  = #{floatVar}"
11  puts "boolVar   = #{boolVar}"
12  puts "StringVar = #{stringVar}"
13  puts "arrayVar  = #{arrayVar}"
14  puts "symbolVar = #{symbolVar}"
15  puts "hashVar   = #{hashVar}"
```

```
intVar    = 10
floatVar  = 12.5
boolVar   = true
StringVar = Hello World!
arrayVar  = [1, 5, 8]
symbolVar = abc
hashVar   = {"Name"=>"jack", "family"=>"Scalia", :Age=>7}
```

در خطوط ۷-۱، متغیر ها تعریف شده اند. اما نوع این متغیرها چیست؟ Ruby نوع متغیرها را بسته به مقداری که به آنها اختصاص داده می شود در نظر می گیرد. مثلا نوع متغیر stringVar در خط ۴ از نوع رشته است، چون یک مقدار رشته ای به آن اختصاص داده شده است. به خطوط ۵، ۶ و ۷ کد بالا توجه کنید. در خط ۵ یک متغیر تعریف شده است و نوع داده ای که به آن اختصاص داده شده است از نوع array است. همانطور که در درس قبل اشاره شد، برای تعریف array علامت [] به کار می رود و آیتم های داخل آن به وسیله کاما از هم جدا می شوند :

```
arrayVar = [1, 5, 8]
```

در خط ۶ هم یک متغیر تعریف شده است و یک مقدار از نوع symbol به آن اختصاص داده شده است. symbol ها نه متغیر هستند و نه string هستند بلکه آنها را به عنوان برچسب ها در نظر می گیرند و در هش ها پرکاربرد هستند. در جلسات آینده کاربرد آنها را در هش ها خواهید دید و با علامت دو نقطه (:) نشان داده می شوند. و اما در خط ۷ یک نوع hash تعریف شده است. برای تعریف hash بین کلید و مقدار علامت => و بین کلید/مقدارها هم علامت , قرار می گیرد :

```
hashVar = {Key1 => Value1, Key2 => Value2, Key3 => Value3}
```

مثلا در مثال بالا یک hash تعریف کرده ایم که سه آیتم یا کلید/مقدار دارد که بین آنها علامت کاما , قرار داده ایم. ولی بین یک کلید و مقدار مربوط به آن علامت : قرار گرفته است. نکته ای که در خطوط ۱۵-۹ کد بالا وجود دارد و بهتر است که در همینجا به آن اشاره شود این است که برای نوشتن نام یک متغیر، کافیست نام آن را در داخل علامت "" بنویسید و اگر بخواهید مقدار متغیر را چاپ کنید باید نام متغیر را در بین {}# بنویسید. مثلا در خط ۹ کد بالا :

```
puts "intVar    =    #{intVar}"
```

سمت چپ علامت مساوی، باعث چاپ نام متغیر یعنی intVar و سمت راست علامت مساوی باعث چاپ عدد ۱۰، یعنی مقدار متغیر می شود.

برای اختصاص یک مقدار به چند متغیر می توان به صورت زیر عمل کرد :

```
identifier1 =  identifier2 = ... indentifierN = Value
```

به مثال زیر توجه کنید :

```
num1 = num2 = num3 = num4 = num5 = 10
```

```ruby
message1 = message2 = message3 = "Hello World!"

puts num1
puts num4
puts message1
puts message3
```

```
10
10
Hello World!
Hello World!
```

دقت کنید که برای متغیرهای تعریف شده در حالت بالا یک خانه حافظه تخصیص داده می شود، یعنی مقدار ۱۰ در حافظه ذخیره شده و متغیرهای num1 و num2 و num3 و num4 و num5 به آن خانه از حافظه اشاره می کنند. همچنین می توان چند متغیر را تعریف کرد و برای هر یک از آن ها مقدار جداگانه ای مشخص نمود:

```ruby
identifier1, identifier2, ... indentifierN = Value1, Value2, ... ValueN
```

به مثال زیر توجه کنید:

```ruby
num1, num2, message1 = 10, 12.5, 'Hello World!'

puts num1
puts num2
puts message1
```

```
10
12.5
Hello World!
```

در کد بالا مقدار num1 برابر 10، num2 برابر 12.5 و message1 برابر Hello World! می باشد. در Ruby، متغیر ها هم باید تعریف و هم مقداردهی شوند. یعنی اگر متغیری را تعریف کرده و به آن مقداری را اختصاص ندهید و برنامه را اجرا کنید با خطا مواجه می شوید :

```ruby
number

puts number
```

```
undefined local variable or method `number' for main:Object (NameError)
```

همانطور که در درس قبل هم اشاره کردیم، یک رشته در اصل یک مجموعه از کاراکترهاست که در داخل علامت "" یا ' ' قرار دارند. هر کدام از این کاراکترها دارای یک اندیس است که به وسیله آن اندیس قابل دسترسی هستند. اندیس کاراکترها در رشته از ۰ شروع می شود. به رشته زیر توجه کنید :

```ruby
message = 'Hello World!'
```

در رشته بالا اندیس کاراکتر 0 برابر H است. برای درک بهتر به شکل زیر توجه کنید :

```
H e l l o   W o r l d  !

0 1 2 3 4 5 6 7 8 9 10 11
```

حال برای چاپ یک کاراکتر مثلا W از این رشته کافیست که به صورت زیر عمل کنیم :

```
message = 'Hello World!'

puts message[6]
```

```
W
```

همانطور که در کد بالا مشاهده می کنید کافیست که نام متغیر را نوشته، در جلوی آن یک جفت کروشه و در داخل کروشه ها اندیس آن کاراکتری را که می خواهیم چاپ شود را بنویسیم. چاپ مقدار با استفاده از اندیس در مورد array هم صدق می کند :

```
arrayVar  = [1, 5, 8]

puts arrayVar[2]
```

```
8
```

و اما در مورد hash، شما باید نام کلید را بنویسید تا مقدار آن برای شما نمایش داده شود:

```
hashVar = {'Name' => 'jack', 'Family' => 'Scalia', 'Age' => 7}

puts hashVar['Family']
```

```
Scalia
```

نکته ای که بهتر است در همینجا به آن اشاره کنیم این است که کلید/مقدارها در hash می توانند از هر نوعی باشند و شما برای چاپ مقدار مربوط به یک کلید باید نام کلید را دقیق بنویسید. به مثال زیر توجه کنید :

```
hashVar   = {1 => 'Jack', '2' => 'Scalia', 3 => 7}

puts hashVar['2']
```

```
Scalia
```

در مثال بالا ما مقدار کلید '۲' را چاپ کرده ایم. حال اگر به جای '۲' عدد ۲ را بنویسیم، یعنی علامت کوتیشن را نگذاریم، هیچ خروجی به ما نمایش داده نمی شود :

```
hashVar = {1 => 'Jack', '2' => 'Scalia', 3 => 7}

puts hashVar[2]
```

ثابت ها

ثابت‌ها، انواعی از متغیرها هستند که مقدار آنها در طول برنامه تغییر نمی‌کند. بعد از این که به ثابت‌ها مقدار اولیه اختصاص داده شد هرگز در زمان اجرای برنامه نمی‌توان آن را تغییر داد. تفاوت ثابت ها با متغیرها در این است که نام ثابت‌ها را طبق قرارداد با حروف بزرگ می‌نویسند. نحوه تعریف ثابت در زیر آمده است :

```
IDENTIFIER = initial_value
```

مثال :

```
NUMBER = 1

NUMBER = 10 #ERROR, Cant modify a constant
```

در این مثال می‌بینید که مقدار دادن به یک ثابت، که قبلاً مقدار دهی شده برنامه را با خطا مواجه می‌کند. چون از دید مفسر، این ها یکبار برای همیشه تعریف شده اند و نمی توان آنها را تغییر داد. نکته‌ی دیگری که نباید فراموش شود این است که نباید مقدار ثابت را با مقدار دیگر متغیرهای تعریف شده در برنامه برابر قرار داد. مثال :

```
MY_CONST = 1
someVariable = 10

MY_CONST = someVariable
```

ممکن است این سؤال برایتان پیش آمده باشد که دلیل استفاده از ثابت‌ها چیست؟ اگر مطمئن هستید که مقادیری در برنامه وجود دارند که هرگز در طول برنامه تغییر نمی‌کنند بهتر است که آنها را به صورت ثابت تعریف کنید. این کار هر چند کوچک کیفیت برنامه شما را بالا می‌برد.

تبدیل انواع داده

در زبان Ruby امکان تبدیل یک نوع به نوع دیگر وجود دارد که اصطلاحاً به آن Type Casting گفته می شود. Ruby دارای مجموعه ای از توابع از پیش تعریف شده است، که می توانند مقادیر را از یک نوع بـه نوع دیگر تبدیل کنند. در جدول زیر به برخی از این توابع اشاره شده است :

متد	کاربرد
Value.to_i Value.to_int Integer(Value)	Value را به نوع int یا صحیح تبدیل می کند.
Value.to_f Float(Value)	Value را به نوع float یا اعشاری تبدیل می کند.
Value.to_c	Value را به نوع Complex تبدیل می کند.
Value.to_d	Value را به نوع decimal تبدیل می کند.
Value.to_r	Value را به نوع گویا تبدیل می کند.
Value.to_s	Value را به نوع رشته تبدیل می کند.

مقدار Value هر نوع داده‌ای می‌تواند باشد اما باید مطمئن شد که امکان تبدیل به نوع داده‌ای مورد نظر وجود داشته باشد. در برنامه زیر یک نمونه از تبدیل متغیرها با استفاده متدهای بالا نمایش داده شده است :

```
x = 9.99
convertedValue = x.to_int

puts "Original value is: #{x}"
puts "Converted value is: #{convertedValue}"
```

```
Original value is: 9.99
Converted value is: 9
```

در کد بالا متغیر x از نوع float است و ما با استفاده از متد to_int آن را به نوع صحیح تبدیل کرده ایم. این کار باعث می شود که قسمت اعشار این متغیر حذف شود. خط دوم کد بالا را به صورت های زیر هم می توان نوشت:

```
convertedValue = x.to_i
```

یا

```
convertedValue = Integer(x)
```

در کد بالا مشاهده می کنید که برای استفاده از متدهای تبدیل نوع، ابتدا باید نام متغیر، سپس علامت نقطه و در نهایت نام متد را بنویسیم. برای به دست آوردن نوع یک متغیر هم می توان از متد class استفاده کرد. به مثال زیر توجه کنید :

```
x = 9.99

puts x.class
```

در خط اول یک متغیر تعریف کرده ایم. حال اگر بخواهیم که نوع داده ای که در این متغیر قرار داده شده است چیست، کافیست که نام متغیر را نوشته، سپس علامت نقطه و بعد متد class را بنویسیم. بعد از اجرای کد بالا، نتیجه به صورت زیر خواهد بود :

```
Float
```

همانطور که مشاهده می کنید، نتیجه نمایش کلمه Float است. یعنی متغیر ما از نوع Float می باشد.

عبارات و عملگرها

ابتدا با دو کلمه آشنا شوید :

- عملگر: نمادهایی هستند که اعمال خاص انجام می‌دهند.
- عملوند: مقادیری که عملگرها بر روی آنها عملی انجام می‌دهند.

مثلاً X+Y: یک عبارت است که در آن X و Y عملوند و علامت + عملگر به حساب می‌آیند.

زبانهای برنامه نویسی جدید دارای عملگرهایی هستند که از اجزاء معمول زبان به حساب می‌آیند. Ruby دارای عملگرهای مختلفی از جمله عملگرهای ریاضی، تخصیصی، مقایسه‌ای، منطقی و بیتی می‌باشد. از عملگرهای ساده ریاضی می‌توان به عملگر جمع و تفریق اشاره کرد. سه نوع عملگر در Ruby وجود دارد :

- یگانی - (Unary) به یک عملوند نیاز دارد
- دودویی - (Binary) به دو عملوند نیاز دارد
- سه تایی - (Ternary) به سه عملوند نیاز دارد

انواع مختلف عملگر که در این بخش مورد بحث قرار می‌گیرند، عبارتند از :

- عملگرهای ریاضی
- عملگرهای تخصیصی
- عملگرهای مقایسه‌ای
- عملگرهای منطقی
- عملگرهای بیتی
- عملگرهای محدوده

عملگرهای ریاضی

Ruby از عملگرهای ریاضی برای انجام محاسبات استفاده می‌کند. جدول زیر عملگرهای ریاضی Ruby را نشان می‌دهد :

عملگر	دسته	مثال	نتیجه
+	Binary	var1 = var2 + var3	Var1 برابر است با حاصل جمع var2 و var3
-	Binary	var1 = var2 - var3	Var1 برابر است با حاصل تفریق var2 و var3
*	Binary	var1 = var2 * var3	Var1 برابر است با حاصلضرب var2 در var3
/	Binary	var1 = var2 / var3	Var1 برابر است با حاصل تقسیم var2 بر var3
%	Binary	var1 = var2 % var3	Var1 برابر است با باقیمانده تقسیم var2 و var3
**	Unary	var1 = var2 ** var3	Var1 برابر است با مقدار var2 به توان var3

مثال بالا در از نوع عددی استفاده شده است. اما استفاده از عملگرهای ریاضی برای نوع رشته‌ای نتیجه متفاوتی دارد. اگر از عملگر + برای رشته‌ها استفاده کنیم دو رشته را با هم ترکیب کرده و به هم می‌چسباند. حال می‌توانیم با ایجاد یک برنامه نحوه عملکرد عملگرهای ریاضی در Ruby را یاد بگیریم :

```ruby
#Assign test values
num1 = 5
num2 = 3

#Demonstrate use of mathematical operators
puts "The sum of #{num1} and #{num2} is #{num1 +  num2}."
puts "The difference of #{num1} and #{num2} is #{num1 -  num2}."
puts "The product of #{num1} and #{num2} is #{num1 *  num2}."
puts "The quotient of #{num1} and #{num2} is #{num1 /  num2}."
puts "The remainder of #{num1} divided by #{num2} is #{num1 %  num2}."
puts "The result of #{num1} power #{num2} is #{num1 **  num2}."

#Demonstrate concatenation on strings using the + operator
msg1 = "Hello "
msg2 = "World!"
puts msg1 + msg2
```

```
The sum of 5 and 3 is 8.
The difference of 5 and 3 is 2.
The product of 5 and 3 is 15.
The quotient of 5 and 3 is 1.
The remainder of 5 divided by 3 is 2.
The result of 5 power 3 is 125.
Hello World!
```

برنامه بالا نتیجه هر عبارت را نشان می‌دهد. در این برنامه از متد puts() برای نشان دادن نتایج در سطرهای متفاوت استفاده شده است . Ruby خط جدید و فاصله و فضای خالی را نادیده می‌گیرد. در خط ۱۷ مشاهده می‌کنید که دو رشته به وسیله عملگر + به هم متصل شده‌اند. نتیجه

استفاده از عملگر + برای چسباندن دو کلمه “Hello ” و “World!” رشته “Hello World!” خواهد بود. به فاصله‌های خالی بعد از اولین کلمه توجه کنید. اگر آنها را حذف کنید، از خروجی برنامه نیز حذف می‌شوند.

عملگرهای تخصیصی (جایگزینی)

نوع دیگر از عملگرهای Ruby عملگرهای جایگزینی نام دارند. این عملگرها مقدار متغیر سمت راست خود را در متغیر سمت چپ قرار می‌دهند.

جدول زیر انواع عملگرهای تخصیصی در Ruby را نشان می‌دهد:

عملگر	مثال	نتیجه
=	var1 = var2	مقدار var1 برابر است با مقدار var2
+=	var1 += var2	مقدار var1 برابر است با حاصل جمع var1 و var2
-=	var1 -= var2	مقدار var1 برابر است با حاصل تفریق var1 و var2
*=	var1 *= var2	مقدار var1 برابر است با حاصل ضرب var1 در var2
/=	var1 /= var2	مقدار var1 برابر است با حاصل تقسیم var1 بر var2
%=	var1 %= var2	مقدار var1 برابر است با باقیمانده تقسیم var1 بر var2
**=	var1 **= var2	مقدار var1 برابر است با var1 به توان var2

از عملگر =+ برای اتصال دو رشته نیز می‌توان استفاده کرد. استفاده از این نوع عملگرها در واقع یک نوع خلاصه نویسی در کد است. مثلاً شکل اصلی کد var1 += var2 به صورت var1 = var1 + var2 می‌باشد. این حالت کدنویسی زمانی کارایی خود را نشان می‌دهد که نام متغیرها طولانی باشد. برنامه زیر چگونگی استفاده از عملگرهای تخصیصی و تأثیر آنها را بر متغیرها نشان می‌دهد:

```ruby
puts "Assigning 10 to number..."
number = 10
puts "Number = #{number}"

puts "Adding 10 to number..."
number += 10
puts "Number = #{number}"

puts "Subtracting 10 from number..."
number -= 10

puts "Number = #{number}"
```

```
Assigning 10 to number...
Number = 10
Adding 10 to number...
Number = 20
Subtracting 10 from number...
Number = 10
```

در برنامه از ۳ عملگر تخصیصی استفاده شده است. ابتدا یک متغیر و مقدار 10 با استفاده از عملگر = به آن اختصاص داده شده است. سپس به آن با استفاده از عملگر =+ مقدار ۱۰ اضافه شده است. و در آخر به وسیله عملگر =- عدد ۱۰ از آن کم شده است.

عملگرهای مقایسه ای

از عملگرهای مقایسه‌ای برای مقایسه مقادیر استفاده می‌شود. نتیجه این مقادیر یک مقدار بولی (منطقی) است. این عملگرها اگر نتیجه مقایسه دو مقدار درست باشد مقدار ۱ و اگر نتیجه مقایسه اشتباه باشد مقدار ۰ را نشان می‌دهند. این عملگرها به طور معمول در دستورات شرطی به کار می‌روند به این ترتیب که باعث ادامه یا توقف دستور شرطی می‌شوند. جدول زیر عملگرهای مقایسه‌ای در Ruby را نشان می‌دهد:

عملگر	دسته	مثال	نتیجه
==	Binary	var1 = var2 == var3	var1 در صورتی True است که مقدار var2 با مقدار var3 برابر باشد در غیر اینصورت False است
!=	Binary	var1 = var2 != var3	var1 در صورتی True است که مقدار var2 با مقدار var3 برابر نباشد در غیر اینصورت False است
<>	Binary	var1 = var2 <> var3	var1 در صورتی True است که مقدار var2 با مقدار var3 برابر نباشد در غیر اینصورت False است
<	Binary	var1 = var2 < var3	var1 در صورتی True است که مقدار var2 کوچک‌تر از var3 مقدار باشد در غیر اینصورت False است
>	Binary	var1 = var2 > var3	var1 در صورتی True است که مقدار var2 بزرگ‌تر ازمقدار var3 باشد در غیر اینصورت False است
<=	Binary	var1 = var2 <= var3	var1 در صورتی True است که مقدار var2 کوچک‌تر یا مساوی مقدار var3 باشد در غیر اینصورت False است
>=	Binary	var1 = var2 >= var3	var1 در صورتی True است که مقدار var2 بزرگ‌تر یا مساوی var3 مقدار باشد در غیر اینصورت False است

برنامه زیر نحوه عملکرد این عملگرها را نشان می‌دهد :

```
num1 = 10;
num2 = 5;

puts "#{num1} == #{num2} : #{num1 == num2}"
puts "#{num1} != #{num2} : #{num1 != num2}"
puts "#{num1} <> #{num2} : #{num1 != num2}"
puts "#{num1} <  #{num2} : #{num1 <  num2}"
puts "#{num1} >  #{num2} : #{num1 >  num2}"
puts "#{num1} <= #{num2} : #{num1 <= num2}"
puts "#{num1} >= #{num2} : #{num1 >= num2}"
```

```
10 == 5 : False
```

```
10 != 5 : True
10 <> 5 : True
10 <  5 : False
10 >  5 : True
10 <= 5 : False
10 >= 5 : True
```

در مثال بالا ابتدا دو متغیر را که می‌خواهیم با هم مقایسه کنیم را ایجاد کرده و به آنها مقادیری اختصاص می‌دهیم. سپس با استفاده از یک عملگر مقایسه‌ای آنها را با هم مقایسه کرده و نتیجه را چاپ می‌کنیم. به این نکته توجه کنید که هنگام مقایسه دو متغیر از عملگر == به جای عملگر = باید استفاده شود. عملگر = عملگر تخصیصی است و در عبارتی مانند x = y مقدار y را در به x اختصاص می‌دهد. عملگر == عملگر مقایسه‌ای است که دو مقدار را با هم مقایسه می‌کند مانند x==y و اینطور خوانده می‌شود x برابر است با y.

عملگرهای منطقی

عملگرهای منطقی بر روی عبارات منطقی عمل می‌کنند و نتیجه آنها نیز یک مقدار بولی است. از این عملگرها اغلب برای شرطهای پیچیده استفاده می‌شود. همانطور که قبلاً یاد گرفتید مقادیر بولی می‌توانند false یا true باشند. فرض کنید که var2 و var3 دو مقدار بولی هستند.

عملگر	مثال	توضیح
and	var1 = var2 and var3	اگر هر دو عملوند true باشند، مقدار true را بر می گرداند.
&&	var1 = var2 && var3	اگر هر دو عملوند true باشند، مقدار true را بر می گرداند.
or	var1 = var2 or var3	اگر یکی یا هر دو مقدار دو طرف عملگر true باشد،مقدار true را بر می گرداند.
||	var1 = var2 || var3	اگر یکی یا هر دو مقدار دو طرف عملگر true باشد،مقدار true را بر می گرداند.
not	var1 = not (var1)	اگر مقدار یک عبارت true باشد آنرا false و اگر false باشد آنرا true می کند.
!	var1 = !(var1)	اگر مقدار یک عبارت true باشد آنرا false و اگر false باشد آنرا true می کند.

عملگر منطقی and یا &&

اگر مقادیر دو طرف عملگر True ،and باشند عملگر and مقدار True را بر می‌گرداند. در غیر اینصورت اگر یکی از مقادیر یا هر دوی آنها False باشند مقدار False را بر می‌گرداند. در زیر جدول درستی عملگر and نشان داده شده است:

X and Y	Y	X
True	True	True
False	False	True
False	True	False
False	False	False

برای درک بهتر تأثیر عملگر and یاد آوری می‌کنم که این عملگر فقط در صورتی مقدار True را نشان می‌دهد که هر دو عملوند مقدارشان True باشد. در غیر اینصورت نتیجه تمام ترکیبهای بعدی False خواهد شد. استفاده از عملگر and مانند استفاده از عملگرهای مقایسه‌ای است. به عنوان مثال نتیجه عبارت زیر درست (True) است اگر سن (age) بزرگ‌تر از ۱۸ و salary کوچک‌تر از ۱۰۰۰ باشد.

```
result = (age > 18) and (salary < 1000)
```

عملگر and زمانی کارآمد است که ما با محدود خاصی از اعداد سرو کار داریم. مثلاً عبارت 10 <= x <= 100 بدین معنی است که x می‌تواند مقداری شامل اعداد ۱۰ تا ۱۰۰ را بگیرد. حال برای انتخاب اعداد خارج از این محدوده می‌توان از عملگر منطقی and به صورت زیر استفاده کرد.

```
inRange = (number <= 10) and  (number >= 100)
```

عملگر منطقی or یا ||

اگر یکی یا هر دو مقدار دو طرف عملگر or، درست (True) باشد، عملگر or مقدار True را بر می‌گرداند. جدول درستی عملگر or در زیر نشان داده شده است:

X	Y	X or Y
True	True	True
True	False	True
False	True	True
False	False	False

در جدول بالا مشاهده می‌کنید که عملگر or در صورتی مقدار False را بر می‌گرداند که مقادیر دو طرف آن False باشند. کد زیر را در نظر بگیرید.

نتیجه این کد در صورتی درست (True) است که رتبه نهایی دانش آموز (finalGrade) بزرگ‌تر از ۷۵ یا یا نمره نهایی امتحان آن ۱۰۰ باشد.

```
isPassed = (finalGrade >= 75) or (finalExam == 100)
```

عملگر منطقی not یا !

برخلاف دو اپراتور or و and عملگر منطقی NOT یک عملگر یگانی است و فقط به یک عملوند نیاز دارد. این عملگر یک مقدار یا اصطلاح بولی را نفی می‌کند. مثلاً اگر عبارت یا مقدار True باشد آنرا False و اگر False باشد آنرا True می‌کند. جدول زیر عملکرد اپراتور NOT را نشان می‌دهد:

X	not X
True	False
False	True

نتیجه کد زیر در صورتی درست است که age (سن) بزرگ‌تر یا مساوی ۱۸ نباشد.

```
isMinor = not(age >= 18)
```

عملگرهای بیتی

عملگرهای بیتی به شما اجازه می‌دهند که شکل باینری انواع داده‌ها را دستکاری کنید. برای درک بهتر این درس توصیه می‌شود که شما سیستم باینری و نحوه تبدیل اعداد دهدهی به باینری را از لینک زیر یاد بگیرید :

```
http://www.w3-farsi.com/?p=5698
```

در سیستم باینری (دودویی) که کامپیوتر از آن استفاده می‌کند وضعیت هر چیز یا خاموش است یا روشن. برای نشان دادن حالت روشن از عدد ۱ و برای نشان دادن حالت خاموش از عدد ۰ استفاده می‌شود. بنابراین اعداد باینری فقط می‌توانند صفر یا یک باشند. اعداد باینری را اعداد در مبنای ۲ و اعداد اعشاری را اعداد در مبنای ۱۰ می گویند. یک بیت نشان دهنده یک رقم باینری است و هر بایت نشان دهنده ۸ بیت است. به عنوان مثال برای یک داده از نوع int به ۳۲ بیت یا ۴ بایت فضا برای ذخیره آن نیاز داریم، این بدین معناست که اعداد از ۳۲ رقم ۰ و ۱ برای ذخیره استفاده می‌کنند. برای مثال عدد ۱۰۰ وقتی به عنوان یک متغیر از نوع int ذخیره می‌شود در کامپیوتر به صورت زیر خوانده می‌شود :

```
00000000000000000000000001100100
```

عدد ۱۰۰ در مبنای ده معادل عدد ۱۱۰۰۱۰۰ در مبنای ۲ است. در اینجا ۷ رقم سمت راست نشان دهنده عدد ۱۰۰ در مبنای ۲ است و مابقی صفرهای سمت راست برای پر کردن بیت هایی است که عدد از نوع int نیاز دارد. به این نکته توجه کنید که اعداد باینری از سمت راست به چپ خوانده می‌شوند. عملگرهای بیتی Ruby در جدول زیر نشان داده شده‌اند :

مثال	عملگر
x = y & z	&
x = y | z	|
x = y ^ z	^
x = ~y	~
x &= y	&=
x |= y	|=
x ^= y	^=

عملگر بیتی AND(&)

عملگر بیتی AND کاری شبیه عملگر منطقی AND انجام می‌دهد با این تفاوت که این عملگر بر روی بیتها کار می‌کند. اگر مقادیر دو طرف آن ۱ باشد مقدار ۱ را بر می‌گرداند و اگر یکی یا هر دو طرف آن صفر باشد مقدار صفر را بر می‌گرداند. جدول درستی عمگر بیتی AND در زیر آمده است:

X AND Y	Y	X
1	1	1

0	0	1
0	1	0
0	0	0

در زیر نحوه استفاده از عملگر بیتی AND آمده است :

```
result = 5 & 3

puts result
```

```
1
```

همانطور که در مثال بالا مشاهده می‌کنید نتیجه عملکرد عملگر AND بر روی دو مقدار ۵ و ۳ عدد یک می‌شود. اجازه بدهید ببینیم که چطور این نتیجه را به دست می‌آید:

```
5:  00000000000000000000000000000101
3:  00000000000000000000000000000011
-----------------------------------
1:  00000000000000000000000000000001
```

ابتدا دو عدد ۵ و ۳ به معادل باینری‌شان تبدیل می‌شوند. از آنجاییکه هر عدد صحیح32 (int) بیت است از صفر برای پر کردن بیتهای خالی استفاده می‌کنیم. با استفاده از جدول درستی عملگر بیتی AND می‌توان فهمید که چرا نتیجه عدد یک می‌شود.

عملگر بیتی OR(|)

اگر مقادیر دو طرف عملگر بیتی OR هر دو صفر باشند نتیجه صفر در غیر اینصورت ۱ خواهد شد. جدول درستی این عملگر در زیر آمده است :

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

نتیجه عملگر بیتی OR در صورتی صفر است که عملوندهای دو طرف آن صفر باشند. اگر فقط یکی از دو عملوند یک باشد نتیجه یک خواهد شد.

به مثال زیر توجه کنید :

```
result = 7 | 9

puts result
```

```
15
```

وقتی که از عملگر بیتی OR برای دو مقدار در مثال بالا (۷ و ۹) استفاده می‌کنیم نتیجه ۱۵ می‌شود. حال بررسی می‌کنیم که چرا این نتیجه به دست آمده است؟

```
 7: 00000000000000000000000000000111
 9: 00000000000000000000000000001001
---------------------------------
15: 00000000000000000000000000001111
```

با استفاده از جدول درستی عملگر بیتی OR می‌توان نتیجه استفاده از این عملگر را تشخیص داد. عدد ۱۱۱۱ باینری معادل عدد ۱۵ صحیح است.

عملگر بیتی XOR(^)

جدول درستی این عملگر در زیر آمده است :

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

در صورتیکه عملوندهای دو طرف این عملگر هر دو صفر یا هر دو یک باشند نتیجه صفر در غیر اینصورت نتیجه یک می‌شود. در مثال زیر تأثیر عملگر بیتی XOR را بر روی دو مقدار مشاهده می‌کنید :

```
result = 5 ^ 7

puts result
```

```
2
```

در زیر معادل باینری اعداد بالا (۵ و ۷) نشان داده شده است.

```
5: 00000000000000000000000000000101
7: 00000000000000000000000000000111
---------------------------------
2: 00000000000000000000000000000010
```

با نگاه کردن به جدول درستی عملگر بیتی XOR، می‌توان فهمید که چرا نتیجه عدد ۲ می‌شود.

عملگر بیتی NOT(~)

این عملگر یک عملگر یگانی است و فقط به یک عملوند نیاز دارد. در زیر جدول درستی این عملگر آمده است:

X	NOT X
1	0

1	0

عملگر بیتی NOT مقادیر بیتها را معکوس می‌کند. در زیر چگونگی استفاده از این عملگر آمده است :

```
result = ~ 7

puts result
```

```
-8
```

به نمایش باینری مثال بالا که در زیر نشان داده شده است توجه نمایید.

```
 7: 00000000000000000000000000000111
-----------------------------------
-8: 11111111111111111111111111111000
```

عملگر بیتی تغییر مکان (shift)

این نوع عملگرها به شما اجازه می‌دهند که بیتها را به سمت چپ یا راست جا به جا کنید. دو نوع عملگر بیتی تغییر مکان وجود دارد که هر کدام دو عملوند قبول می‌کنند. عملوند سمت چپ این عملگرها حالت باینری یک مقدار و عملوند سمت راست تعداد جابه جایی بیت ها را نشان می‌دهد.

عملگر	نام	دسته	مثال
>>	تغییر مکان به سمت چپ	Binary	x = y << 2
<<	تغییر مکان به سمت راست	Binary	x = y >> 2

عملگر تغییر مکان به سمت چپ

این عملگر بیتهای عملوند سمت چپ را به تعداد n مکان مشخص شده توسط عملوند سمت راست، به سمت چپ منتقل می‌کند. به عنوان مثال :

```
result = 10 << 2

puts result
```

```
40
```

در مثال بالا ما بیت‌های مقدار ۱۰ را دو مکان به سمت چپ منتقل کرده‌ایم، حال بیایید تأثیر این انتقال را بررسی کنیم :

```
10: 00000000000000000000000000001010
-----------------------------------
40: 00000000000000000000000000101000
```

مشاهده می‌کنید که همه بیت‌ها به اندازه دو واحد به سمت چپ منتقل شده‌اند. در این انتقال دو صفر از صفرهای سمت چپ کم می‌شود و در عوض دو صفر به سمت راست اضافه می‌شود.

عملگر تغییر مکان به سمت راست

این عملگر شبیه به عمگر تغییر مکان به سمت چپ است با این تفاوت که بیت‌ها را به سمت راست جا به جا می‌کند. به عنوان مثال :

```
result = 100 >> 4

puts result
```

```
6
```

با استفاده از عملگر تغییر مکان به سمت راست بیت‌های مقدار ۱۰۰ را به اندازه ۴ واحد به سمت چپ جا به جا می‌کنیم. اجازه بدهید تأثیر این جا به جایی را مورد بررسی قرار دهیم :

```
100: 00000000000000000000000001100100
-----------------------------------
  6: 00000000000000000000000000000110
```

هر بیت به اندازه ۴ واحد به سمت راست منتقل می‌شود، بنابراین ۴ بیت اول سمت راست حذف شده و چهار صفر به سمت چپ اضافه می‌شود.

عملگرهای محدوده

در Ruby دو عملگر وجود دارند که از آنها برای ایجاد محدوده استفاده می شود. به جدول زیر توجه کنید :

عملگر	توضیح
..	برای ایجاد یک بازه به کار می رود که انتهای بازه، جزء بازه محسوب می شود.
...	برای ایجاد یک بازه به کار می رود که انتهای بازه، جزء بازه محسوب نمی شود.

برای روشن شدن عملکرد این دو عملگر به کد زیر توجه کنید :

```
for num in 1..5
  puts num
end
```

```
1
2
3
4
5
```

مشاهده می کنید که عدد ۵ جز بازه محسوب می شود. حال اگر در کد بالا به جای دو نقطه، سه نقطه بگذارید، عدد ۵ در خروجی چاپ نمی شود.

از این عملگرها می توان برای ایجاد محدوده ای از کاراکترها هم استفاده کرد :

```
for char in 'A'...'K'
  puts char
end
```

```
A
B
C
D
E
F
G
H
I
J
```

تقدم عملگرها

تقدم عملگرها مشخص می‌کند که در محاسباتی که بیش از دو عملوند دارند ابتدا کدام عملگر اثرش را اعمال کند. عملگرها در Ruby در محاسبات دارای حق تقدم هستند. به عنوان مثال :

```
number = 1 + 2 * 3 / 1
```

اگر ما حق تقدم عملگرها را رعایت نکنیم و عبارت بالا را از سمت چپ به راست انجام دهیم نتیجه ۹ خواهد شد (۳=۲+۱ سپس ۹=۳×۳ و در آخر ۹/۱=۹). اما کامپایلر با توجه به تقدم عملگرها محاسبات را انجام می‌دهد. برای مثال عمل ضرب و تقسیم نسبت به جمع و تفریق تقدم دارند. بنابراین در مثال فوق ابتدا عدد ۲ ضربدر ۳ و سپس نتیجه آنها تقسیم بر ۱ می‌شود که نتیجه ۶ به دست می‌آید. در آخر عدد ۶ با ۱ جمع می‌شود و عدد ۷ حاصل می‌شود. در جدول زیر تقدم عملگرهای Ruby آمده است :

تقدم	عملگر
بیشترین	::
	[ ] [ ]=
	**
	! ~ + -
	* / %
	+ -
	>> <<
	&
	^ |
	<= < > >=
	<=> == === != =~ !~
	&&
	||
	.. …
	? :
	= %= { /= -= += |= &= >>= <<= *= &&= ||= **=
	defined?
	not
کمترین	or and

ابتدا عملگرهای با بالاترین و سپس عملگرهای با پایین‌ترین حق تقدم در محاسبات تأثیر می‌گذارند. برای ایجاد خوانایی در تقدم عملگرها و انجام محاسباتی که در آنها از عملگرهای زیادی استفاده می‌شود از پرانتز استفاده می‌کنیم :

```
number = ( 1 + 2 ) * ( 3 / 4 ) % ( 5 - ( 6 * 7 ))
```

در مثال بالا ابتدا هر کدام از عباراتی که داخل پرانتز هستند مورد محاسبه قرار می‌گیرند. به نکته‌ای در مورد عبارتی که در داخل پرانتز سوم قرار دارد توجه کنید. در این عبارت ابتدا مقدار داخلی‌ترین پرانتز مورد محاسبه قرار می‌گیرد یعنی مقدار ۶ ضربدر ۷ شده و سپس از ۵ کم می‌شود. اگر دو یا چند عملگر با حق تقدم یکسان موجود باشد ابتدا باید هر کدام از عملگرها را که در ابتدای عبارت می‌آیند مورد ارزیابی قرار دهید. به عنوان مثال :

```
number = 3 * 2 + 8 / 4
```

هر دو عملگر * و / دارای حق تقدم یکسانی هستند. بنابر این شما باید از چپ به راست آنها را در محاسبات تأثیر دهید. یعنی ابتدا ۳ را ضربدر ۲ می‌کنید و سپس عدد ۸ را بر ۴ تقسیم می‌کنید. در نهایت نتیجه دو عبارت را جمع کرده و در متغیر number قرار می‌دهید.

گرفتن ورودی از کاربر

Ruby متد gets را برای گرفتن ورودی از کاربر، در اختیار شما قرار می‌دهد. همانطور که از نام این متد پیداست، تمام کاراکترهایی را که شما در محیط برنامه نویسی تایپ می‌کنید تا زمانی که دکمه Enter را می‌زنید، می‌خواند. به برنامه زیر توجه کنید :

```
1  print 'Enter your name: '
2  name = gets
3
4  print 'Enter your age: '
5  age = gets
6
7  print 'Enter your height: '
8  height = gets
9
10 # Print a blank line
11 puts
12
13 # Show the details you typed
14 puts "Name is #{name}"
15 puts "Age is #{age}"
16 puts "Height is #{height}"
```

```
Enter your name: John
Enter your age: 18
Enter your height: 160.5

Name is John.
Age is 18.
Height is 160.5.
```

در کد بالا و در خطوط ۱، ۴ و ۷ با استفاده از متد print پیغام هایی را به کاربر نشان داده ایم. دلیل استفاده از متد print در این کد، این است که متد print، خط جدید ایجاد نمی کند، در نتیجه نشانگر ماوس، بعد از نمایش پیغام به کاربر در همان خط و منتظر ورود اطلاعات توسط کاربر می ماند. برنامه از کاربر می‌خواهد که نام خود را وارد کند (خط ۱). در خط ۲ شما به عنوان کاربر نام خود را با استفاده از متد gets وارد می‌کنید. سپس برنامه از ما سن را سؤال می‌کند (خط ۴). در خط ۵ سن را وارد می کنید. این کار در خطوط ۷ و ۸ برای دریافت قد کاربر هم تکرار می شود. در خط ۱۱ هم یک خط فاصله به وسیله متد puts ایجاد کرده ایم تا بین ورودی های شما و خروجی فاصله ای جهت تفکیک ایجاد شود. حال برنامه را اجرا کرده و با وارد کردن مقادیر مورد نظر نتیجه را مشاهده کنید.

ساختارهای تصمیم

تقریبا همه زبانهای برنامه نویسی به شما اجازه اجرای کد را در شرایط مطمئن می دهند. حال تصور کنید که یک برنامه دارای ساختار تصمیم گیری نباشد و همه کدها را اجرا کند. این حالت شاید فقط برای چاپ یک پیغام در صفحه مناسب باشد ولی فرض کنید که شما بخواهید اگر مقدار یک متغیر با یک عدد برابر باشد سپس یک پیغام چاپ شود آن وقت با مشکل مواجه خواهید شدRuby . راه های مختلفی برای رفع این نوع مشکلات ارائه می دهد. در این بخش با مطالب زیر آشنا خواهید شد :

- دستور `if`
- دستور `if…else`
- عملگر سه تایی
- دستور `if` چندگانه
- دستور `if` تو در تو
- عملگرهای منطقی

دستور if

می‌توان با استفاده از دستور if و یک شرط خاص که باعث ایجاد یک کد می‌شود یک منطق به برنامه خود اضافه کنید. دستور if ساده‌ترین دستور شرطی است که برنامه می‌گوید اگر شرطی برقرار است کد معینی را انجام بده. ساختار دستور if به صورت زیر است :

```
if condition
   code to execute
end
```

قبل از اجرای دستور if ابتدا شرط بررسی می‌شود. اگر شرط برقرار باشد یعنی درست باشد سپس کد اجرا می‌شود. شرط یک عبارت مقایسه‌ای است. می‌توان از عملگرهای مقایسه‌ای برای تست درست یا اشتباه بودن شرط استفاده کرد. اجازه بدهید که نگاهی به نحوه استفاده از دستور if در داخل برنامه بیندازیم. برنامه زیر پیغام Hello World را اگر مقدار number کمتر از ۱۰ و Goodbye World را اگر مقدار number از ۱۰ بزرگ‌تر باشد در صفحه نمایش می‌دهد.

```
1  #Declare a variable and set it a value less than 10
2  number = 5
3
4  #If the value of number is less than 10
5  if number < 10
6    puts 'Hello World.'
7  end
8
9  #Change the value of a number to a value which is greater than 10
10 number = 15
11
12 #If the value of number is greater than 10
13 if number > 10
14   puts 'Goodbye World.'
15 end
```

```
Hello World.
Goodbye World.
```

در خط ۲ یک متغیر با نام number تعریف و مقدار ۵ به آن اختصاص داده شده است. وقتی به اولین دستور if در خط ۵ می‌رسیم برنامه تشخیص می‌دهد که مقدار number از ۱۰ کمتر است یعنی ۵ کوچک‌تر از ۱۰ است.

منطقی است که نتیجه مقایسه درست می‌باشد، بنابراین دستور if دستور را اجرا می‌کند (خط ۶) و پیغام Hello World چاپ می‌شود. حال مقدار number را به ۱۵ تغییر می‌دهیم (خط ۱۰). وقتی به دومین دستور if در خط ۱۳ می‌رسیم برنامه مقدار number را با ۱۰ مقایسه می‌کند و چون مقدار number یعنی ۱۵ از ۱۰ بزرگ‌تر است برنامه پیغام Goodbye World را چاپ می‌کند (خط ۱۴). به این نکته توجه کنید که دستور if را می‌توان در یک خط نوشت :

```
if number > 10; puts "Goodbye World." end
```

همانطور که در کد بالا مشاهده می کنید، باید بعد از شرط یک علامت سمیکالن قرار دهید. شما می‌توانید چندین دستور را در داخل دستور if بنویسید. کافیست همه آنها را قبل از کلمه end بنویسید. نحوه تعریف چند دستور در داخل بدنه if به صورت زیر است :

```
if condition
    statement1
    statement2
    .
    .
    .
    statementN
end
```

این هم یک مثال ساده :

```
x = 15;

if x > 10
   puts "x is greater than 10."
   puts "This is still part of the if statement."
end
```

در مثال بالا اگر مقدار x از ۱۰ بزرگ‌تر باشد دو پیغام چاپ می‌شود. مثالی دیگر در مورد دستور : if

```
print "Enter a number: "
firstNumber = gets.to_i

print "Enter another number: "
secondNumber = gets.to_i

if firstNumber == secondNumber
    puts "#{firstNumber} == #{secondNumber}"
end

if firstNumber != secondNumber
    puts "#{firstNumber} != #{secondNumber}"
end

if firstNumber < secondNumber
    puts "#{firstNumber} < #{secondNumber}"
end

if firstNumber > secondNumber
    puts "#{firstNumber} > #{secondNumber}"
end

if firstNumber <= secondNumber
    puts "#{firstNumber} <= #{secondNumber}"
end

if firstNumber >= secondNumber
    puts "#{firstNumber} >= #{secondNumber}"
end
```

```
Enter a number: 2
Enter another number: 5
2 != 5
2 < 5
2 <= 5
Enter a number: 10
Enter another number: 3
10 != 3
10 > 3
10 >= 3
```

```
Enter a number: 5
Enter another number: 5
5 == 5
5 <= 5
5 >= 5
```

ما از عملگرهای مقایسه‌ای در دستور `if` استفاده کرده‌ایم. ابتدا دو عدد که قرار است با هم مقایسه شوند را به عنوان ورودی از کاربر می‌گیریم. اعداد با هم مقایسه می‌شوند و اگر شرط درست بود پیغامی چاپ می‌شود. به این نکته توجه داشته باشید که شرطها مقادیر بولی هستند، یعنی دارای دو مقدار `true` یا `flase` می باشند.

دستور if...else

دستور if فقط برای اجرای یک حالت خاص به کار می‌رود یعنی اگر حالتی برقرار بود کار خاصی انجام شود. اما زمانی که شما بخواهید اگر شرط خاصی برقرار شد یک دستور و اگر برقرار نبود دستور دیگر اجرا شود باید از دستور if else استفاده کنید. ساختار دستور if else در زیر آمده است :

```
if (condition)
    statements1
else
    statements2
end
```

از کلمه کلیدی else نمی‌توان به تنهایی استفاده کرد بلکه حتماً باید با if به کار برده شود. کد داخل بلوک else فقط در صورتی اجرا می‌شود که شرط داخل دستور if نادرست باشد. در زیر نحوه استفاده از دستور if...else آمده است.

```
1  number = 5
2
3  #Test the condition
4  if number < 10
5    puts 'The number is less than 10.'
6  else
7    puts 'The number is either greater than or equal to 10.'
8  end
9
10 #Modify value of number
11 number = 15
12
13 #Repeat the test to yield a different result
14 if number < 10
15   puts 'The number is less than 10.'
16 else
17   puts 'The number is either greater than or equal to 10.'
18 end
```

```
The number is less than 10.
The number is either greater than or equal to 10.
```

در خط ۱ یک متغیر به نام number تعریف کرده‌ایم و در خط ۴ تست می‌کنیم که آیا مقدار متغیر number از ۱۰ کمتر است یا نه و چون کمتر است در نتیجه کد داخل بلوک if اجرا می‌شود (خط ۵) و اگر مقدار number را تغییر دهیم و به مقداری بزرگتر از ۱۰ تغییر دهیم (خط ۱۱)، شرط نادرست می‌شود (خط ۱۴) و کد داخل بلوک else اجرا می‌شود (خط ۱۷).

دستور if...elsif...else

اگر بخواهید چند شرط را بررسی کنید چکار می‌کنید؟ می‌توانید از چندین دستور if استفاده کنید و بهتر است که این دستورات if را به صورت زیر بنویسید :

```
if (condition)
  code to execute
else
  if (condition)
        code to execute
  else
        if (condition)
            code to execute
        else
          code to execute
        end
  end
end
```

خواندن کد بالا سخت است. بهتر است دستورات را به صورت تو رفتگی در داخل بلوک else بنویسید. می‌توانید کد بالا را ساده‌تر کنید :

```
if (condition1)
    code to execute1
elsif (condition2)
    code to execute2
elsif (condition n)
    code to execute3
else
    code to execute4
end
```

حال که نحوه استفاده از دستور if else را یاد گرفتید باید بدانید که مانند elif، else نیز به دستور if وابسته است. دستور elif وقتی اجرا می‌شود که اولین دستور if اشتباه باشد حال اگر elif اشتباه باشد دستور elif بعدی اجرا می‌شود. و اگر آن نیز اجرا نشود در نهایت دستور else اجرا می‌شود. برنامه زیر نحوه استفاده از دستورelif را نشان می‌دهد :

```
1  puts 'What\'s your favorite color?'
2  puts '[1] Black'
3  puts '[2] White'
4  puts '[3] Blue'
5  puts '[4] Red'
6  puts '[5] Yellown'
7
8  print 'Enter your choice: '
9  choice = gets.to_i
10
11 if choice == 1
12   puts 'You might like my black t-shirt.'
13 elsif choice == 2
14   puts 'You might be a clean and tidy person.'
15 elsif choice == 3
16   puts 'You might be sad today.'
17 elsif choice == 4
18   puts 'You might be inlove right now.'
19 elsif choice == 5
20   puts 'Lemon might be your favorite fruit.'
```

```
21  else
22    puts 'Sorry, your favorite color is not in the choices above.'
23  end
```

```
What's your favorite color?
[1] Black
[2] White
[3] Blue
[4] Red
[5] Yellow

Enter your choice: 1
You might like my black t-shirt.
What's your favorite color?
[1] Black
[2] White
[3] Blue
[4] Red
[5] Yellow

Enter your choice: 999
Sorry, your favorite color is not in the choices above.
```

خروجی برنامه بالا به متغیر choice وابسته است. بسته به اینکه شما چه چیزی انتخاب می‌کنید پیغامهای مختلفی چاپ می‌شود. اگر عددی که شما تایپ می‌کنید در داخل حالتهای انتخاب نباشد کد مربوط به بلوک else اجرا می‌شود.

دستور if تو در تو

می‌توان از دستور if تو در تو در Ruby استفاده کرد. یک دستور ساده if در داخل دستور if دیگر.

```
if (condition)
  code to execute
  if (condition)
      code to execute
  elsif (condition)
      if (condition)
          code to execute
      else
          if (condition)
              code to execute
          end
      end
  end
end
```

اجازه بدهید که نحوه استفاده از دستور if تو در تو را نشان دهیم :

```
1  print 'Enter your age: '
2  age = gets.to_i
3
4  print 'Enter your gender (male/female)'
5  gender = gets
6
7  if age > 12
8      if age < 20
9          if gender == "male"
10             puts "You are a teenage boy."
11         else
12             puts "You are a teenage girl."
13         end
14     else
15         puts "You are already an adult."
16     end
17 else
18     puts "You are still too young."
19 end
```

```
Enter your age: 18
Enter your gender: male
You are a teenage boy.
Enter your age: 12
Enter your gender: female
You are still too young.
```

اجازه بدهید که برنامه را کالبد شکافی کنیم. ابتدا برنامه از شما درباره سنتان سؤال می‌کند (خط ۱). در خط ۴ درباره جنستان از شما سؤال می‌کند. سپس به اولین دستور if می‌رسد (خط ۷). در این قسمت اگر سن شما بیشتر از ۱۲ سال باشد برنامه وارد بدنه دستور if می‌شود در غیر اینصورت وارد بلوک else (خط ۱۷) مربوط به همین دستور if می‌شود.

حال فرض کنیم که سن شما بیشتر از ۱۲ سال است و شما وارد بدنه اولین if شده‌اید. در بدنه اولین if دو دستور if دیگر را مشاهده می‌کنید. اگر سن کمتر ۲۰ باشد شما وارد بدنه if دوم می‌شوید و اگر نباشد به قسمت else متناظر با آن می‌روید (خط ۱۴). دوباره فرض می‌کنیم که سن

شما کمتر از ۲۰ باشد، در اینصورت وارد بدنه `if` دوم شده و با یک `if` دیگر مواجه می‌شوید (خط ۹) در اینجا جنسیت شما مورد بررسی قرار می‌گیرد که اگر برابر "`male`" باشد، کدهای داخل بدنه سومین `if` اجرا می‌شود در غیر اینصورت قسمت `else` مربوط به این `if` اجرا می‌شود (خط ۱۱). پیشنهاد می‌شود که از `if` تو در تو در برنامه کمتر استفاده کنید چون خوانایی برنامه را پایین می‌آورد.

استفاده از عملگرهای منطقی

عملگرهای منطقی به شما اجازه می‌دهند که چندین شرط را با هم ترکیب کنید. این عملگرها حداقل دو شرط را در گیر می‌کنند و در آخر یک مقدار بولی را بر می‌گردانند. در جدول زیر برخی از عملگرهای منطقی آمده است :

عملگر	مثال	تأثیر
and	z = (x > 2) and (y < 10)	مقدار Z در صورتی true است که هر دو شرط دو طرف عملگر مقدارشان true باشد. اگر فقط مقدار یکی از شروط false باشد مقدار z، false خواهد شد.
or	z = (x > 2) or (y < 10)	مقدار Z در صورتی true است که یکی از دو شرط دو طرف عملگر مقدارشان true باشد. اگر هر دو شرط مقدارشان false باشد مقدار z، false خواهد شد.
not	z = not(x > 2)	مقدار Z در صورتی true است که مقدار شرط false باشد و در صورتی false است که مقدار شرط true باشد.

به عنوان مثال جمله z = (x > 2) and (y < 10) را به این صورت بخوانید: "در صورتی مقدار z برابر true است که مقدار x بزرگ‌تر از 2 و مقدار y کوچک‌تر از ۱۰ باشد در غیر اینصورت false است". این جمله بدین معناست که برای اینکه مقدار کل دستور true باشد باید مقدار همه شروط true باشد. عملگر منطقی or تأثیر متفاوتی نسبت به عملگر منطقی and دارد. نتیجه عملگر منطقی or برابر true است اگر فقط مقدار یکی از شروط true باشد. و اگر مقدار هیچ یک از شروط true نباشد نتیجه false خواهد شد. می‌توان عملگرهای منطقی and و or را با هم ترکیب کرده و در یک عبارت به کار برد مانند :

```
if  (x == 1) and ((y > 3) or z < 10)
   #do something here
end
```

در اینجا استفاده از پرانتز مهم است چون از آن در گروه بندی شرطها استفاده می‌کنیم. در اینجا ابتدا عبارت (y > 3) or (z < 10) مورد بررسی قرار می‌گیرد (به علت تقدم عملگرها). سپس نتیجه آن بوسیله عملگر and با نتیجه (x == 1) مقایسه می‌شود. حال بیایید نحوه استفاده از عملگرهای منطقی در برنامه را مورد بررسی قرار دهیم :

```
print "Enter your age: "
age = gets.to_i

print "Enter your gender male/female: "
gender = gets

if age > 12 and age < 20
  if gender == "male"
    puts "You are a teenage boy."
  else
    puts "You are a teenage girl."
  end
else
    puts "You are not a teenager."
```

```
15  end
```

```
Enter your age: 18
Enter your gender (male/female): female
You are a teenage girl.
Enter you age: 10
Enter your gender (male/female): male
You are not a teenager.
```

برنامه بالا نحوه استفاده از عملگر منطقی and را نشان می‌دهد (خط ۷). وقتی به دستور if می‌رسید (خط ۷) برنامه سن شما را چک می‌کند. اگر سن شما بزرگتر از ۱۲ و کوچکتر از ۲۰ باشد (سنتان بین ۱۲ و ۲۰ باشد) یعنی مقدار هر دو true باشد سپس کدهای داخل بلوک if اجرا می‌شوند. اگر نتیجه یکی از شروط false باشد کدهای داخل بلوک else اجرا می‌شود. عملگر and عملوند سمت چپ را مورد بررسی قرار می‌دهد. اگر مقدار آن false باشد دیگر عملوند سمت راست را بررسی نمی‌کند و مقدار false را بر می‌گرداند. بر عکس عملگر or عملوند سمت چپ را مورد بررسی قرار می‌دهد و اگر مقدار آن true باشد سپس عملوند سمت راست را نادیده می‌گیرد و مقدار true را بر می‌گرداند.

```
if x == 2 and y == 3
   #Some code here
end

if x == 2 or y == 3
   #Some code here
end
```

بین عملگرهای منطقی و بیتی دو تفاوت وجود دارد. عملگر منطقی همیشه ۱ (برای درست) یا ۰ (برای غلط) را برمی گرداند. علاوه بر این، یک عملگر منطقی از منطق "اتصال کوتاه" برای محاسبه استفاده می کند، به این معنی که اگر نتیجه، پس از بررسی شرط اول مشخص شود، شرط دوم نادیده گرفته می شود. به عنوان مثال، هنگام استفاده از عملگر منطقی AND، می دانیم که هر دو شرط باید صحیح باشند تا نتیجه true باشد. اگر اولین عملوند false باشد، نتیجه false خواهد شد و در نتیجه شرط دوم مورد بررسی قرار نمی گیرد. اگر شرطها را در برنامه ترکیب کنید استفاده از عملگرهای منطقی and و or به جای عملگرهای بیتی and و or بهتر خواهد بود. یکی دیگر از عملگرهای منطقی عملگر not است که نتیجه یک عبارت را خنثی یا منفی می‌کند. به مثال زیر توجه کنید:

```
if not(x == 2)
   puts "x is not equal to 2."
end
```

اگر نتیجه عبارت x == 2 برابر false باشد عملگر not آن را true می‌کند.

دستور case

در Ruby ساختاری به نام case وجود دارد که به شما اجازه می‌دهد که با توجه به مقدار ثابت یک متغیر چندین انتخاب داشته باشید. دستور case معادل دستور if…elsif است با این تفاوت که در دستور case متغیر فقط مقادیر ثابتی از اعداد، رشته‌ها و یا کاراکترها را قبول می‌کند. مقادیر ثابت مقادیری هستند که قابل تغییر نیستند. در زیر نحوه استفاده از دستور case آمده است :

```
case (testVar)
  when expression1
      statements1
  when expression2
      statements2
  else
      statements3
end
```

ابتدا یک مقدار در متغیر case که در مثال بالا testVar است قرار می‌دهید. این مقدار با هر یک از عبارتهای when داخل بلوک case مقایسه می‌شود. اگر مقدار متغیر با هر یک از مقادیر موجود در دستورات when برابر بود کد مربوط به آن when اجرا خواهد شد. به این نکته توجه کنید که می توان در داخل دستور when تعداد دستورات بیشتری نوشت. دستور case یک بخش else دارد. این دستور در صورتی اجرا می‌شود که مقدار متغیر با هیچ یک از مقادیر دستورات when برابر نباشد. دستور else اختیاری است و اگر از بدنه case حذف شود هیچ اتفاقی نمی‌افتد. این دستور باید در پایان دستورات when نوشته شود. به مثالی در مورد دستور case توجه کنید :

```
puts "What's your favorite pet?"
puts "[1] Dog"
puts "[2] Cat"
puts "[3] Rabbit"
puts "[4] Turtle"
puts "[5] Fish"
puts "[6] Not in the choices\n\n"

print "Enter your choice: "
choice = gets.to_i

case choice
  when 1
      puts "Your favorite pet is Dog."
  when 2
      puts "Your favorite pet is Cat."
  when 3
      puts "Your favorite pet is Rabbit."
  when 4
      puts "Your favorite pet is Turtle."
  when 5
      puts "Your favorite pet is Fish."
  when 6
      puts "Your favorite pet is not in the choices."
  else
      puts "You don't have a favorite pet."
end
```

```
What's your favorite pet?
[1] Dog
[2] Cat
[3] Rabbit
```

```
[4] Turtle
[5] Fish
[6] Not in the choices

Enter your choice: 2
Your favorite pet is Cat.
What's your favorite pet?
[1] Dog
[2] Cat
[3] Rabbit
[4] Turtle
[5] Fish
[6] Not in the choices

Enter your choice: 99
You don't have a favorite pet.
```

برنامه بالا به شما اجازه انتخاب حیوان مورد علاقه‌تان را می‌دهد. به اسم هر حیوان یک عدد نسبت داده شده است. شما عدد را وارد می‌کنید و این عدد در دستور case با مقادیر when مقایسه می‌شود و با هر کدام از آن مقادیر که برابر بود پیغام مناسب نمایش داده خواهد شد. اگر هم با هیچ کدام از مقادیر when ها برابر نبود دستور else اجرا می‌شود. یکی دیگر از ویژگیهای دستور case این است که شما می‌توانید مقدار مورد نظر خود را با از دو یا چند مقدار در داخل یک when مقایسه کنید. در مثال زیر اگر مقدار number عدد ۱، ۲ یا ۳ باشد یک کد اجرا می‌شود:

```
case number
  when 1, 2, 3
    puts "This code is shared by three values."
end
```

همانطور که قبلاً ذکر شد دستور case معادل دستور if…elsif است. برنامه بالا را به صورت زیر نیز می‌توان نوشت :

```
if choice == 1
  puts "Your favorite pet is Dog."
elsif choice == 2
  puts "Your favorite pet is Cat."
elsif choice == 3
  puts "Your favorite pet is Rabbit."
elsif choice == 4
  puts "Your favorite pet is Turtle."
elsif choice == 5
  puts "Your favorite pet is Fish."
elsif choice == 6
  puts "Your favorite pet is not in the choices."
else
  puts "You don't have a favorite pet."
end
```

کد بالا دقیقاً نتیجه‌ای مانند دستور case دارد. دستور else معادل دستور else می‌باشد. حال از بین این دو دستور (case و if else) کدامیک را انتخاب کنیم. از دستور case موقعی استفاده می‌کنیم که مقداری که می‌خواهیم با دیگر مقادیر مقایسه شود ثابت باشد.

عملگر شرطی

عملگر شرطی در Ruby مانند دستور شرطی if…else عمل می‌کند. در زیر نحوه استفاده از این عملگر آمده است:

```
condition ? true : false
```

عملگر شرطی تنها عملگر سه تایی Ruby است که نیاز به سه عملوند دارد، یک مقدار زمانی که شرط درست باشد، شرط و یک مقدار زمانی که شرط نادرست باشد. اجازه بدهید که نحوه استفاده این عملگر را در داخل برنامه مورد بررسی قرار دهیم.

```
pet1 = 'puppy'
pet2 = 'kitten'

type1 = (pet1 == 'puppy' ) ? 'dog' : 'cat'
type2 = (pet2 == 'kitten') ? 'cat' : 'dog'

puts type1
puts type2
```

```
dog
cat
```

برنامه بالا نحوه استفاده از این عملگر شرطی را نشان می‌دهد. خط ۴ به این صورت ترجمه می‌شود که مقدار dog را در متغیر type1 قرار بده اگر مقدار pet1 برابر با puppy بود در غیر این صورت مقدار cat را type1 قرار بده. خط ۵ به این صورت ترجمه می‌شود که مقدار cat را در type2 قرار بده اگر مقدار pet2 برابر با kitten بود در غیر این صورت مقدار dog. حال برنامه بالا را با استفاده از دستور if else می‌نویسیم:

```
if (pet1 == "puppy")
   type1 = "dog"
else
   type1 = "cat"
end
```

هنگامی که چندین دستور در داخل یک بلوک if یا else دارید از عملگر شرطی استفاده نکنید چون خوانایی برنامه را پایین می‌آورد.

تکرار

ساختارهای تکرار به شما اجازه می‌دهند که یک یا چند دستور کد را تا زمانی که یک شرط برقرار است تکرار کنید. بدون ساختارهای تکرار شما مجبورید همان تعداد کدها را بنویسید که بسیار خسته کننده است. مثلاً شما مجبورید ۱۰ بار جمله "Hello World." را تایپ کنید مانند مثال زیر :

```
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
puts "Hello World!"
```

البته شما می‌توانید با کپی کردن این تعداد کد را راحت بنویسید ولی این کار در کل کیفیت کدنویسی را پایین می‌آورد. راه بهتر برای نوشتن کدهای بالا استفاده از حلقه‌ها است. حلقه‌ها در Ruby عبارتند از :

- while
- for in
- until
- each

حلقه While

ابتدایی‌ترین ساختار تکرار در Ruby حلقه While است. ابتدا یک شرط را مورد بررسی قرار می‌دهد و تا زمانیکه شرط برقرار باشد کدهای درون بلوک اجرا می‌شوند. ساختار حلقه While به صورت زیر است :

```
while condition
    #code to loop
end
```

می‌بینید که ساختار While مانند ساختار if بسیار ساده است. ابتدا یک شرط را که نتیجه آن یک مقدار بولی است می‌نویسیم اگر نتیجه درست یا true باشد سپس کدهای داخل بلوک While اجرا می‌شوند. اگر شرط غلط یا false باشد وقتی که برنامه به حلقه While برسد هیچکدام از کدها را اجرا نمی‌کند. برای متوقف شدن حلقه باید مقادیر داخل حلقه While اصلاح شوند.

به یک متغیر شمارنده در داخل بدنه حلقه نیاز داریم. این شمارنده برای آزمایش شرط مورد استفاده قرار می‌گیرد و ادامه یا توقف حلقه به نوعی به آن وابسته است. این شمارنده را در داخل بدنه باید کاهش یا افزایش دهیم. در برنامه زیر نحوه استفاده از حلقه While آمده است :

```
counter = 1

while counter <= 10
  puts "Hello World!"
  counter = counter + 1
end
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

برنامه بالا ۱۰ بار پیغام Hello World! را چاپ می‌کند. اگر از حلقه در مثال بالا استفاده نمی‌کردیم مجبور بودیم تمام ۱۰ خط را تایپ کنیم. اجازه دهید که نگاهی به کدهای برنامه فوق بیندازیم. ابتدا در خط ۱ یک متغیر تعریف و از آن به عنوان شمارنده حلقه استفاده شده است. سپس به آن مقدار ۱ را اختصاص می‌دهیم چون اگر مقدار نداشته باشد نمی‌توان در شرط از آن استفاده کرد.

در خط ۳ حلقه while را وارد می‌کنیم. در حلقه while ابتدا مقدار اولیه شمارنده با ۱۰ مقایسه می‌شود که آیا از ۱۰ کمتر است یا با آن برابر است. نتیجه هر بار مقایسه ورود به بدنه حلقه While و چاپ پیغام است. همانطور که مشاهده می‌کنید بعد از هر بار مقایسه مقدار شمارنده یک واحد اضافه می‌شود (خط ۵). حلقه تا زمانی تکرار می‌شود که مقدار شمارنده از ۱۰ کمتر باشد.

اگر مقدار شمارنده یک بماند و آن را افزایش ندهیم و یا مقدار شرط هرگز false نشود یک حلقه بینهایت به وجود می‌آید. به این نکته توجه کنید که در شرط بالا به جای علامت > از => استفاده شده است. اگر از علامت > استفاده می‌کردیم کد ما ۹ بار تکرار می‌شد چون مقدار اولیه ۱ است و

هنگامی که شرط به ۱۰ برسد `false` می‌شود چون `10 < 10` نیست. اگر می‌خواهید یک حلقه بی نهایت ایجاد کنید که هیچگاه متوقف نشود باید یک شرط ایجاد کنید که همواره درست (`true`) باشد.

```
while true
   #code to loop
end
```

این تکنیک در برخی موارد کارایی دارد و آن زمانی است که شما بخواهید با استفاده از دستورات `break` و `return` که در آینده توضیح خواهیم داد از حلقه خارج شوید.

حلقه for…in

یکی دیگر از ساختارهای تکرار حلقه for…in است. این حلقه عملی شبیه به حلقه while انجام می‌دهد. ساختار حلقه for…in به صورت زیر است :

```
for iterator_var in sequence
   #code to repeat
end
```

iterator_var یک متغیر موقتی، in کلمه کلیدی و sequence هم می تواند یک محدوده، آرایه و ... باشد. می توان حلقه for را اینگونه ترجمه کرد، که به ازای یا به تعداد آیتم های موجود در سری، فلان کارها یا کدها را تکرار کن. در زیر یک مثال از حلقه for…in آمده است:

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    puts "Number #{i}"
end
```

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
Number 10
```

برنامه بالا اعداد ۱ تا ۱۰ را با استفاده از حلقه for…in می‌شمارد. ابتدا یک متغیر موقتی (i)، سپس کلمه کلیدی in و در آخر یک سری از اعداد که در اینجا یک آرایه می باشد، تعریف می‌کنیم. کد اجرا می‌شود. هر بار که حلقه اجرا می شود، ابتدا یکی از آیتم های آرایه در متغیر i قرار گرفته و در خط بعد چاپ می شود. این کار تا چاپ آخرین آیتم ادامه می یابد. به جای آرایه در کد بالا می توانید از محدوده هم استفاده کنید :

```
for i in 1..10

end
```

دستور until

این دستور شباهت زیادی به دستور while دارد با این تفاوت که عبارت شرطی آن باید برابر با false باشد تا اجرا گردد. ساختار دستور until به صورت زیر است:

```
until (condition)
    #statements
end
```

در دستور بالا عبارت statements زمانی اجرا می گردد که مقدار عبارت شرطی condition برابر با false باشد. به مثال زیر توجه کنید:

```
counter = 1

until counter > 10
  puts "Hello World!"
  counter = counter + 1
end
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

در کد بالا و در خط ۱، یک متغیر تعریف کرده ایم و مقدار ان را برابر ۱ گذاشته ایم. در خط ۳ چک می شود که آیا مقدار متغیر ما از ۱۰ بزرگتر است یا نه؟ چون مقدار متغیر ما از ۱۰ بزرگتر نیست در نتیجه شرط ما false شده و خطوط ۴ و ۵ اجرا می شوند. دستور until را به صورت معکوس هم می توان نوشت. در این حالت، قسمت شرط حلقه، در انتهای حلقه بررسی می شود، بنابراین، این حلقه حداقل یکبار اجرا می گردد. ساختار until معکوس به صورت زیر است:

```
code
until conditional
```

یا

```
begin
  #Code to execute
end until condition
```

به مثال زیر توجه کنید:

```
number = 1
begin
  puts "Hello World!"
  number += 1
end until number > 10
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

همانطور که در کد بالا مشاهده می کنید، ما در خط آخر شرط حلقه را بررسی می کنیم و این باعث می شود که حداقل یک بار دستورات بین دو کلمه begin و end اجرا شوند.

دستور each

حلقه each یکی دیگر از حلقه های تکرار در Ruby می باشد. ساختار این حلقه به صورت زیر است:

```
Iterable.each do |variable|
    #Code to execute
end
```

Iterable در کد بالا می تواند یک آرایه یا محدوده باشد. variable هم یک متغیر موقتی است که عناصر محدوده و یا آرایه در آن ریخته می شوند و با هر بار اجرای حلقه یکی از آنها خوانده می شود. به مثال زیر توجه کنید:

```
myArray = [1, 2, 3, 4, 5]

myArray.each do |number|
  puts "#{number}"
end
```

```
1
2
3
4
5
```

می توان از یک محدوده هم به صورت زیر استفاده کرد:

```
(1..5).each do |number|
  puts "#{number}"
end
```

```
1
2
3
4
5
```

خارج شدن از حلقه با استفاده از break و next

گاهی اوقات با وجود درست بودن شرط می‌خواهیم حلقه متوقف شود. سؤال اینجاست که چطور این کار را انجام دهید؟ با استفاده از کلمه کلیدی break حلقه را متوقف کرده و با استفاده از کلمه کلیدی next می‌توان بخشی از حلقه را رد کرد و به مرحله بعد رفت. برنامه زیر نحوه استفاده از next و break را نشان می‌دهد :

```
puts "Demonstrating the use of break\n"

for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  if x == 5
      break
  end
  puts "Number #{x}"
end


puts "\nDemonstrating the use of continue\n"

for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  if x == 5
    next
  end
  puts "Number #{x}"
end
```

```
Demonstrating the use of break

Number  1
Number  2
Number  3
Number  4

Demonstrating the use of continue

Number  1
Number  2
Number  3
Number  4
Number  6
Number  7
Number  8
Number  9
Number  10
```

در این برنامه از حلقه for برای نشان دادن کاربرد دو کلمه کلیدی فوق استفاده شده است اگر به جای for از حلقه while استفاده می‌شد نتیجه یکسانی به دست می‌آمد. همانطور که در شرط برنامه (خط ۴) آمده است، وقتی که مقدار x به عدد ۵ برسد، سپس دستور break اجرا (خط ۵) و حلقه بلافاصله متوقف می‌شود، حتی اگر شرط x < 10 برقرار باشد. از طرف دیگر در خط ۱۴ حلقه for فقط برای یک تکرار خاص متوقف شده و سپس ادامه می‌یابد. (وقتی مقدار x برابر ۵ شود حلقه از ۵ رد شده و مقدار ۵ را چاپ نمی‌کند و بقیه مقادیر چاپ می‌شوند).

آرایه

آرایه نوعی متغیر است که لیستی از آدرسهای مجموعه ای از داده های همنوع یا غیر همنوع را در خود ذخیره می کند. تعریف چندین متغیر از یک نوع برای هدفی یکسان بسیار خسته کننده است. مثلا اگر بخواهید صد متغیر از نوع اعداد صحیح تعریف کرده و از آنها استفاده کنید. مطمئنا تعریف این همه متغیر بسیار کسالت آور و خسته کننده است. اما با استفاده از آرایه می توان همه آنها را در یک خط تعریف کرد. در زیر راهی ساده برای تعریف یک آرایه نشان داده شده است :

```
numbers = Array.new
```

numbers نام آرایه را نشان می دهد. هنگام نامگذاری آرایه بهتر است که نام آرایه نشان دهنده نوع آرایه باشد. به عنوان مثال برای نامگذاری آرایه ای که اعداد را در خود ذخیره می کند از کلمه number استفاده کنید. حتی می توانیم به هنگام ایجاد آرایه مقادیر خانه های آن را نیز مشخص کنیم :

```
numbers = Array [1, 2, 3, 4, 5]
```

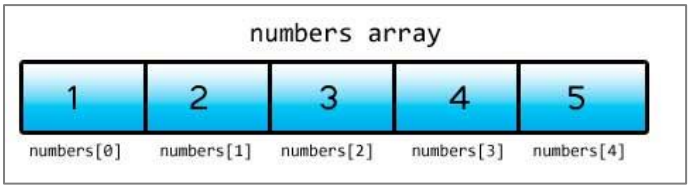
یا

```
numbers = Array.[] (1, 2, 3, 4, 5)
```

در این مثال ۵ مقدار در ۵ آدرس از فضای حافظه کامپیوتر شما ذخیره می شود. مثالا بالا را به روش زیر هم می توان پیاده سازی کرد :

```
numbers = []

numbers[0] = 1
numbers[1] = 2
numbers[2] = 3
numbers[3] = 4
numbers[4] = 5
```

اندیس یک آرایه از صفر شروع شده و به یک واحد کمتر از طول آرایه ختم می شود.به عنوان مثال شما یک آرایه ۵ عضوی دارید، اندیس آرایه از ۰ تا ۴ می باشد چون طول آرایه ۵ است پس ۱-۵ برابر است با ۴. این بدان معناست که اندیس ۰ نشان دهنده اولین عضو آرایه است و اندیس ۱ نشان دهنده دومین عضو و الی آخر. برای درک بهتر مثال بالا به شکل زیر توجه کنید :



به هر یک از اجزاء آرایه و اندیسهای داخل کروشه توجه کنید.کسانی که تازه شروع به برنامه نویسی کرده اند معمولا در گذاشتن اندیس دچار اشتباه می شوند و مثلا ممکن است در مثال بالا اندیسها را از ۱ شروع کنند. در مثال های بالا، ما یک آرایه با طول نا مشخص تعریف کردیم. یعنی

مشخص نکردیم که چه تعداد عنصر قرار است در آرایه قرار بگیرند. اگر بخواهیم که تعداد عناصر را هم مشخص کنیم می توانیم تعداد آنها را در داخل پرانتز بنویسیم:

```
numbers = Array.new(5)
```

ساده ترین روش تعریف آرایه، به صورت زیر است:

```
numbers = [1, 2, 3, 4, 5]
```

در ادامه این درس و درس های بعدی از این روش برای تعریف آرایه استفاده می کنیم.

دستیابی به مقادیر آرایه با استفاده از حلقه for

در زیر مثالی در مورد استفاده از آرایه‌ها آمده است. در این برنامه ۵ مقدار از کاربر گرفته شده و میانگین آنها حساب می‌شود:

```
numbers = Array.new(5)
total   = 0
average = 0.0

i = 0
while i < numbers.length
  print "Enter a number: "
  num = gets.to_i
  numbers[i] = num
  i+=1
end

j = 0
while j < numbers.length
  total += numbers[j]
  j+=1
end

average = total / numbers.length
puts "Average = #{average}"
```

```
Number 0: 95
Number 1: 85
Number 2: 80
Number 3: 87
Number 4: 92
Average =  87
```

در خط ۱ یک آرایه تعریف شده است که می‌تواند ۵ آیتم را در خود ذخیره کند. خطوط ۲ و ۳ متغیرهایی تعریف شده‌اند که از آنها برای محاسبه میانگین استفاده می‌شود. توجه کنید که مقدار اولیه `total` صفر است تا از بروز خطا هنگام اضافه شدن مقدار به آن جلوگیری شود. در خطوط ۵ تا ۱۱ حلقه `while` برای تکرار و گرفتن ورودی از کاربر تعریف شده است. از خاصیت طول (`length`) آرایه برای تشخیص تعداد اجزای آرایه استفاده می‌شود. اگر چه می‌توانستیم به سادگی در حلقه `while` مقدار ۵ را برای شرط قرار دهیم ولی استفاده از خاصیت طول آرایه، کار راحت‌تری است و می‌توانیم طول آرایه را تغییر دهیم و شرط حلقه `while` با تغییر جدید هماهنگ شود. در خط ۸ ورودی دریافت شده از کاربر به نوع صحیح تبدیل و در آرایه ذخیره می‌شود. اندیس استفاده شده در `number` (خط ۹) مقدار `i` جاری در حلقه است. برای مثال در ابتدای حلقه مقدار `i` صفر است

بنابراین وقتی در خط ۹ اولین داده از کاربر گرفته می‌شود اندیس آن برابر ۰ می‌شود. در تکرار بعدی `i` یک واحد اضافه می‌شود و در نتیجه در خط ۹ و بعد از ورود دومین داده توسط کاربر اندیس آن برابر ۱ می‌شود. این حالت تا زمانی که شرط در حلقه `while` برقرار است ادامه می‌یابد.

در خطوط ۱۷-۱۳ از حلقه `while` دیگر برای دسترسی به مقدار هر یک از داده‌های آرایه استفاده شده است. در این حلقه نیز مانند حلقه قبل از مقدار متغیر شمارنده به عنوان اندیس استفاده می‌کنیم. هر یک از اجزای عددی آرایه به متغیر `total` اضافه می‌شوند. بعد از پایان حلقه می‌توانیم میانگین اعداد را حساب کنیم (خط ۱۵). مقدار `total` را بر تعداد اجزای آرایه (تعداد عددها) تقسیم می‌کنیم (خط ۱۹). برای دسترسی به تعداد اجزای آرایه می‌توان از خاصیت `length` آرایه استفاده کرد. خط ۲۰ مقدار میانگین را در صفحه نمایش چاپ می‌کند. خروجی برنامه بالا می تواند به صورت زیر باشد (البته ممکن است برای شما متفاوت باشد). آرایه‌ها در برخی شرایط بسیار پر کاربرد هستند و تسلط شما بر این مفهوم و اینکه چطور از آنها استفاده کنید بسیار مهم است.

آرایه های چند بعدی

آرایه‌های چند بعدی آرایه‌هایی هستند که برای دسترسی به هر یک از عناصر آنها باید از چندین اندیس استفاده کنیم. یک آرایه چند بعدی را می‌توان مانند یک جدول با تعدای ستون و ردیف تصور کنید. با افزایش اندیسها اندازه ابعاد آرایه نیز افزایش می‌یابد و آرایه‌های چند بعدی با بیش از دو اندیس به وجود می‌آیند. نحوه ایجاد یک آرایه با دو بعد به صورت زیر است :

```
array_name = [ [value1,value2,value3], [val1,val2,val3] ]
```

می توان گفت که یک آرایه دو بعدی، خود آرایه ای از آرایه هاست. یعنی هر عنصر این نوع آرایه، خود یک آرایه است. آرایه دو بعدی رو می توان به صورت یک جدول تصور کرد که دارای سطر و ستون می باشد. در یک آرایه دو بعدی برای دسترسی به هر یک از عناصر به دو مقدار نیاز داریم، یکی اندیس سطر و دیگری اندیس ستونی که آن عنصر در آن قرار دارد. یک مثال از آرایه دو بعدی در زیر آمده است :

```
numbers = [
            [ 1,  2,  3,  4,  5],
            [ 6,  7,  8,  9, 10],
            [11, 12, 13, 14, 15]
          ];
```

در کد بالا یک آرایه به نام numbers تعریف شده است که دارای سه عنصر است. البته هر کدام از این عناصر خود یک آرایه می باشند. یعنی آرایه numbers آرایه ای است از ۳ آرایه. هر کدام از این آرایه ها هم ۵ عنصر دارند. پس می توان گفت که آرایه numbers در واقع یک جدول با ۳ سطر و ۵ ستون می باشد. می‌توان مقدار دهی به عناصر را به صورت دستی انجام داد مانند :

```
numbers = [Array.new(5), Array.new(5), Array.new(5)]

numbers[0][0] =  1
numbers[0][1] =  2
numbers[0][2] =  3
numbers[0][3] =  4
numbers[0][4] =  5
numbers[1][0] =  6
numbers[1][1] =  7
numbers[1][2] =  8
numbers[1][3] =  9
numbers[1][4] = 10
numbers[2][0] = 11
numbers[2][1] = 12
numbers[2][2] = 13
numbers[2][3] = 14
numbers[2][4] = 15
```

همانطور که مشاهده می‌کنید برای دسترسی به هر یک از عناصر در یک آرایه دو بعدی به سادگی می‌توان از اندیسهای سطر و ستون و یک جفت کروشه مانند مثال استفاده کرد.

گردش در میان عناصر آرایه‌های چند بعدی

گردش در میان عناصر آرایه‌های چند بعدی نیاز به کمی دقت دارد. یکی از راههای آسان استفاده از حلقه for…in تو در تو است. به مثال زیر توجه کنید :

```ruby
nestedArray = [
    [1, 2, 3, 4, 5     ],
    [6, 7, 8, 9, 10    ],
    [11, 12, 13, 14, 15]
]

for array in nestedArray
  for number in array
    print "#{number} "
  end
  puts
end
```

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

گردش در میان مقادیر عناصر یک آرایه چند بعدی خیلی راحت است. حلقه for…in اول (خطوط ۱۲-۷) برای گردش در میان عناصر آرایه اصلی یعنی numbers و حلقه for…in دوم (خطوط ۱۰-۸) برای گردش در میان عناصر آرایه های عضو، به کار رفته است. یعنی حلقه اول ابتدا وارد سطر اول می شود و فورا حلقه دوم در میان ستون های این سطر گردش می کند و مقادیر داخل آنها را چاپ می کند، سپس حلقه for اول وارد دومین سطر می شود و . … بعد از اینکه دومین حلقه تکرار به پایان رسید، فوراً دستورات بعد از آن اجرا خواهند شد، که در اینجا دستور puts است که به برنامه اطلاع می‌دهد که یک خط جدید ایجاد کند. حال بیایید آنچه را از قبل یاد گرفته‌ایم در یک برنامه به کار بریم. این برنامه نمره چهار درس مربوط به سه دانش آموز را از ما می‌گیرد و معدل سه دانش آموز را حساب می‌کند.

```ruby
studentGrades = [
    Array.new(4),
    Array.new(4),
    Array.new(4)
]

student = 0
while student < studentGrades.length
  puts "Enter grades for Student #{student + 1}"

  total  = 0
  grade  = 0
  while grade < studentGrades[student].length
    print "Enter Grade ##{grade + 1}: "
    studentGrades[student][grade] = gets.to_f
    total += (studentGrades[student][grade]).to_f
    grade += 1
  end

  puts "Average is: #{(total / (studentGrades[student].length))}\n\n"
  student += 1
end
```

```
Enter grades for Student 1
Enter Grade #1: 92
Enter Grade #2: 87
Enter Grade #3: 89
Enter Grade #4: 95
Average is 90.75

Enter grades for Student 2
Enter Grade #1: 85
```

```
Enter Grade #2: 85
Enter Grade #3: 86
Enter Grade #4: 87
Average is 85.75

Enter grades for Student 3
Enter Grade #1: 90
Enter Grade #2: 90
Enter Grade #3: 90
Enter Grade #4: 90
Average is 90.00
```

در برنامه بالا یک آرایه دو بعدی بعدی تعریف شده است (خط ۵-۱). برای درک بهتر آرایه و برنامه به کد زیر توجه کنید:

```
            Grade 1     Grade 2     Grade 3     Grade 4

Student 1     92          87          89          95

Student 2     85          85          86          87

Student 3     90          90          90          90
```

برای پیمایش جدول بالا، که سطرهای آن را دانش آموزان و ستون های آن را نمرات دانش آموزان تشکیل می دهند باید از یک حلقه تو در تو استفاده کنیم (خط ۲۲-۷). در اولین حلقه while یک متغیر به نام student تعریف کرده‌ایم که تعداد دانش آموزان یا تعداد سطرها در آن قرار می‌گیرد (خط ۷). از خاصیت length برای تشخیص تعداد دانش آموزان استفاده شده است. در اینجا lenght عدد ۳ را به ما می دهد. وارد بدنه حلقه while می‌شویم. یک متغیر به نام total تعریف می‌کنیم که جمع نمرات وارد شده برای دانش آموز در آن قرار می‌گیرد (خط ۱۱). یک متغیر هم تعریف کرده ایم که تعداد نمرات دانش آموزان را در خود ذخیره می کند (خط ۱۲). سپس برنامه یک پیغام را نشان می‌دهد و از شما می‌خواهد که نمرات دانش آموز را وارد کنید .(student + 1) عدد ۱ را به student اضافه کرده‌ایم تا به جای نمایش Student 0، با Student 1 شروع شود، تا طبیعی‌تر به نظر برسد (خط ۱۲).

سپس به دومین حلقه while در خط ۱۳ می‌رسیم. وظیفه این حلقه گردش در میان ستون ها، که همان نمرات دانش آموز است، می‌باشد. برنامه چهار نمره مربوط به دانش آموز را می‌گیرد. هر وقت که برنامه یک نمره را از کاربر دریافت می‌کند، نمره به متغیر total اضافه می‌شود (خط ۱۶). وقتی همه نمره‌ها وارد شدند، متغیر total هم جمع همه نمرات را نشان می‌دهد. در خط ۲۱ معدل دانش آموز نشان داده می‌شود. معدل از تقسیم کردن total (جمع) بر تعداد نمرات به دست می‌آید. از studentGrades[student].length هم برای به دست آوردن تعداد نمرات استفاده می‌شود.

متد

متدها به شما اجازه می‌دهند که یک رفتار یا وظیفه را تعریف کنید و مجموعه‌ای از کدها هستند که در هر جای برنامه می‌توان از آنها استفاده کرد. متدها دارای آرگومانهایی هستند که وظیفه متد را مشخص می‌کنند. متد در داخل کلاس تعریف می‌شود. می‌توان یک متد را در داخل متد دیگر تعریف کرد. وقتی که شما در برنامه یک متد را صدا می‌زنید برنامه به قسمت تعریف متد رفته و کدهای آن را اجرا می‌کند.

پارامترها همان چیزهایی هستند که متد منتظر دریافت آنها است.

آرگومان‌ها مقادیری هستند که به پارامترها ارسال می‌شوند.

گاهی اوقات دو کلمه پارامتر و آرگومان به یک منظور به کار می‌روند. ساده‌ترین ساختار یک متد به صورت زیر است :

```
def methodName(Parameter List)
   code to execute
end
```

به برنامه ساده زیر توجه کنید. در این برنامه از یک متد برای چاپ یک پیغام در صفحه نمایش استفاده شده است :

```
1  def printMessage
2      print "Hello World!"
3  end
4
5  printMessage
```

```
Hello World!
```

در خطوط ۱-۳ یک متد تعریف کرده‌ایم. در تعریف متد بالا کلمه کلیدی def آمده است که نشان دهنده تعریف متد است. نام متد ما printMessage است. به این نکته توجه کنید که نام متد با حرف کوچک شروع می شود. بهتر است در نامگذاری متدها از کلماتی استفاده شود که کار آن متد را مشخص می‌کند مثلاً نام‌هایی مانند goToBed یا openDoor. در خط ۵ متد printMessage را صدا می‌زنیم. برای صدا زدن یک متد کافیست نام آن را بنویسیم.

به این نکته توجه کنید که در Ruby، برخلاف بقیه زبان های برنامه نویسی لازم نیست که بعد از نام متد و یا هنگام صدا زدن آن، از پرانتزهای باز و بسته استفاده کنید

اگر متد دارای پارامتر باشد باید شما آرگومانها را به ترتیب در داخل پرانتزها قرار دهید. در این مورد نیز در درسهای آینده توضیح بیشتری می‌دهیم. با صدا زدن یک متد کدهای داخل بدنه آن اجرا می‌شوند. برای اجرای متد printMessage برنامه به محل تعریف متد printMessage می‌رود. مثلاً وقتی ما متد printMessage را در خط ۴ صدا می‌زنیم برنامه از خط ۵ به خط ۱، یعنی جایی که متد تعریف شده می‌رود و کدهای بدنه آن را اجرا می کند.

مقدار برگشتی از یک متد

متدها می‌توانند مقدار برگشتی از هر نوع داده‌ای داشته باشند. این مقادیر می‌توانند در محاسبات یا به دست آوردن یک داده مورد استفاده قرار بگیرند. در زندگی روزمره فرض کنید که کارمند شما یک متد است و شما او را صدا می‌زنید و از او می‌خواهید که کار یک سند را به پایان برساند. سپس از او می‌خواهید که بعد از اتمام کارش سند را به شما تحویل دهد. سند همان مقدار برگشتی متد است. نکته مهم در مورد یک متد، مقدار برگشتی و نحوه استفاده شما از آن است. برگشت یک مقدار از یک متد آسان است. کافیست در تعریف متد به روش زیر عمل کنید :

```
def methodName
    return value
end
```

در داخل بدنه متد کلمه کلیدی return و بعد از آن یک مقدار یا عبارتی که نتیجه آن یک مقدار است را می‌نویسیم. مثال زیر یک متد که دارای مقدار برگشتی است را نشان می‌دهد.

```
1  def calculateSum
2      firstNumber = 10
3      secondNumber = 5
4      sum = firstNumber + secondNumber
5
6      return sum
7  end
8
9  result = calculateSum
10
11 puts "Sum is #{result}."
```

```
Sum is 15.
```

همانطور که مشاهده می‌کنید، در خطوط ۱-۷ یک متد تعریف کرده ایم. در خطوط 2 و ۳ دو متغیر تعریف و مقدار دهی شده‌اند. توجه کنید که این متغیرها، متغیرهای محلی هستند. و این بدان معنی است که این متغیرها در سایر متدها، قابل دسترسی نیستند و فقط در متدی که در آن تعریف شده‌اند قابل استفاده هستند. در خط ۴ جمع دو متغیر در متغیر sum قرار می‌گیرد. در خط ۶ مقدار برگشتی sum توسط دستور return فراخوانی می‌شود. در خط ۹ یک متغیر به نام result تعریف کرده و متد calculateSum را فراخوانی می‌کنیم.

متد calculateSum مقدار ۱۵ را بر می‌گرداند که در داخل متغیر result ذخیره می‌شود. در خط ۱۱ مقدار ذخیره شده در متغیر result چاپ می‌شود. متدی که در این مثال ذکر شد متد کاربردی و مفیدی نیست. با وجودیکه کدهای زیادی در متد بالا نوشته شده ولی همیشه مقدار برگشتی ۱۵ است، در حالیکه می‌توانستیم به راحتی یک متغیر تعریف کرده و مقدار ۱۵ را به آن اختصاص دهیم. این متد در صورتی کارامد است که پارامترهایی به آن اضافه شود که در درسهای آینده توضیح خواهیم داد. هنگامی که می‌خواهیم در داخل یک متد از دستور if استفاده کنیم باید تمام کدها دارای مقدار برگشتی باشند. برای درک بهتر این مطلب به مثال زیر توجه کنید :

```
1  def getNumber
2    print "Enter a number greater than 10: "
3    number = gets.to_i
4    if (number > 10)
5      return number
```

```
  else
    return 0
  end
end

result = getNumber

puts "Result = #{result}."
```

```
Enter a number greater than 10: 11
Result = 11
Enter a number greater than 10: 9
Result = 0
```

در خطوط ۹-۱ یک متد با نام getNumber تعریف شده است که از کاربر یک عدد بزرگتر از ۱۰ را می‌خواهد. اگر عدد وارد شده توسط کاربر درست نباشد متد مقدار صفر را بر می‌گرداند. و اگر قسمت else دستور gif یا دستور return را از آن حذف کنیم در هنگام اجرای برنامه با پیغام خطا مواجه می‌شویم.

چون اگر شرط دستور if نادرست باشد (کاربر مقداری کمتر از ۱۰ را وارد کند) برنامه به قسمت else می‌رود تا مقدار صفر را بر گرداند و چون قسمت else حذف شده است برنامه با خطا مواجه می‌شود و همچنین اگر دستور return حذف شود چون برنامه نیاز به مقدار برگشتی دارد پیغام خطا می‌دهد. و آخرین مطلبی که در این درس می‌خواهیم به شما آموزش دهیم این است که شما می‌توانید از یک متد که مقدار برگشتی ندارد خارج شوید. استفاده از return باعث خروج از بدنه متد و اجرای کدهای بعد از آن می‌شود:

```
def testReturnExit
    puts "Line 1 inside the method TestReturnExit()"
    puts "Line 2 inside the method TestReturnExit()"

    return

    #The following lines will not execute
    puts "Line 3 inside the method TestReturnExit()"
    puts "Line 4 inside the method TestReturnExit()"
end

testReturnExit
puts "Hello World!"
```

```
Line 1 inside the method TestReturnExit()
Line 2 inside the method TestReturnExit()
Hello World!
```

در برنامه بالا نحوه خروج از متد با استفاده از کلمه کلیدی return و نادیده گرفتن همه کدهای بعد از این کلمه کلیدی نشان داده شده است. در کد بالا انتظار ما این است که با فراخوانی متد در خط ۱۲، همه کدهای بدنه متد (۹-۲) اجرا شوند. ولی با فراخوانی متد خطوط ۲ و ۳ چاپ می شوند، چون هنگامی که برنامه به خط ۵ می رسد، از بدنه متد خارج می شود. سپس مفسر به خط ۱۳ رفته و رشته Hello World را چاپ می کند.

پارامترها و آرگومان ها

پارامترها داده‌های خامی هستند که متد آنها را پردازش می‌کند و سپس اطلاعاتی را که به دنبال آن هستید، در اختیار شما قرار می‌دهد. فرض کنید پارامترها مانند اطلاعاتی هستند که شما به یک کارمند می‌دهید که بر طبق آنها کارش را به پایان برساند. یک متد می‌تواند هر تعداد پارامتر داشته باشد. هر پارامتر می‌تواند از انواع مختلف داده باشد. در زیر یک متد با N پارامتر نشان داده شده است :

```
def methodName(param1, param2, ... paramN)
   code to execute
end
```

پارامترها بعد از نام متد و بین پرانتزها قرار می‌گیرند. بر اساس کاری که متد انجام می‌دهد می‌توان تعداد پارامترهای زیادی به متد اضافه کرد. بعد از فراخوانی یک متد باید آرگومانهای آن را نیز تأمین کنید. آرگومان‌ها مقادیری هستند که به پارامترها اختصاص داده می‌شوند. اجازه بدهید که یک مثال بزنیم :

```
1  def calculateSum(number1, number2)
2    return number1 + number2
3  end
4
5  print "Enter the first number: "
6  num1 = gets.to_i
7
8  print "Enter the second number: "
9  num2 = gets.to_i
10
11 puts "Sum = #{calculateSum(num1, num2)}"
```

```
Enter the first number: 10
Enter the second number: 5
Sum = 15
```

در برنامه بالا یک متد به نام calculateSum() (خطوط ۳-۱) تعریف شده است و می خواهیم مقدار دو عدد را با این متد جمع کنیم. در بدنه متد دستور return نتیجه جمع دو عدد را بر می‌گرداند. در خطوط ۹-۵ برنامه از کاربر دو مقدار را درخواست می‌کند و آنها را داخل متغیرها قرار می‌دهد. حال متد را که آرگومانهای آن را آماده کرده‌ایم فراخوانی می‌کنیم. مقدار num1 به پارامتر اول و مقدار num2 به پارامتر دوم ارسال می‌شود. حال اگر مکان دو مقدار را هنگام ارسال به متد تغییر دهیم (یعنی مقدار num2 به پارامتر اول و مقدار num1 به پارامتر دوم ارسال شود) هیچ تغییری در نتیجه متد ندارد چون جمع خاصیت جابه جایی دارد.

فقط به یاد داشته باشید که باید تعداد آرگومانها هنگام فراخوانی متد دقیقاً با تعداد پارامترها تعریف شده در متد مطابقت داشته باشد. بعد از ارسال مقادیر ۱۰ و ۵ به پارامترها، پارامترها آنها را دریافت می‌کنند. به این نکته نیز توجه کنید که نام پارامترها طبق قرارداد به شیوه کوهان شتری یا camelCasing (حرف اول دومین کلمه بزرگ نوشته می‌شود) نوشته می‌شود. در داخل بدنه متد (خط ۲) دو مقدار با هم جمع می‌شوند و نتیجه به متد فراخوان (متدی که متد calculateSum() را فراخوانی می‌کند(ارسال می‌شود. در درس آینده از یک متغیر برای ذخیره نتیجه محاسبات استفاده می‌کنیم ولی در اینجا مشاهده می‌کنید که می‌توان به سادگی نتیجه جمع را نشان داد (خط ۱۱).

در خط ۱۱ متد calculateSum() را فراخوانی می‌کنیم و دو مقدار صحیح به آن ارسال می‌کنیم. دو عدد صحیح در داخل متد با هم جمع شده و نتیجه آنها برگردانده می‌شود. مقدار برگشت داده شده از متد به وسیله متد puts() نمایش داده می‌شود. و نکته آخر اینکه یک متد را می توان به عنوان آرگومان به متد دیگر ارسال کرد. به کد زیر توجه کنید:

```
1  def methodA(mymethod)
2    return mymethod
3  end
4
5  def methodB()
6      return "Hello World!"
7  end
8
9  puts "#{methodA(methodB)}"
```

```
Hello World!
```

در کد بالا ما دو متد تعریف کرده ایم. که متد اول یعنی methodA که یک آرگومان دریافت کند. در خطوط ۷-۵ یک متد تعریف کرده ایم که مقدار Hello World را بر می گرداند. در نتیجه هنگامی که در خط ۷ ما methodB را به عنوان آرگومان به متد methodA می دهیم، مانند این است که ما جمله Hello World را به آن پاس داده ایم.

آرگومان های کلمه کلیدی (Keyword Arguments)

یکی دیگر از راه‌های ارسال آرگومانها استفاده از نام آنهاست. استفاده از نام آرگومانها شما را از به یاد آوری و رعایت ترتیب پارامترها هنگام ارسال آرگومان‌ها راحت می‌کند. در عوض شما باید نام پارامترهای متد را به خاطر بسپارید. استفاده از نام آرگومانها خوانایی برنامه را بالا می‌برد چون شما می‌توانید ببینید که چه مقادیری به چه پارامترهایی اختصاص داده شده است. در زیر نحوه استفاده از آرگومان های کلمه کلیدی، وقتی که متد فراخوانی می‌شود نشان داده شده است :

```
functionToCall( paramName1: value, paramName2: value, ... paramNameN: value)
```

حال به مثال زیر توجه کنید :

```ruby
def tellinformation(jack:, andy:, mark:)
  puts "Jack's family is #{jack}."
  puts "Andy's family is #{andy}."
  puts "Mark's family is #{mark}."
end

tellinformation(jack: "Scalia", andy: "Brown", mark: "OverMars")

#puts  a newline
puts

tellinformation(andy: "Brown", mark: "OverMars", jack: "Scalia")

puts

tellinformation(mark: "OverMars", jack: "Scalia", andy: "Brown")
```

```
Jack's family is Scalia.
Andy's family is Brown.
Mark's family is OverMars.

Jack's family is Scalia.
Andy's family is Brown.
Mark's family is OverMars.

Jack's family is Scalia.
Andy's family is Brown.
Mark's family is OverMars.
```

خروجی نشان می‌دهد که حتی اگر ما ترتیب آرگومانها در سه بار فراخوانی متد را تغییر دهیم مقادیر مناسب به پارامترهای مربوطه‌شان اختصاص داده می‌شود. وقتی از آرگومان های کلمه کلیدی استفاده می کنید دیگر نمی توانید هنگام فراخوانی متد، آرگومانهای ثابت (مقداری) را به کار ببرید.

مثلا همه کدهای زیر با خطا مواجه می شوند:

```ruby
#The following codes are wrong and will lead to errors

tellinformation("Scalia", andy: "Brown", mark: "OverMars")

tellinformation(jack: "Scalia", andy: "Brown", "OverMars")

tellinformation(jack: "Scalia", "OverMars", andy: "Brown")
```

آرگومان های متغیر

با استفاده از دستورات خاص *args و **kwargs می‌توان تعداد دلخواهی از آرگومان‌ها را به متد ارسال کرد. همانطور که در درس‌های قبل ذکر شد، هنگام فراخوانی متد باید به تعداد پارامترهایی که در داخل پرانتز تعریف شده‌اند، آرگومان به متد ارسال کرد. گاهی اوقات در برنامه نویسی ممکن است بخواهید که در هر بار فراخوانی متد تعداد دلخواهی آرگومان به آن ارسال کنید. این کار با استفاده از * و ** ممکن است. به کد زیر توجه کنید :

```
def varArguments(*args)
    total = 0

    for number in args
      total = total + number
    end

    return total
end

puts "1 + 2 + 3 = #{varArguments(1, 2, 3)}"
puts "1 + 2 + 3 + 4 = #{varArguments(1, 2, 3, 4)}"
puts "1 + 2 + 3 + 4 + 5 = #{varArguments(1, 2, 3, 4, 5)}"
```

```
1 + 2 + 3 = 6
1 + 2 + 3 + 4 = 10
1 + 2 + 3 + 4 + 5 = 15
```

ابتدا به این نکته توجه کنید که نامهای args و kwargs اختیاری هستند و هم نام دیگری می‌تواند به جای آنها به کار رود. تنها چیزی که مهم است تعداد علامت * می‌باشد که در ادامه کاربرد آنها را توضیح می‌دهیم.

همانطور که در کد بالا مشاهده می‌کنید، با قرار دادن یک علامت ستاره قبل از نام پارامتر، می‌توان هر بار که متد را فراخوانی کرد، تعداد دلخواهی از آرگومانها را به آن ارسال کرد. وجود یک علامت * باعث می‌شود که آرگومانها در یک متغیر از جنس آرایه ذخیره شوند و در نتیجه می‌توان با یک دستور for مقادیر آنها را با هم جمع کرد. این نوع پارامتر را می‌توان با پارامترهای ثابت هم به کار برد. به مثال زیر توجه کنید:

```
def varArguments(number, *args)
    puts "number = #{number}"
    puts "args = #{args}"
end

varArguments(1, 2, 3)
```

```
number =  1
args =  [2, 3]
```

در کد بالا اولین آرگومان به اولین پارامتر)یعنی ۱ به (number و بقیه آرگومانها به args اختصاص داده می‌شوند. وقتی از چندین پارامتر در یک متد استفاده می‌کنید فقط یکی از آنها باید دارای * بوده و همچنین از لحاظ مکانی باید آخرین پارامتر باشد. اگر این پارامتر (پارامتری که دارای علامت * است) در آخر پارامترهای دیگر قرار نگیرد و یا از چندین پارامتر علامت دار استفاده کنید با خطا مواجه می‌شوید. به مثالهای اشتباه و درست زیر توجه کنید :

```
def someMethod(*args, *args) #ERROR

def someMethod(*args, param1, param2) #ERROR

def someMethod(param1, param2, *args) #Correct
```

البته می‌توان پارامتر ستاره دار را در ابتدای پارامترهای دیگر قرار داد ولی پارامترهای بعد از این پارامتر یا باید دارای مقدار پیشفرض باشند

```
def varArguments(*args, number:10)
  puts "args = #{args}"
  puts "number = #{number}"
end

varArguments(1, 3, 5)
```

```
args = [1, 3, 5]
number = 10
```

و یا هنگام فراخوانی متد، باید پارامترهای بعد از این پارامتر را با استفاده از اسمشان مقداردهی کرد. به کد زیر توجه کنید:

```
def varArguments(*args, number:)
  puts "args = #{args}"
  puts "number = #{number}"
end

varArguments(1, 3, 5, number: 10)
```

```
args = (1, 3, 5)
number = 10
```

حال فرض کنید که می‌خواهید چند آرایه به پارامتر ستاره دار ارسال کنید. به کد زیر توجه نمایید :

```
def varArguments(number, *args)
  puts "number = #{number}"
  puts "args = #{args}"
end

varArguments(1, *[2,3,4], *[5,6,7,8])
```

```
number = 1
args = [2, 3, 4, 5, 6, 7, 8]
```

همانطور که در کد بالا مشاهده می‌کنید، کافیست که آرگومان‌ها به صورت آرایه ارسال شوند و قبل از آنها علامت * را قرار دهیم. خط آخر کد بالا را به صورت زیر هم می‌توان نوشت :

```
varArguments(1, *[2,3,4], *[5,6,7,8])
```

**kwargs هم شبیه *args عمل می‌کند با این تفاوت که هنگام فراخوانی متد می توان آرگومان‌ها را به صورت hash به آن ارسال کرد:

```
def varArguments(**kwargs)
    puts (kwargs)
end

varArguments(person1: "Jack", person2: "Joe", person3: "Smith")
```

```
{:person1=>"Jack", :person2=>"Joe", :person3=>"Smith"}
```

محدوده متغیر

متغیرها در Ruby دارای محدوده هستند. محدوده یک متغیر به شما می‌گوید که در کجای برنامه می‌توان از متغیر استفاده کرد و یا متغیر قابل دسترسی است. به عنوان مثال متغیری که در داخل یک متد تعریف می‌شود فقط در داخل بدنه متد قابل دسترسی است. می‌توان دو متغیر با نام یکسان در دو متد مختلف تعریف کرد. برنامه زیر این ادعا را اثبات می‌کند :

```
1  def firstLocalVariable
2    number = 10
3    puts number
4  end
5
6  def secondLocalVariable
7    number = 20
8    puts number
9  end
10
11 firstLocalVariable
12
13 secondLocalVariable
```

```
10
20
```

مشاهده می‌کنید که حتی اگر ما دو متغیر با نام یکسان تعریف کنیم (خطوط ۲ و ۷) که دارای محدوده‌های متفاوتی هستند، می‌توان به هر کدام از آنها مقادیر مختلفی اختصاص داد. متغیر تعریف شده در داخل متد firstLocalVariable هیچ ارتباطی به متغیر داخل متد secondLocalVariable ندارد. وقتی به مبحث کلاسها رسیدیم در این باره بیشتر توضیح خواهیم داد. Ruby دارای چهار محدوده است:

- متغیرهای محلی (Local variable)
- متغیرهای سراسری (Global variable)
- متغیرهای نمونه (Instance variable)
- متغیرهای کلاس (Class variable)

متغیرهای محلی (Local variable)

متغیرهایی که داخل متدها تعریف می‌شوند محلی هستند و فقط داخل همان متد قابل استفاده‌اند. به مثال زیر توجه کنید:

```
1  def LocalVariable
2    number = 10
3    puts number
4  end
5
6  LocalVariable()
7
8  puts number
```

```
10
undefined local variable or method `number' for main:Object (NameError)
```

همانطور که مشاهده می‌کنید با فراخوانی متد در خط ۶ مقدار متغیر number چاپ می‌شود ولی در خط ۸ که سعی در چاپ مقدار این متغیر داریم با پیغام خطا مواجه می‌شویم چون طول عمر این متغیر تا زمانی است که متد به پایان نرسیده است. با پایان متد متغیر و مقدار آن هم از بین می‌رود در نتیجه در خارج از متد نمی‌توان مقدار آن را چاپ کرد.

متغیرهای سراسری (Global variable)

متغیرهایی که در بیرون متد تعریف می‌شوند و نام آنها با علامت $ شروع می شود، از نوع سراسری هستند. به مثال زیر توجه کنید:

```
1  $firstNumber  = 10
2  $secondNumber = 5
3  $sum
4
5  def globalVariable
6    $sum = $firstNumber + $secondNumber
7    puts $sum
8  end
9
10 globalVariable()
11 puts $sum
```

```
15
15
```

متغیرهای $firstNumber و $secondNember و $sum در بیرون متد تعریف شده‌اند و از نوع سراسری هستند، در داخل متد اگر بخواهیم به آنها دسترسی پیدا کنیم باید قبل از نام آنها از علامت $ استفاده نماییم. در مورد متغیرهای نمونه و کلاس در آینده توضیح می دهیم.

پارامترهای پیشفرض

پارامترهای پیشفرض همانگونه که از اسمشان پیداست دارای مقادیر پیشفرضی هستند و می‌توان به آنها آرگومان ارسال کرد یا نه. اگر به اینگونه پارامترها، آرگومانی ارسال نشود از مقادیر پیشفرض استفاده می‌کنند. به مثال زیر توجه کنید :

```ruby
def printMessage(message = "Welcome to Ruby Tutorials!")
  puts message
end

printMessage()
printMessage("Learn Ruby Today!")
```

```
Welcome to Ruby Tutorials!
Learn Ruby Today!
```

متد printMessage() (خطوط ۳-۱) یک پارامتر اختیاری دارد. برای تعریف یک پارامتر اختیاری می‌توان به آسانی و با استفاده از علامت = یک مقدار را به یک پارامتر اختصاص داد (مثال بالا خط ۱). دو بار متد را فراخوانی می‌کنیم. در اولین فراخوانی (خط ۵) ما آرگومانی به متد ارسال نمی‌کنیم بنابراین متد از مقدار پیشفرض (Welcome to Ruby Tutorials!) استفاده می‌کند. در دومین فراخوانی (خط ۶) یک پیغام (آرگومان) به متد ارسال می‌کنیم که جایگزین مقدار پیشفرض پارامتر می‌شود. اگر از چندین پارامتر در متد استفاده می‌کنید همه پارامترهای اختیاری باید در آخر بقیه پارامترها ذکر شوند. به مثالهای زیر توجه کنید:

```ruby
def someMethod(default1 = 10, default2 = 20, require1, require2) #ERROR

def someMethod(require1, default1 = 10, require2, default2 = 20) #ERROR

def someMethod(require1, require2, default1 = 10, default2 = 20) #Correct
```

وقتی متدها با چندین پارامتر اختیاری فراخوانی می‌شوند باید به پارامترهایی که از لحاظ مکانی در آخر بقیه پارامترها نیستند مقدار اختصاص داد. به یاد داشته باشید که نمی‌توان برای نادیده گرفتن یک پارامتر به صورت زیر عمل کرد :

```ruby
def someMethod(required1, default1 = 10,  default2 = 20)
   #Some Code
end

someMethod(10, , 100); #Error
```

بازگشت (Recursion)

بازگشت فرایندی است که در آن متد مدام خود را فراخوانی می‌کند تا زمانی که به یک مقدار مورد نظر برسد. بازگشت یک مبحث پیچیده در برنامه نویسی است و تسلط به آن کار راحتی نیست. به این نکته هم توجه کنید که بازگشت باید در یک نقطه متوقف شود وگرنه برای بی نهایت بار، متد، خود را فراخوانی می‌کند. در این درس یک مثال ساده از بازگشت را برای شما توضیح می‌دهیم. فاکتوریل یک عدد صحیح مثبت (n!) شامل حاصل ضرب همه اعداد مثبت صحیح کوچکتر یا مساوی آن می‌باشد. به فاکتوریل عدد ۵ توجه کنید.

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

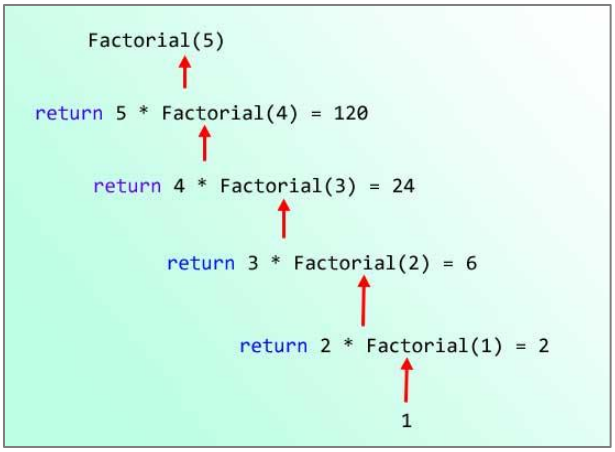
بنابراین برای ساخت یک متد بازگشتی باید به فکر توقف آن هم باشیم. بر اساس توضیح بازگشت، فاکتوریل فقط برای اعداد مثبت صحیح است. کوچک‌ترین عدد صحیح مثبت ۱ است. در نتیجه از این مقدار برای متوقف کردن بازگشت استفاده می‌کنیم.

```
def factorial(number)

  if (number == 1)
    return 1
  end

  return number * factorial(number - 1)
end

print(factorial(5))
```

```
120
```

متد مقدار بزرگی را بر می‌گرداند چون محاسبه فاکتوریل می‌تواند خیلی بزرگ باشد. متد یک آرگومان که یک عدد است و می‌تواند در محاسبه مورد استفاده قرار گیرد را می‌پذیرد. در داخل متد یک دستور if می‌نویسیم و در خط 3 می گوییم که اگر آرگومان ارسال شده برابر ۱ باشد سپس مقدار ۱ را برگردان در غیر اینصورت به خط بعد برو. این شرط باعث توقف تکرارها نیز می‌شود.

در خط ۷ مقدار جاری متغیر number در عددی یک واحد کمتر از خودش (number - 1) ضرب می‌شود. در این خط متد factorial خود را فراخوانی می‌کند و آرگومان آن در این خط همان number - 1 است. مثلاً اگر مقدار جاری number 10 باشد یعنی اگر ما بخواهیم فاکتوریل عدد ۱۰ را به دست بیاوریم آرگومان متد factorial در اولین ضرب ۹ خواهد بود. فرایند ضرب تا زمانی ادامه می‌یابد که آرگومان ارسال شده با عدد ۱ برابر نشود. شکل زیر فاکتوریل عدد ۵ را نشان می‌دهد.



کد بالا را به وسیله یک حلقه while نیز می‌توان نوشت.

```
num = 5
factorial = 1

while num > 1
  factorial = factorial * num
  num = num - 1
end

print(factorial)
```

این کد از کد معادل بازگشتی آن آسان‌تر است. از بازگشت در زمینه‌های خاصی در علوم کامپیوتر استفاده می‌شود. استفاده از بازگشت حافظه زیادی اشغال می‌کند پس اگر سرعت برای شما مهم است از آن استفاده نکنید.

عبارات لامبدا (Lambda expressions)

عبارات لامبدا در اصل توابع یک خطی هستند که در برخی از زبان ها به عنوان توابع بی نام شناخته می شوند. گاهی اوقات اتفاق می افتد که در برنامه نمی خواهید یک تابع را جهت انجام یک کار تعریف کنید. در این صورت می توان از عبارات لامبدا استفاده کرد. نحوه استفاده از لامبدا به صورت زیر است :

```
nameOfLambda = lambda  {|parameter| expression }
```

یا

```
nameOfLambda = -> (parameter) { expression }
```

nameOfLambda نامی است که برای عبارت لامبدا در نظر می گیریم و در برنامه از همین نام برای اجرای کدهای آن استفاده می کنیم. حال یک مثال ساده از عبارت لامبدا را به دو روش فوق پیاده سازی می کنیم:

```
showMessage = lambda { |message| print(message) }
showMessage.call("Hello World!")
```

```
Hello World!
```

در کد بالا ما یک عبارت لامبدا به نام showMessage تعریف کرده ایم که یک پارامتر به نام message دارد. برای اجرای کدهای یک عبارت لامبدا، باید نام آن را نوشته، علامت نقطه بگذارید و سپس متد call را فراخوانی کنید. اگر عبارت لامبدا، پارامتر قبول کند باید آرگومان هایی که قرار است به آن ارسال شوند را در داخل پرانتزهای متد call بنویسید. در مثال بالا عبارت لامبدا یک پارامتر به نام message دارد. در نتیجه ما از طریق متد call یک آرگومان به آن ارسال کرده ایم. کد بالا را به صورت زیر هم می توان نوشت:

```
showMessage = -> (message) { print(message) }
showMessage.call("Hello World!")
```

لامبدا می تواند، هیچ پارامتری نگیرد:

```
showMessage = lambda { print("Hello World!") }
showMessage.call
```

```
Hello World!
```

هنگام فراخوانی متد باید تعداد آرگومان ها با تعداد پارامترها برابر باشد. مثلا برنامه زیر با خطا مواجه می شود:

```
showMessage = lambda { |message1, message2| puts(message1, message2) }
showMessage.call("Hello World!")
```

در مثال بالا، عبارت لامبدا دو پارامتر قبول می کنید (|message1, message2|) ولی ما در داخل متد call یک آرگومان به آن ارسال کرده ایم.

عبارات لامبدا نمی توانند دارای کلمه return باشند. می توان گفت که دستورات لامبدا در حالت عادی برگردانده می شوند و نیازی به این کلمه نیست:

```
getSquare = lambda { |number| number * number }
print(getSquare.call(5))
```

```
25
```

پایان ویرایش اول کتاب – ۹۹/۰۵/۲۳

راه‌های ارتباط با نویسنده

وب سایت: www.w3-farsi.com

لینک تلگرام: https://telegram.me/ebrahimi_younes

ID تلگرام: @ebrahimi_younes

پست الکترونیکی: younes.ebrahimi.1391@gmail.com