کتاب آموزش روبی
(نسخه دوم)

محمدرضا حقیری

Table of Contents

نسخه دوم آموزش روبی

در این کتاب، همانند کتاب پیشین، سعی بر آن است که تمامی مباحث مربوط به زبان برنامه نویسی روبی، در بر گرفته شود. همچنین، در این نسخه علاوه بر مباحث مرسوم که در نسخه پیشین مورد بحث بود، مباحث شیء گرایی نیز اضافه شده اند. همچنین، کاربردهای زبان روبی نیز به طور خلاصه در انتهای کتاب، بررسی گشته است.

این کتاب برای چه کسانی است؟

بدون شک همه علاقمندان به برنامه نویسی و بخصوص حوزه وب، این کتاب جذاب خواهد بود. بخصوص اگر به دنبال زبانی ساده و نزدیک به زبان انسان هستید، به شما توصیه می شود که از روبی شروع کنید. تمامی مفاهیمی که در زبانهای دیگر وجود دارند، در روبی نیز به صورت بسیار ساده و قابل فهم تر پیاده شده اند.

آشنایی با روبی

روبی یک زبان اسکریپتی و شی گراست که در دهه نود میلادی، توسط Yukihiro Matsumoto ساخته شد. این زبان، با رویکرد جالب خود، یعنی «همه چیز شیء است» توانست توجه بسیاری از برنامه نویسان بخصوص آنهایی که با پردازش متن سر و کار داشتند را به خود جلب کند. اگرچه، روبی تنها برای پردازش متن نیست بلکه برای هر منظوری، مانند طراحی بکند وب سرویس ها، پیکربندی سیستم عامل و ... ای قابل استفاده است. در این کتاب، ما در محیط سیستم عامل لینوکس (همچون نسخه قبلی) به استفاده از روبی میپردازیم.

نصب و راه اندازی

نصب و راه اندازی روبی، روی سیستم عامل لینوکس بسیار ساده است. با فرض این که شما از اوبونتو استفاده میکنید، کافیست یک پنجره ترمینال باز کرده سپس تایپ کنید :

```
sudo apt-get install ruby
```

پس از اتمام نصب، کافیست در پنجره ترمینال تایپ کنید :

```
irb
```

سپس باید یک prompt به این شکل دریافت کنید :

```
irb(main):001:0>
```

در این محیط، میتوانید دستورات مورد نظر خود را تایپ کرده و سپس نتیجه آن را ببینید.

اولین خط برنامه شما

بسیار خوب، اکنون بیایید یک کد واقعی بنویسیم، برای تست کردن در محیط irb تایپ کنید :

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

و باید چنین نتیجه ای بگیرید :

```
irb(main):001:0> puts "Hello, World!"
Hello, World!
=> nil
irb(main):002:0>
```

در واقع با استفاده از دستور puts در زبان روبی، میتوان رشته ای را چاپ نمود و سپس به خط بعدی رفت. در مورد رشته ها و چگونگی آن ها، در فصل بعدی صحبت بیشتری خواهد شد.

پایه ها و مقدمات

در فصل قبلی، با کلیت روبی، نصب و همچنین استفاده از محیط تعاملی آن، آشنا شدیم. در این فصل، این موارد را بیشتر بررسی خواهیم کرد.

دستورات پایه در محیط تعاملی

در محیط تعاملی روبی، میتوان هر دستورالعملی را به کار گرفت و نتیجه اش را همان لحظه دید. برای مثال این عملیات های ساده ریاضی را در نظر بگیرید:

```
2 + 2
4 + 5
3 / 4
2 % 3
```

این دستورات همگی در محیط تعاملی قابل انجام بوده، و همچنین نتیجه ای را برمیگردانند. برای مثال اعداد بالا را در نظر بگیرید :

```
irb(main):001:0> 2 + 2
=> 4
irb(main):002:0> 4 + 5
=> 9
irb(main):003:0> 3 / 4
=> 0
irb(main):004:0> 2 % 3
=> 2
irb(main):005:0>
```

در اینجا، نتایجی برگشت که انتظارش را داشتیم. دلیل برگشت داده شدن مقدار صفر توسط عمل تقسیم نیز در فصل بعدی، به تفصیل مورد بررسی قرار خواهد گرفت.

نوشتن اسکریپت و سپس اجرای آن

بیاید فرض کنیم میخواهیم برنامه های روبی را در یک فایل نوشته و سپس اجرا کنیم. برای این کار کافیست تا یک فایل با نام دلخوه و همچنین پسوند rb در مسیر دلخواه خود ایجاد کرده، اسکریپت خود را در آن بنویسیم. البته برای اجرای اسکریپت دو روش وجود دارد، که این روش، روش اول ماست. برای مثال درون اسکریپت خود این چنین مینویسیم :

```
puts 2 + 2
puts 4 + 5
puts 3 / 4
puts 2 % 3
```

سپس، یک پنجره ترمینال باز کرده ، این دستور را در پنجره باز شده تایپ میکنیم :

```
ruby /path/to/script.rb
```

دقت کنید که باید مسیری که اسکریپت در آن ذخیره شده است به عنوان ورودی به مفسر داده شود. همچنین، به این دلیل اینجا از puts استفاده شد که بتوان نتیجه را چاپ کرد. اگر از puts استفاده نکنیم، اسکریپت اجرا می شود اما خروجی ای روی کنسول نگاشته نخواهد شد.

روش دوم

روش دوم، نیازی به پسوند rb هم ندارد، البته در لینوکس و سیستم های مبتنی بر یونیکس. برای این کار کافیست به این شکل اسکریپت خود را بنویسیم :

```
#!/usr/bin/ruby

puts 2 + 2
puts 4 + 5
puts 3 / 4
puts 2 % 3
```

سپس ، یک پنجره ترمینال باز کرده و این دستور را در آن تایپ میکنیم :

```
chmod +x script.rb
```

به این شکل، به فایل خود دسترسی اجرایی داده ایم. حالا کافیست به این شکل از ترمینال فایل را اجرا کنیم :

```
./script.rb
```

و نتایج مطلوب ما، به این طریق چاپ خواهد شد.

جمع بندی

در این فصل، با محیط تعاملی و همچنین روش های نگارش و اجرای اسکریپت های روبی آشنا شدیم. در فصلهای بعدی، با مفاهیم پایه ای روبی آشنا شده و سپس به سراغ مفاهیم برنامه نویسی و نوشتن برنامه های جدی تر خواهیم رفت.

متغیرها و انواع داده ها

در این فصل، انواع اصلی متغیرها و داده ها در روبی، و همچنین نحوه تعریف و به کارگیری و تبدیل آنها را بررسی خواهیم کرد. پس از این فصل، شما قادر خواهید بود برنامه هایتان را با استفاده از متغیر ها کنترل کنید.

انواع داده های عددی

اعداد صحیح

هر گونه عددی که فاقد بخش اعشاری باشد، از دید مفسر روبی، یک متغیر از نوع عدد صحیح تلقی می شود. مثلا اگر دو عدد زیر را بررسی کنیم :

```
a = 2
b = 2.0
```

با این که از نظر منطقی، مقادیر برابری دارند، اما از نظر برنامه نویسی دو «نوع» متفاوت هستند.

ممیز شناور

این گونه از اعداد، دارای بخش اعشاری بوده و از دید مفسر از نوع Float در نظر گرفته می شوند. با توجه به مثال قبل، متغیر a یک عدد صحیح و متغیر b یک عدد ممیز شناور است.

اعداد در سایر مبناها

با وجود این که مبنای مرسوم، مبنای ده است، ما ممکن است گاهی نیاز داشته باشیم مبنای دیگری را نیز به عنوان ورودی یا خروجی خود بخوانیم و بنویسیم. با این حساب، باید بدانیم که مفسر روبی بین اعداد مبناهای مختلف چه تفاوتی قائل است.

اعداد دودویی

برای استفاده از اعداد دودوئی کافیست به این فرمت آنها را تعریف کنیم :

```
a = 0b10
b = 0b101
c = 0b1010
```

و سپس مفسر، مقدار دهدهی آن ها را حساب کرده، در محاسبات برنامه از آن استفاده خواهد کرد.

اعداد مبنای ۸

برای اینکه عددی در مبنای ۸ را بعنوان مقدار ورودی یک متغیر بدهیم، کافیست از این فرمت استفاده کنیم :

```
a = 02
b = 05
c = 012
```

اعداد شانزده شانزدهی

اعداد شانزده شانزدهی یا هگزادسیمال نیز میتوانند به عنوان مقادیر معتبر یک متغیر پذیرفته شوند. برای این که مقدار هگزادسیمال را وارد یک متغیر کنیم کافیست به این شکل عمل کنیم :

```
a = 0x2
b = 0x5
c = 0xa
```

داده های بولین

داده های بولی، مقادیری هستند که صرفا درست یا غلطند. از این مقادیر به کرات در نوشتن توابع، و یا کنترل برنامه توسط ساختارهای کنترلی و حلقه های تکرار استفاده خواهیم کرد. مقادیری که متغیرهای بولین قبول میکنند، یا true است یا false .

رشته ها

هر رشته، خود مجموعه ای از چندین کاراکتر است که دنبال هم قرار گرفته اند. مثلا :

```
puts "Hello, World"
```

در اینجا، مقداری که بین علامت نقل قول قرار گرفته، یک رشته است.

تعریف متغیر ها در روبی

برای تعریف یک متغیر، باید نامی برای آنها برگزینیم. دقت کنید که متغیر همانطور که از نامش پیداست، مقدارش تغییر خواهد کرد. متغیرها با حروف کوچک شروع می شوند. سپس با علامت مساوی، باید مقدار سمت راست را به متغیر خود نسبت دهیم. به طور کلی فرم متغیرهای ما اینگونه است :

```
var = "This is variable"
var = 2
```

اگر مقدار var را چاپ کنیم، قطعا عدد ۲ چاپ خواهد شد، چرا که مقدارش را تغییر داده ایم.

ثابت ها

برای تعریف ثابت در روبی، کافیست که نام را با حرف بزرگ شروع کنیم. ثابت ها فقط و فقط توسط برنامه نویس ها قابل تغییرند، و در حین اجرای برنامه مقدارشان تغییر نخواهد کرد.

```
Var = 100
```

چنانچه در کد خود بخواهید مقدار ثابت فوق را تغییر دهید، با خطا مواجه خواهید شد.

جمع بندی

در این فصل، با کلیت ثوابت و متغیرها در روبی آشنا شدید. اکنون تا حدود زیادی دانش برنامه نویسی در روبی را کسب کرده اید. در فصول بعدی، با عملگر ها آشنا خواهیم شد و سپس با انواع داده ای جدید و دستوراتی که میتوانند ما را به نوشتن برنامه های واقعی، نزدیک تر کنند.

آرایه ها و جداول درهم سازی

در فصل قبل، با انواع داده های روبی آشنا شدیم. در این فصل، با دو ساختمان داده مهم و کلیدی در روبی و همچنین خواص آن ها آشنا خواهیم شد.

آرایه ها

یک آرایه، مجموعه ای از انواع داده های مختلف است که به ترتیب در مکان مشخصی از حافظه قرار گرفته اند. در روبی، امکان این وجود دارد که اعضای یک آرایه از انواع مختلف باشند. مثلا یک آرایه میتواند شامل اعداد صحیح، رشته ها و حتی آرایه های دیگر باشد. برای تعریف یک آرایه در روبی به این شکل عمل میکنیم :

```
array = [1, 2, 3, 4]
```

هر کدام از این خانه ها، یک اندیس مخصوص به خود دارند که از صفر شروع می شود. مثلا :

```
array[0]
```

مقدار ۱ را برمیگرداند، چرا که مقدار ۱ را در خانه صفرم آرایه قرار داده ایم. از آنجایی که آرایه ها در روبی، داینامیک هستند، و ممکن است بعد از عملیاتی سایز آرایه را ندانیم، میتوانیم با استفاده از اندیس منفی، آرایه را برعکس بخوانیم :

```
array[-1]
```

که در اینجا مقدار ۴ را بر خواهد گرداند.

مرتب سازی آرایه

معمولا، برای راحت شدن عملیاتی همچون جست و جو در یک آرایه، مقادیر آرایه را به صورت صعودی یا نزولی، مرتب می شوند. در روبی، متد خاصی برای این کار در نظر گرفته شده است :

```
array = [2 , 3, 1, 4]
array = array.sort
```

با استفاده از متد sort ، این آرایه به شکل صعودی مرتب می شود.

معکوس کردن یک آرایه

با استفاده از متد reverse میتوان آرایه را از آخر به اول چید، و با ترکیب آن با متد sort میتوان به صورت نزولی مرتب نمود. این متد در رشته ها نیز وجود دارد (رشته نیز خود به نوعی یک آرایه است) :

```
array.reverse
```

بدست آوردن طول یک آرایه

طول آرایه ها نیز برای ما حائز اهمیت است. بخصوص اگر قرار باشد در آرایه، جست و جو و ... انجام دهیم. برای بدست آوردن طول یک آرایه کافیست به این شکل عمل کنیم :

```
array.length
```

این متد نیز با رشته ها مشترک است.

جست و جو در آرایه

برای جست و جو در آرایه نیز کافیست از متد مربوطه در روبی استفاده شود. مثلا میخواهیم ببینیم مقدار ۵ در آرایه ما وجود دارد یا نه :

```
array.include?5
```

از این متد میتوان برای جست و جوی یک کاراکتر در رشته نیز استفاده نمود.

درج عضو جدید در آرایه

برای درج عضو جدید، کافیست آخرین اندیس ممکن را به دست آورده، که مثلا در آرایه نمونه ما ۳ بزرگترین اندیس است. سپس، مقدار جدید را وارد کنید :

```
array[4] = 5
```

حذف عضو از آرایه

برای حذف عضوی از یک آرایه، کافیست به این شکل عمل کنیم :

```
array.delete_at(4)
```

به این شکل مقداری که در اندیس ۴ ذخیره شده است ، حذف می شود. اگرچه، مستقیما هم میتوان توسط مقدار عملیات حذف را انجام داد.

```
array.delete(4)
```

و مقدار ۴، در اینجا در اندیس ۳ ذخیره شده و بعد از انجام متد delete حذف میگردد.

جداول درهم سازی

یک جدول درهم سازی، نوعی ساختمان داده است که هر عضو آن، یک کلید مخصوص دارد. یکی از کاربردهای جدوال درهم سازی در رمز نگاری است. اگرچه در برنامه ها، بیشتر بعنوان نوعی دیکشنری از آن استفاده خواهیم کرد. یک جدول درهم سازی ساده، به این شکل تعریف می شود :

```
hash_table = {:key => "value"}
```

برای مثال، میتوان اعداد را به این شکل وارد یک جدول درهم سازی نمود :

```
hash_table = {1 => "One", 2 => "Two", 3 => "Three", 4 => "Four"}
```

به این شکل، با فراخوانی کلیدی که تعیین شده، میتوان مقدار را فراخوانی کرد. مثلا :

```
hash_table[2]
```

مقدار Two را برمیگرداند.

درج عضو

برای درج یک عضو در جدول درهم سازی، کافیست کلید جدیدی در نظر بگیریم و مقداری را به آن نسبت دهیم. به این شکل :

```
hash_table[5] = "Five"
```

سپس، عضو مورد نظر ما در جدول درهم سازی، درج خواهد شد.

حذف عضو

برای حذف کردن اعضای جداول درهم سازی، کافیست تا کلید آن را پاک کنیم. سپس، مقدار آن نیز از جدول درهم سازی ما، پاک خواهد شد :

```
hash_table.delete(5)
```

جست و جو

برای جست و جو کردن در یک جدول درهم سازی، دو روش داریم. اولین روش، جست و جو با کلید (و معمولا روش مرسوم) است و روش دوم جست و جو با مقدار.

جست و جو با مقدار

برای جست و جو کردن در یک جدول درهم سازی با مقدار، کافیست این چنین عمل کنیم :

```
hash_table.has_value?2
```

در این روش مطمئن می شویم مقداری که به دنبالش هستیم، حتما در جدول درهم سازی ما وارد شده است.

جست و جو با کلید

برای جست و جو کردن توسط کلید ، کافیست به این شکل عمل کنیم :

```
hash_table.has_key?3
```

و سپس در صورت درست بودن نتیجه میتوانیم کلیدی که وارد کردیم را در خود جدول درهم سازی، فراخوانی کنیم.

جمع بندی

در این فصل، با دو ساختمان داده بسیار مهم در روبی آشنا شدیم. در فصول بعدی، برنامه هایی خواهیم نوشت که از این دو ساختمان داده در آنها استفاده شده است. همچنین، فراخواهیم گرفت که چطور از این ساختمان های داده ای و خواصشان، در برنامه های خود استفاده نماییم.

عملگرها

در دو فصل گذشته، در مورد انواع داده ها و همچنین ساختمان داده ها در روبی بحث شد. در این فصل، در مورد عملگرها و انجام عملیات های محاسباتی و منطقی صحبت می شود.

عملگرهای محاسباتی

عملگرهای محاسباتی، عملگرهایی هستند که برای انجام عملیات ریاضی به کار می روند و در مقدار تغییر ایجاد میکنند. عملیاتی مانند جمع، تفریق، ضرب و تقسیم از عملگرهای محاسباتی هستند. البته عملگرهای محاسباتی رو انواع دیگر داده ها نیز کار میکنند. در اینجا مثالی از به کارگیری عملیات محاسباتی روی اعداد را می بینیم :

```
a = 2
b = 3
c = a + b
d = c - b
```

عملیات روی رشته ها

از عمل جمع، میتوان برای الحاق دو رشته به یکدیگر، و از عمل ضرب میتوان برای تکرار یک رشته استفاده کرد.

```
str1 = "Hello "
str2 = "World!"
str = str1 + str2
str * 3
```

با اجرای کد فوق عبارت Hello, World! سه بار روی کنسول نمایان می شود.

عملیات روی آرایه ها

با عمل جمع، میتوان دو آرایه را به یکدیگر الحاق کرد. با عمل تفریق، میتوان تعدادی از اعضا را حذف نمود و با عمل ضرب نیز میتوان یک آرایه را چندین بار تکرار کرد.

```
array1 = [1, 2, 3]
array2 = [4, 5, 6]
array3 = array1 + array2
array = array3 - [2, 5]
array * 2
```

عملگرهای منطقی

عملگرهای منطقی، برای مقایسه و بررسی تساوی به کار می روند. این عملگرها، با مقدار کاری ندارند بلکه نتیجه درست یا غلط بر میگردانند. عملگرهای زیر همه از عملگرهای منطقی محسوب می شوند :

```
a > b # بـزرگتر
a < b # کـوچکتر
a >= b # بـزرگتر مـساوی
a <= b # کـوچکتر مـساوی
a == b # تـساوی
a === b # تـساوی
```

عملگرهای بیتی

این عملگرها، روی بیت به بیت داده های ورودی، اعمال می شوند. با استفاده از این عملگرها می توان به اشتراک بیت های دو مقدار، نقیض یک مقدار و حتی حاصل جمع یک مقدار به مقدار دیگر، پی برد.

AND

با استفاده از این عملگر، اشتراک بیت ها گرفته می شود، میتوان فهمید دو مقدار چند بیت مشترک دارند.

```
2 & 3
```

OR

با استفاده از این عملگر، میتوان اجتماع دو مقدار را به دست آورد

```
2 | 3
```

NOT

با استفاده از این عملگر نیز، ما نقیض یک مقدار یا مکمل ۱ آن را به دست می آوریم:

```
~2
```

XOR

با استفاده از این عملگر، میتوان حاصل جمع دو مقدار را حساب نمود، همچنین توازن بیت های فرد دو مقدار نیز محاسبه می شود.

```
2 ^ 3
```

جمع بندی

در این فصل، با سه نوع عملگر مهم در زبان روبی آشنا شدید. از فصل بعدی، به طور جدی، شروع به نوشتن برنامه های واقعی خواهیم کرد، برنامه هایی که میتوانند در دنیای واقعی، مورد استفاده قرار گرفته و بخشی از پروژه های بزرگتر باشند.

ساختار های کنترلی

در این فصل، با روش های برنامه سازی ساخت یافته آشنا می شویم. در این روش، ما «ساختار» هایی داریم که کمک میکنند برنامه را کنترل کنیم. از این ساختارها برای کنترل ورودی و خروجی ها، و همچنین کنترل مقادیر و تعیین نتایج، استفاده میکنیم. در کل، چند ساختار کنترلی ساده در روبی وجود دارند که در این فصل بررسی خواهند شد.

ساختار if

این ساختار، ساده ترین ساختار شرطی در روبی است که تنها در صورت درست بودن یک شرط، یک نتیجه به ما میدهد. به این شکل می توان از این ساختار استفاده کرد :

```
if CONDITION
  STATEMENT
end
```

برای مثال، یک عدد را از ورودی خوانده ایم و میخواهیم بررسی کنیم زوج است یا خیر :

```
if n%2 == 0
 puts "#{n} is even"
end
```

ساختار else

گاهی، باید در صورت غلط بودن شرطمان نیز نتیجه ای تعیین کنیم (برای مثال موقع چک کردن شماره شناسه، رمز عبور و ...) . برای این کار، از ساختار else استفاده میکنیم :

```
if n%2 == 0
 puts "#{n} is even"
else
 puts "#{n} is odd"
end
```

برنامه نمونه

در اینجا، برنامه ای مینویسیم که چک میکند یک عدد زوج است یا فرد، و عدد را توسط کیبرد و با پیامی مناسب، از کاربر میگیرد و سپس نتیجه را با یک پیام مناسب به اطلاع کاربر میرساند

```ruby
print "Enter a number: "
n = gets.chomp
n = n.to_i #از آنجایی که ورودی به صورت رشته دریافت می شود، باید به عدد صحیح تبدیل شود
if n%2 == 0
 puts "#{n} is even"
else
 puts "#{n} is odd"
end
```

بررسی بیش از یک شرط

در ساختار مرسوم، با استفاده از if میتوانستیم در صورت درست بودن شرط، نتیجه ای برگردانیم و همچنین میتوانستیم توسط else در صورت غلط بودن شرط، نتیجه ای را به کاربر برگردانیم. اما، گاهی اوقات لازم است بیشتر از یک شرط را بررسی
برای این کار، دو روش در روبی وجود دارد.

روش اول : استفاده از elsif

این روش، روش مرسوم تر استفاده از چند شرط است. برای مثال، میخواهیم چک کنیم که عدد ما در رنج هایی که مد نظر داریم هست یا خیر. در نظر بگیرید چهار گروه سنی با چهار حرف اول الفبای انگلیسی داریم :

```
A : 0-5
B : 5-10
C : 10-18
D : 18 and higher
```

و در صورت ورود رقم ها این پیام ها چاپ شود :

```
A : Your child is too young for our products
B : Your child can use our blue products
C : Your child can use our purple products
D : You (or your child) can use our gold products.
```

برنامه ما به این شکل خواهد شد :

```ruby
if age > 0 || age < 5
 puts "Your child is too young for our products"
elsif age > 5 || age < 10
 puts "Your child can use our blue products"
elsif age > 10 || age < 18
 puts "Your child can use our purple products"
else
 puts "You (or your child) can use our gold products"
end
```

نکته

در اینجا، از یک عملگر منطقی استفاده شد، که توضیح آن چنین است : از عملگر `||` زمانی استفاده میکنیم که درستی یکی از بخشهای شرط برای ما کافیست. از عملگر `&&` زمانی استفاده می شود که همه بخش های شرط ما باید درست باشند. از عملگر `!` نیز زمانی استفاده می شود که نقیض شرط برای ما مهم باشد.

روش دوم: استفاده از case

این روش، ساده تر از روش قبلی است، و معمولا فقط زمانی که چندین شرط داشته باشیم، از این روش استفاده میکنیم. روش کلی case به این شکل است :

```
case VARIABLE
 when CONDITION
   STATEMENT
 else
   STATEMENT
 end
```

حال بیایید مثال قبلی را بار دگر ، با استفاده از case بنویسیم :

```
case age
 when 0...5
    puts "Your child is too young for our products"
 when 5...10
    puts "Your child can use our blue products"
 when 10...18
    puts "Your child can use our purple products"
 else
    puts "You (or your child) can use our gold products"
end
```

نکته

در اینجا، از یک نوع داده خاص به اسم «رنج» استفاده کردیم. رنج ها به دو دسته هستند. دسته اول، رنج هایی هستند که کران بالا عضو آنها نیست و این شکل نمایش داده می شوند:

```
0...10
```

و رنجهایی که کران بالا نیز عضوشان است که به این شکل نشان داده می شوند :

```
0..10
```

جمع بندی

با استفاده از دانشی که تا اینجا به دست آورده اید، میتوانید برنامه های تقریبا کاربردی ای بنویسید، برنامه هایی که بتوانند از کاربر ورودی بگیرند، و ورودی را دستخوش تغییر کنند، سپس نتیجه تغییرات را با یک ساختار کنترلی چک کنند و نتیجه مطلوب را به کاربر بازگردانند. با استفاده از فصل های بعدی، به کاربردهای برنامه هایتان نیز افزوده می شود، چرا که اصلی ترین مفاهیم را در فصل های بعدی فرا خواهید گرفت.

حلقه های تکرار

در فصل قبلی، با ساختار های کنترلی روبی آشنا شدیم. با استفاده از ساختارهای کنترلی توانستیم برنامه های هدفمند تر و کنترل شده تری را عرضه کنیم. در این فصل، با حلقه های تکرار آشنا خواهیم شد. گاهی نیاز است یک عملیات ریاضی، منطقی یا مقایسه ای، چندین بار روی یک ورودی انجام شده و ما نیز نیاز داریم تا هر بار، خروجی را بررسی کنیم. حلقه های تکرار، در این جا به درد ما خواهند خورد. همچنین، برای محاسبات پیچیده ریاضی که نیاز داریم تا یک عدد را چندین بار دچار دستخوش کنیم، حلقه های تکرار بسیار کاربرد خواهند داشت. برای ساخت یک منو در کنسول هم می توان از حلقه های تکرار استفاده نمود. در اینجا، با سه حلقه پرکاربرد در روبی آشنا می شویم.

حلقه while

این حلقه، پر کاربرد ترین حلقه در زبان روبی است. روش استفاده از این حلقه به این شکل است :

```
while CONDITION
 STATEMENT
end
```

فرض کنیم میخواهیم یک برنامه بنویسیم که اعداد یک تا ده را برای ما چاپ کند، باید به این شکل عمل کنیم :

```
n = 1
while n <= 10
  puts n
  n += 1
end
```

بعد از اجرای این حلقه، اعداد یک تا ده در ترمینال نمایان می شوند.

تجزیه و تحلیل برنامه

در ابتدای برنامه گفتیم `n=1` یا به عبارتی عملیات initialization را انجام دادیم. سپس یک حلقه را تعریف کردیم و شرط آن را `n <= 10` تعیین کردیم. در واقع تا زمانی که n مقدارش کوچکتر یا مساوی ۱۰ باشد، حلقه اجرا خواهد شد. سپس درون حلقه فرمان چاپ را صادر کردیم. برای این که حلقه در یک جای مشخص پایان یابد، یک شمارنده به صورت `n += 1` نیز معرفی نمودیم. وظیفه شمارنده این است که کنترل کند حلقه حتما تمام شود.

چاپ اعضای یک آرایه با حلقه

اکنون، نوبت آن رسیده که کمی از حلقه ها بهتر استفاده کنیم. در این جا، قصد داریم با استفاده از حلقه while همه مقادیری که در یک آرایه ذخیره کرده ایم را چاپ کنیم :

```
n = 0
while n < array.length
  puts array[n]
  n += 1
end
```

این برنامه نیز مانند برنامه پیش است، با این تفاوت که مقدار n تاثیر مستقیمی در خروجی ندارد.

حلقه until

در این حلقه، بر خلاف while، شرط غلط به عنوان شرط به حلقه داده شده و حلقه، تا زمان درست شدن شرط ادامه خواهد یافت. برای مثال، برنامه چاپ اعداد را در نظر بگیرید، اکنون با استفاده از until این برنامه را باز نویسی میکنیم :

```
n = 0
until n > 10
 puts n
 n += 1
end
```

همانگونه که دیدید، شرط حلقه در اینجا شد `n > 10` در واقع حلقه تا زمانی ادامه میابد که مقدار n از عدد ده بزرگتر شود.

حلقه for

این حلقه، به صورت اختصاصی برای کار با ساختمان های داده طراحی شده است. فرمت کلی حلقه به این شکل است :

```
for VARIABLE in DATA_STRUCTURE
 STATEMENT
end
```

برای مثال، میخواهیم اعداد یک تا ده را با استفاده از این حلقه چاپ کنیم. تنها نیاز داریم از یک رنج بهره ببریم :

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

همچنین برای چاپ اعضای یک آرایه نیز میتوان به صورت مستقیم از این حلقه استفاده نمود :

```
for n in array
 puts n
end
```

متد each

متد each یک متد است که در آرایه ها و جداول درهم سازی، به کار رفته و عملکرد آن دقیقا حلقه for است. برای مثال اگر بخواهیم مقادیر موجود در یک آرایه را با این متد چاپ کنیم به این شکل عمل میکنیم :

```
array.each do |x|
 puts x
end
```

همچنین برای چاپ مقادیر درون یک جدول درهم سازی کافیست به این شکل عمل کنیم :

```
hash_table.each do |key, value|
 puts "#{key} : #{value}"
end
```

حلقه های بی نهایت

حلقه های بی نهایت، به حلقه هایی گفته می شود که شرطشان همیشه درست، یا همیشه غلط باشد. با استفاده از یک حلقه بی نهایت، میتوان توابع مختلف را صدا زد، با استفاده از تابع system دستورات سیستم عامل را بصورت پیوسته خواند و اجرا کرد و ... . برای پیاده سازی حلقه بی نهایت دو راه داریم :

حلقه بی نهایت با شرط همیشه درست

این حلقه را باید با while پیاده سازی کرد. فرم کلی آن به این شکل است :

```
while true
 ....
end
```

این حلقه تا زمانی ادامه خواهد داشت که سیگنال اتمام برنامه توسط `CTRL + C` توسط کاربر اجرا شود.

حلقه بی نهایت با شرط همیشه غلط

این حلقه نیز باید توسط until پیاده شود :

```
until false
 ....
end
```

تمامی توضیحات حلقه بی نهایت با شرط همیشه درست، برای این حلقه نیز صدق میکنند.

جمع بندی

تا اینجا، شما با دانشی که کسب کرده اید، قادر خواهید بود برنامه هایی خلق کنید که نیازی را از خودتان یا دوستانتان رفع کند. مفاهیم برنامه سازی ساخت یافته، مانند ساختارهای کنترلی و حلقه ها، اکنون در ذهن شما نقش بسته و آماده اید تا برنامه های خود را ماژولار تر از قبل بنویسید.

توابع

در این فصل، قصد داریم برنامه های خود را به چندین بخش تقسیم کرده، بخش هایی که لازم داریم را در مواقع نیاز صدا بزنیم. در واقع، قرار است یک برنامه به چند زیربرنامه تقسیم شود.

تابع چیست؟

قبل از این که به سراغ پیاده سازی توابع در روبی برویم، بیایید ببینیم که تابع چیست. یک تابع، یک رابطه از مجموعه A به B است که از هز عضو A دقیقا یک عضو از B نظیر می شود. به صورت کلی، و در شبه کد ها تابع را به این شکل نمایش میدهیم :

```
f(A) -> B
```

در برنامه نویسی نیز، دقیقا به همین شکل است. یک ورودی به تابع مورد نظر داده شده، سپس خروجی منحصر به فرد دریافت خواهیم کرد.

توابع در روبی

تابع در روبی، به این شکل تعریف میشود :

```
def funcName(ARGUMENTS)
 STATEMENTS
end
```

با این که برای نام توابع، محدودیتی جز این که با کاراکتر یا `_` شروع شوند نیست، ما قرار داد میکنیم که تابع ها را با نام کوچک نام گذاری کنیم.

توابع تهی یا void

این توابع، توابعی هستند که مقداری را بر نمیگردانند (از return در این توابع استفاده نمی شود). از این توابع، برای خواندن ورودی، یا چاپ خروجی استفاده میکنیم. برای مثال تابع زیر یک پیام ساده چاپ میکند :

```
def hello()
 puts "Hello, World!"
end
```

برای صدا زدن این تابع در کد، کافیست به این شکل نام تابع را در کد خود قرار دهیم :

```
hello()
```

تابع تهی با دریافت آرگومان

توابع تهی، گرچه مقداری را بر نمیگردانند، اما قابلیت دریافت آرگومان را دارند. برای مثال، تابع بالا را به شکل دیگری باز نویسی میکنیم :

```
def hello(name = "World")
 puts "Hello, #{name}!"
end
```

در این تابع، ما آرگومانی به نام name با مقدار پیشفرض World برای تابع تعریف کردیم. اگر تابع را به این شکل فراخوانی کنیم :

```
hello()
```

پیام Hello, World! چاپ خواهد شد. اما اگر نامی به آن بدهیم، برای مثال Muhammadreza ، به این شکل خواهد بود :

```
hello("Muhammadreza")
#=> Hello, Muhammadreza!
```

توابع غیرتهی

در این توابع، مقدار خروجی حتما باید برگردد. در واقع، در این توابع از دستور العمل return استفاده میکنیم تا نتیجه نهایی را به کاربر برگردانیم. فرض کنیم میخواهیم تابعی بنویسیم که سه عدد را با یکدیگر جمع میکند. تابع ما چنین شکلی خواهد داشت :

```
def addNums(a, b, c)
 return a + b + c
end
```

اگر این تابع را به این شکل، تنها با سه پارامتر صدا کنیم :

```
addNums(1, 2, 4)
```

اتفاقی رخ نمیدهد. دلیل آن نیز این است که تابع اجرا شده است، اما نتیجه آن را چاپ نکرده ایم. برای این که کد ما بتواند نتیجه مناسبی نیز به ما بدهد، باید این کار را بکنیم :

```
puts addNums(1, 2, 4)
```

به این شکل، عدد ۷ در خروجی نمایان می شود.

برنامه جمع اعداد بدون محدودیت در تعداد

اکنون قصد داریم برنامه ای بنویسیم که بتواند تعداد نامحدودی از اعداد را به عنوان ورودی دریافت کرده و سپس حاصل جمعشان را بر گرداند. برای دریافت تعداد زیادی ورودی، و در قالب یک آرایه تنها کافیست این چنین عمل کنیم :

```
def funcName(*args)
 STATEMENTS
end
```

پس ، مشخص شد که شکل کلی تابع ما چگونه است. اکنون، نوبت آن است که با استفاده از یک حلقه، حاصل جمع اعضای یک آرایه را حساب کرده و در متغیری برگردانیم.

```
def addNums(*nums)
  sum = 0
  for i in nums
   sum += i
  end
  return sum
end
```

برای مثال، خروجی این تابع به اعضای ۳ مقدار ۱ و ۱و ۳ ، عدد ۵ خواهد بود. استفاده از این روش، به ما کمک میکند که بتوانیم یک عملیات خاص را بدون محدودیت در تعداد ورودی ها، انجام دهیم.

توابع بازگشتی

این توابع، توابعی هستند که خروجی آنها، بستگی به عملیات هایی دارد که قبل از آن انجام شده است. معروف ترین توابع بازگشتی، توابع فاکتوریل و فیبوناچی هستند. گرچه، اگر نیاز باشد که تابع درون خودش، صدا زده شود، آن تابع بازگشتی است و نیازی نیست که حتما از توابع بازگشتی معروف باشد.

تابع چاپ یک متن

فرض کنیم میخواهیم یک صفحه را با یک متن پر کنیم، صرفا برای این که تست کنیم متن چه رفتاری در صفحه ما دارد و یک نمونه داشته باشیم. برای نوشتن چنین متنی، کافیست این گونه عمل کنیم :

```
def _printLines(n)
 if n > 0
  puts "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur semper."
  _printLines(n - 1)
 end
end
```

در این تابع، به این شکل عمل می شود، که ابتدا بررسی میکند n از صفر بزرگتر است یا خیر. در صورت بزرگتر بودن، آنقدر این تابع صدا زده می شود که مقدار n با صفر برابر شود. در این صورت است که متنی که به تابع داده ایم، به تعداد دلخواه ما چاپ می شود.

تابع فاکتوریل

این تابع نیز از معروف ترین توابع بازگشتی است. تعریف ریاضی این تابع به این شکل است :

```
f(n) -> n * f(n - 1)
```

اکنون، بیایید این تابع را در روبی بازنویسی کنیم. باید در نظر داشته باشیم که صفر فاکتوریل، به صورت قراردادی، برابر با یک در نظر گرفته می شود. پس تابع ما به این شکل خواهد بود :

```
def factorial(n)
 if n == 0
  return 1
 else
  return n * factorial(n - 1)
 end
end
```

در اینجا نیز، همانند قبل، چک می شود که آیا مقدار n برابر صفر است یا خیر. سپس آنقدر این تابع صدا زده می شود که مقدار n صفر شده، و سپس ضرب ها انجام میگیرند.

جمع بندی

در این فصل، فرا گرفتیم که چگونه برنامه های خود را ماژولار تر و بهینه تر بنویسیم. فایده این نوع برنامه نویسی این است که توابع نوشته شده توسط ما در یک برنامه ، ممکن است بعد ها در برنامه های دیگر خودمان یا دیگران به کار بیاید. به این شکل، میتوان یک کتابخانه خوب از توابع مورد نیاز تولید کرد تا در صورت نیاز، از آن استفاده کنیم. در فصلهای آتی، وارد مباحث پیشرفته تر و در نهایت کاربردهای روبی خواهیم شد.

برنامه نویسی پیشرفته در روبی

در این فصل، دیگر وارد بحث های حرفه ای و پیشرفته و البته قشنگ روبی می شویم. بحث هایی که می‌توانند شما را وادار به مطالعه و همچنین کد زدن بیشتر کنند. در این فصل، سعی ما بر این است که بر مفاهیم تئوری تمرکز کنیم و سپس، به سراغ کد زدن در فصل بعدی برویم.

برنامه نویسی شی گرا

اگر بخواهیم به ساده ترین شکل ممکن برنامه نویسی شی گرا را توضیح دهیم، باید بگوییم که برنامه نویسی با گرفتن ایده از دنیای اطراف، و شبیه سازی و مدل سازی از آنها است. در واقع، وقتی کد شی گرا مینویسیم، یک مفهوم کلی را در نظر میگیریم، سپس اشیا را از آن خارج میکنیم. برای مثال، مفهوم عمومی اتومبیل را در نظر بگیرید، و سپس اشیائی که از این مفهوم خارج می شوند را متصور شوید. برای مثال، یک اتومبیل سواری و یک کامیون، هر دو اتومبیل هستند و از تعریف کلی اتومبیل استخراج شده اند، هرچند متفاوتند.

مفاهیم شی گرایی

به طور کلی، در برنامه نویسی شی گرا، ما با این مفاهیم مواجهیم :

انتزاع
وراثت
چند ریختی
کپسوله سازی

هر کدام از این مفاهیم، در قسمتی از برنامه، به کار می آیند در ادامه با آن آشنا خواهیم شد.

انتزاع

همان مثال اتومبیل را در نظر بگیرید. مفهوم «اتومبیل» به خودی خود یک مفهوم انتزاعی است. و اتومبیل هایی که از آن ساخته می شوند، اشیا هستند. در برنامه نویسی، به مفهوم انتزاعی ساخته شده، «کلاس» گفته می شود.

وراثت

کلاس یا همان مفهوم انتزاعی ساخته شده «اتومبیل» را در نظر بگیرید. اکنون وقت آن است که ما «اتومبیل شخصی» و «کامیون» بسازیم. هم اتومبیل های شخصی و هم کامیون ها، ویژگی های کلاس والد، یعنی اتومبیل را دارند. اگرچه، فرق های اساسی با یکدیگر دارند، اما در نهایت، هر دو اتومبیل هستند. در واقع، یک فرزند هیچوقت دقیقا والد خود نیست، اما ویژگی هایی دارد که میتوان تشخیص داد که کدام کلاس و کدام شی، فرزند کدام کلاس است.

چند ریختی

چند ریختی بدین معناست که یکی از ویژگی هایی که در مفهوم انتزاعی داریم، به شکل های متفاوتی کار کند، در عین این که عمل یکسان انجام میدهد. مثلا در کلاس اتومبیل، ما یک فرمان داریم که وظیفه یکسانی دارد، اما نحوه کارکردش میتواند مکانیکی، الکتریکی، هیدرولیکی و ... باشد. همانگونه که ترمز یا لنت ترمز میتواند متفاوت باشد. همینطور موتور اتومبیل، وظیفه اش تبدیل انرژی شیمیایی به مکانیکی است، اما این عملیات به شکل های مختلفی انجام می شود. مثلا اتومبیل شخصی موتور بنزینی دارد که میتواند کاربراتوری یا انژکتوری باشد، و کامیون نیز از موتور دیزلی استفاده می کند. در واقع، اگر یک ویژگی مشترک داشته باشیم، که به طور متفاوت کار کند ولی نتیجه کارکردش یکسان باشد، از مفهوم چندریختی استفاده کرده ایم.

کپسوله سازی

کپسوله سازی، بدین معناست که عملکرد بخش های خاصی را از دید کاربر، مخفی کنیم. برای مثال اتومبیل، کارکرد موتور را در نظر بگیرید. بدون این که کاربر از عملکرد موتور آگاه باشد، موتور کار میکند. کاربر اتومبیل تنها اتومبیل را روشن میکند، توسط پدالها، فرمان، دسته دنده و سایر امکاناتی که می تواند از آن استفاده کند، اتومبیل را کنترل میکند. اما کاربر، از نحوه تزریق سوخت داخل سیلندر اطلاعی ندارد، بلکه سازنده اتومبیل این قسمت را هندل کرده است. کاربر از نحوه استفاده از سیالاتی که برای کنترل ترمز و فرمان استفاده می شود خبر ندارد. حتی موقع خرابی یا به وجود آمدن اشکال در کارکرد بخشی از اتومبیل، باید اتومبیل را به دست تعمیر کار همان اتومبیل بخصوص سپرد. در واقع هر گاه بخواهیم از پیچیدگی کار نرم افزار خود بکاهیم، بخش هایی از نرم افزار را به اصطلاح کپسوله میکنیم. با استفاده از این کار، کاربر پیچیدگی قسمت های اعظمی از برنامه ما را نمی بیند، و تنها بخشی که به خودش مربوط است را تحت کنترل می گیرد.

جمع بندی

در این فصل، با مفاهیم پایه ای شی گرایی آشنا شدیم. در فصل های آینده، کد های شی گرا خواهیم نوشت و سپس وارد مسائل کاربردی زبان روبی می شویم. تا اینجا، شما کانسپت های خوبی را در ذهن خود جای داده اید، و این بدین معنیست که از این پس میتوانید برنامه های واقعا کاربردی بنویسید و منتشر کنید.

برنامه نویسی شی گرا در روبی

در فصل قبلی از نظر تئوری، با مسائل مربوط به برنامه نویسی شی گرا، آشنا شدیم. در اینجا، کمی به مبحث وارد تر شده، و تقریبا تمام مفاهیمی که در فصل گذشته فرا گرفتیم، به صورت کد در آمده و پیاده سازی خواهد شد.

ایجاد یک کلاس

در اینجا، میخواهیم یک مفهوم انتزاعی تولید کنیم. برای مثال، همان کلاس اتومبیل را در نظر بگیرید ، کلاس ایجاد شده به این شکل خواهد شد :

```
class Automobile
...
end
```

دقت کنید، نام کلاس ها حتما باید با حرف بزرگ شروع شود. اما کلاس ما یک چیز کم دارد. با ساختن یک کلاس، معمولا متدی برای مقداردهی اولیه به آن، ساخته می شود :

```
class Automobile
 def initialize(name)
   @name = name
 end
end
```

ایجاد یک شیء از یک کلاس

اکنون، ما یک کلاس کلی اتومبیل داریم، بیاید یک شیء از این کلاس ایجاد کنیم :

```
pride = Automobile.new("Pride")
```

به این شکل، یک اتومبیل جدید به اسم Pride ایجاد کردیم. یادتان باشد که متد new توسط خود روبی برای کلاس ها در نظر گرفته شده،و پارامترهای آن به متد initalize که توسط ما تعریف شده است، پاس داده می شود.

اضافه کردن متد به کلاس

متد، روش های مختلفیست که یک کلاس و در نتیجه یک شیء از یک کلاس، قادر به انجام آن است. در اینجا، میخواهیم یک متد اضافه کنیم که وقتی یک عدد را از ورودی دریافت میکند، در خروجی میزان حرکت به همان اندازه را به کاربر نشان دهد. نتیجتا کلاس به این شکل خواهد شد :

```
class Automobile
 def initialize(name)
   @name = name
 end

 def move(distance)
   puts "#{@name} moved about #{distance} kilometers"
 end

end
```

بگذارید به شی خود برگردیم. وقتی چنین چیزی را فراخوانی میکنیم :

```
pride.move(50)
```

در خروجی باید چنین چیزی تحویل بگیریم :

```
Pride moved about 50 kilometers
```

در واقع، اینجا مشخص می شود که خودروی تعریف شده، چقدر حرکت کرده است.

ایجاد یک کلاس فرزند

همانگونه که در فصل پیش نیز گفته شد، یکی از ویژگی های شی گرایی، ویژگی ارث بری یا Inheritance است. در واقع، ما یک کلاس و یک مفهوم انتزاعی کلی در نظر میگیریم، سپس از آن یک سری مفهوم دیگر مشتق میکنیم. اکنون، بیاید کلاس کامیون را تعریف کنیم :

```
class Truck < Automobile
 ...
end
```

دو متد initalize و move اکنون به این کلاس ما، منتقل شده اند. ولی برای مثال، برای کامیون نیاز است تا ظرفیتش دانسته شود. پس یک متد برای ست کردن ظرفیت در کامیون خود، تعریف میکنیم :

```
class Truck < Automobile
 set_capacity(cap)
  @cap = cap
 end
end
```

اما، همچنان، ممکن است ندانیم که این کامیون، چه قدر ظرفیت دارد و حتی نامش چیست. پس یک متد دیگر مینویسیم :

```
class Truck < Automobile
 set_capacity(cap)
  @cap = cap
 end
 def print_info
  puts "Name : #{@name}"
  puts "Capacity : #{@cap} tons"
 end
end
```

اکنون، یک شیء کامیون جدید میسازیم :

```
scania = Truck.new("Scania")
scania.set_capacity(20)
```

حال میتوانیم با استفاده از متد print_into مشخصات کامیون را ببینیم. که نتیجه کد ما چنین است :

```
Name : Scania
Capacity : 20 tons
```

توجه کنید که متد move نیز برای این شیء همچنان معتبر است.

چندریختی

در این قسمت، ویژگی «چند ریختی» از برنامه نویسی شی گرا را بررسی خواهیم کرد. بیایید ببینیم که اصلا چندریختی یعنی چه؟ همانطور که در فصل پیش گفته شد، چند ریختی دقیقا به معنای آن است که یک شی، به چند شکل مختلف ساخته شود، اما در نهایت همان کارکرد را داشته باشد. بیایید ابتدا یک کلاس Parser را در نظر بگیریم. این کلاس را ابتدا به این شکل مینویسیم :

```
class Parser
 def parse
  raise NotImplementedError, 'You need to define this method'
 end
end
```

اگر از این کلاس، یک شی ایجاد کنیم و سپس متد parse را فراخوانی کنیم، مفسر روبی به ما یک ارور خواهد داد و از ما خواهد خواست که متد parse را تعریف کنیم. اکنون ما یک Parser برای JSON از روی این کلاس مینویسیم :

```
class JsonParser < Parser
 def parse
  puts 'JSON parser received the parse message'
 end
end
```

کپسوله سازی

کپسوله سازی یا Encapsulation بدین معنیست که کارکرد بخش های درونی یک کلاس را از دید کاربر مخفی کنیم. در واقع، این کار برای آسایش بیشتر کاربر است. برگردیم به مثال فصل گذشته، اگر فرض را بر این بگیریم که شما قرار باشد خودتان عملکرد کلاچ را در خودرویتان داشته باشید، تقریبا استفاده از خودرو غیرممکن خواهد شد. اما سازندگان خودرو، یک فرایند پیچیده را از دید شما مخفی کرده اند و تنها کاری که شما برای انجام آن نیاز دارید تا انجام دهید، آن است که یک پدال را فشار دهید. حالا ببینیم که کپسوله سازی به چند شکل قابل انجام است؟

متدهای خصوصی

متدهای خصوصی ، یا private متدهایی هستند که خود برنامه به آن ها دسترسی دارد. میتوانند از درون متدهای دیگر هم میتوانند این متدها را درون خود صدا بزنند. برای مثال :

```
class Test
 def initialize()
  private_method
 end

 private
 def private_method
  puts "Private method called"
 end
end
```

متدهای خصوصی، باید درون متدهای دیگر صدا زده شوند. به این شکل، ما میتوانیم از آنها استفاده کنیم، بدون آن که کاربر متوجه شود چنین متدهایی نیز وجود داشته اند.

متدهای محافظت شده

متدهای محافظت شده یا protected هم متدهایی هستند که خارج از کلاس قابل استفاده نیستند و باید در متدهای دیگر استفاده شوند. برای استفاده از این متد ها، باید از کلمه کلیدی `self` استفاده کنیم.

```
class Test
 def initalize()
  self.protected_method
 end

 protected
 def protected_method
  puts "Protected method called!"
 end
end
```

به این شکل، با فراخوانی متد مربوطه، متد محافظت شده نیز اجرا می شود.

جمع بندی

در این فصل، تمام موضوعاتی که در فصل گذشته توضیح داده شد را به صورت کاربردی و برنامه نویسی توضیح دادیم. اکنون، شما دیگر میتواند برنامه های کاربردی را توسط روبی توسعه دهید. از این به بعد، محتوای کتاب بیش از آن که به مسائل برنامه نویسی روتین بپردازد، به مسائل کاملا کاربردی در روبی خواهد پرداخت. شما در فصل آینده، کمی از تکنیک های شی گرایی را فرا خواهید گرفت و سپس، با کاربرد روبی در وب آشنا خواهید شد.