Ruby Kullanıcı Kılavuzu

Çeviren:
Pınar Yanardağ
<pinar (at) comu.edu.tr>

Yazan:
Mark Slagell
<slagell (at) ruby–lang.org>

22 Nisan 2003

Özet

Ruby, nesneye yönelik, kolay bir dildir. İlk başta tuhaf bir dil gibi görünse de, kolayca yazılıp okunmak için tasarlanmıştır. Bu kullanıcı kılavuzu, ruby'yi ilk defa kullanacaklara ruby'nin doğası hakkında fikir vermek amacıyla yazılmıştır, kesinlikle bir dil başvuru kılavuzu değildir.

Konu Başlıkları

Geçmiş

1.0	Temmuz 2005	PY
Çevirinin ilk sürümü.		
-	30 Mart 2003	MS
Özgün belge Japonca'dır. İngilizce sürümünü http://www.rubyist.net/~slagell/ruby/ adresinde bulabilirsiniz.		

Yasal Açıklamalar

1. Ruby Nedir?

Ruby 'hızlı ve kolay', nesneye yönelik yazılım geliştirmeye yarayan yorumlanan bir betik dilidir.

Peki bu ne anlama gelmektedir?

Yorumlanan betik dili:

- Doğrudan işletim sistemi çağrılarını yapabilme yeteneği
- Güçlü dizge işlemleri ve düzenli ifadeler
- Geliştirme sırasına anında geribesleme

Kolay ve hızlı:

- Değişken bildirimleri gerekmez
- Değişken türleri yoktur
- Sözdizimi basit ve tutarlıdır
- Bellek yönetimi özdevinimlidir

Nesneye dayalı olmak:

- Herşey birer nesnedir
- Sınıflar, kalıtım, yöntemler, vs.
- Tekil yöntemler
- Modüllerle çalışılabilir
- Yineleyiciler ve sonlandırıcılar

Ayrıca:

- Çoklu duyarlıklı tamsayılar
- Olağandışılık işleme modeli
- Özdevimli yükleme
- Evreler

Yukardaki kavramların bazılarına yabancıysanız, endişelenmeyin, okumaya devam edin. Ruby dili çabuk ve kolay öğrenilir.

2. Giriş

Öncelikle sisteminizde Ruby'nin kurulu olup olmadığına bakalım. Kabuk istemcisinden (burada "**$**" ile temsil edilmiştir o yüzden **$** işaretini yazmanıza gerek yok) aşağıdaki kodu yazalım:

```
$ ruby -v
```

(**-v** ruby'nin sürüm bilgilerini yazmasını sağlar), ve sonra da Enter tuşuna basalım. Eğer sisteminizde Ruby kuruluysa aşağıdakine benzer bir satır görmeniz gerekecek:

```
ruby 1.6.6 (2001-12-26) [i586-linux]
```

Eğer Ruby yüklü değilse, sistem yöneticinizle görüşebilir ya da kendiniz kurabilirsiniz.

Artık Ruby ile oynamaya başlayabiliriz. **-e** seçeneği ile Ruby yazılımlarını doğrudan komut satırına yerleştirebilirsiniz:

```
$ ruby -e 'print "merhaba dunya\n"'
merhaba dunya
```

Daha uzlaşımsal olarak bir Ruby yazılımı bir dosyada saklanabilir.

```
$ cat > test.rb
print "merhaba dunya\n"
^D
$ cat test.rb
print "merhaba dunya\n"
$ ruby test.rb
merhaba dunya
```

^D, control-D'yi ifade eder. Yukarıdakiler sadece UNIX ve türevleri için geçerlidir. Eğer DOS kullanıyorsanız şunu deneyin:

```
C:\ruby>  copy con: test.rb
print "merhaba dunya\n"
^Z
C:\ruby>  type test.rb
print "merhaba dunya\n"
C:\ruby>  ruby test.rb
merhaba dunya
```

Daha kapsamlı yazılımlar geliştirirken, muhtemelen gerçek bir metin düzenleyiciye ihtiyaç duyacaksınız!

Bazen şaşırtıcı biçimde karmaşık ve kullanışlı yazılımlar komut satırına sığabilecek minyatür yazılımlarla yapılabilmektedir. Örneğin aşağıdaki yazılım, çalışılan dizindeki tüm C kaynak ve başlık dosyalarında bulunan **foo**'ları **bar** ile değiştirir ve orjinal dosyaların .bak uzantısıyla yedeklerini alır.

```
% ruby -i.bak -pe 'sub "foo", "bar"' *.[ch]
```

Bu yazılım UNIX'in **cat** komutu gibi çalışır (ama **cat**'ten daha yavaş çalışır):

```
$ ruby -pe 0 file
```

3. Basit Örnekler

Şimdi faktöriyel hesabı yapan basit bir işlev yazalım. **Faktöriyel**'in matematiksel tanımı şöyledir:

```
n! = 1                (n==0 ise)
   = n * (n-1)!       (aksi taktirde)
```

Ruby'de bunu aşağıdaki gibi yazabiliriz:

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

Tekrarlanan **end** deyiminin varlığını fark etmiş olmalısınız. Sırf bu yüzden Ruby "Algol benzeri" olarak anılır. (Aslında Ruby'nin sözdizimi Eiffel dilini daha çok andırmaktadır.) Ayrıca **return** deyiminin eksikliğini de hissetmiş olmalısınız. Bu deyim Ruby için gereksizdir çünkü bir Ruby işlevi değerlendirdiği en son şeyi geri döndürür. **return** deyimi kullanılabilir ancak gereksizdir.

Şimdi faktöriyel işlevimizi deneyelim. Ekleyeceğimiz bir satır, bize çalışan bir yazılım sunacaktır:

```
# Sayının faktöriyelini bulan yazılım
# fact.rb olarak kaydedin
```

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

print fact(ARGV[0].to_i), "\n"
```

Burada **ARGV** komut satırı argümanlarını içeren bir dizidir ve **to_i** alınan bir dizgeyi tamsayıya çevirmeye yarar.

```
$ ruby fact.rb 1
1
$ ruby fact.rb 5
120
```

Acaba argüman olarak 40 versek çalışır mı? Muhtemelen hesap makineniz taşma hatası verecektir...

```
$ ruby fact.rb 40
815915283247897734345611269596115894272000000000
```

Bu çalışır, üstelik Ruby makinenizin belleğinin izin verdiği kadar tamsayıyla da çalışabilir. 400 için aşağıdaki gibi bir çıktı alacaksınız:

```
$ ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325998055516106359423769613992
31917134063858996537970147827206606320217379472010321356624
61380907794230459736069956759583609615871512991382228657857
95493616176544804532220078258184008484364155912294542753848
03558374518022675900061399560145595206127211192918105032491
00800000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000
```

Sonucun doğruluğunu kontrol edemeyiz ama öyle olması gerekiyor `:)`.

Girdi/Değerlendirme döngüsü

Eğer Ruby'yi hiçbir argüman olmadan çağırırsanız, komutları standart girdi olarak alır ve girdinin bitiminde komutları çalıştırır:

```
$ ruby
print "merhaba dunya\n"
print "hoscakal dunya\n"
^D
merhaba dunya
hoscakal dunya
```

Ruby, **eval.rb** adında bir araç ile gelir. Bu, ruby kodunuzu etkileşimli olarak klavyeden alan ve yazdıklarınızı gösteren bir araçtır. Bu öğreticinin devamında sıkça kullanacağız.

Eğer ANSI uyumlu bir uçbirimle çalışıyorsanız[1], ek olarak girintileme yapabilen, kodu renklendirebilen ve uyarılar veren gelişmiş eval.rb[B4]'yi kullanmalısınız. Aksi takdirde, ruby dağıtımının `sample` dizininde bulunan ve ANSI uyumluluğu gerektirmeyen sürümünü kullanabilirsiniz. Kısa bir **`eval.rb`** oturumu:

```
$ ruby eval.rb
ruby> puts "Merhaba dünya.\n"
Merhaba dünya.
   nil
ruby> exit
```

`Merhaba dünya`, `puts` ile çıktılandı. Sonraki satırdaki **nil** ise son değerlemenin sonucunu göstermektedir; ruby, deyimlerle ifadeler arasında bir ayrım yapmaz, yani kod parçalarının değerlendirilmesi temel olarak kodun icra edilmesidir. Burada **nil**, `print`'in anlamlı bir değer döndürmediğini ifade etmektedir. Son olarak, **exit** yazarak yorumlayıcı döngüsünü sonlandırıyoruz.

Bu kılavuzun devamında `ruby>` bir **`eval.rb`** oturumunun girdi istemi olarak kullanılmış olacaktır.

4. Dizgeler

Ruby'de dizgelerle sayısal bir veriymiş gibi işlem yapabilirsiniz. Bir dizge çift ("...") ya da tek ('...') tırnaklı olabilir.

```
ruby> "abc"
  "abc"
ruby>  'abc'
  "abc"
```

Bazı durumlarda çift ve tek tırnak farklı işlevler görür. Çift tırnaklı bir dizge tersbölü öncelemeli karakterleri kullanmayı ve **#{}** kullanan gömülü ifadeleri çalıştırmayı mümkün kılar. Halbuki, tek tırnaklı dizgelerle bunlar yapılamaz: ne görürseniz onu alırsınız. Örneğin:

```
ruby>  print "a\nb\nc","\n"
a
b
c
  nil
ruby> print 'a\nb\n',"\n"
a\nb\nc
  nil
ruby> "\n"
  "\n"
ruby> '\n'
  "\\n"
ruby> "\001"
  "\001"
ruby> '\001'
  "\\001"
ruby> "abcd #{5*3} efg"
  "abcd 15 efg"
ruby> var = " abc "
  " abc "
ruby> "1234#{var}5678"
  "1234 abc 5678"
```

Ruby'nin dizge işlemleri C'ye kıyasla daha esnek ve şıktır.Örneğin **+** ile iki dizgeyi birleştirebilirsiniz ya da ***** ile bir dizgeyi birçok kez tekrar ettirebirsiniz:

```
ruby> "foo" + "bar"
```

```
   "foobar"
ruby> "foo" * 2
   "foofoo"
```

Dizgeleri birleştirme işi C'de, doğrudan bellek yönetimi nedeniyle oldukça yakışıksızdır:

```
char *s = malloc(strlen(s1)+strlen(s2)+1);
strcpy(s, s1);
strcat(s, s2);
/* ... */
free(s);
```

Ruby kullandığımız için dizgelere herhangi bir alan ayırmamıza gerek yok. Bellek yönetimi açısından tamamen özgürüz.

Aşağıda dizgelerle yapabileceğiniz birkaç örnek var:

Örnek 1. Birleştirme

```
ruby> word = "fo" + "o"
  "foo"
```

Örnek 2. Tekrarlatma

```
ruby> word = word * 2
  "foofoo"
```

Örnek 3. Karakterler seçimi

Ruby'de karakterlerin tamsayı olduğuna dikkat edelim:

```
ruby> word[0]
  102             # 102, 'f' harfinin ASCII kodudur.
ruby> word[-1]
  111             # 111 'o' harfinin ASCII kodudur.
```

(Negatif indisler dizgenin başlangıcı yerine sonundan itibaren konumlanır.)

Örnek 4. Altdizge seçimi

```
ruby> herb = "parsley"
  "parsley"
ruby> herb[0,1]
  "p"
ruby> herb[-2,2]
  "ey"
ruby> herb[0..3]
  "pars"
ruby> herb[-5..-2]
  "rsle"
```

Örnek 5. Aynılığın sınanması

```
ruby> "foo" == "foo"
  true
ruby> "foo" == "bar"
```

```
  false
```

Bilgi

ruby–1.0 sürümünde yukarıdaki sonuçlar büyük harflidir (TRUE gibi).

Şimdi bu özelliklerin bazılarını hayata geçirelim: Bulmacamız bir "kelimeyi bil" bulmacası ama sanırım "bulmaca" kelimesi fazla mütevazi oldu **;)**

```
# guess.rb olarak kaydedin
words = ['kestane', 'gurgen', 'palamut']
secret = words[rand(3)]

print "tahmin et? "
while guess = STDIN.gets
  guess.chop!
  if guess == secret
    print "Bildin!\n"
    break
  else
    print "Üzgünüm kaybettin.\n"
  end
  print "tahmin et? "
end
print "Kelime ", secret, ".\n"
```

Şimdilik kodun ayrıntıları hakkında fazla kafa yormayalım. Aşağıda yazılımın nasıl çalışması gerektiği görülüyor:

```
$ ruby guess.rb
tahmin et? kestane
Üzgünüm kaybettin.
tahmin et? gurgen
Üzgünüm kaybettin.
tahmin et? ^D
Kelime palamut.
```

(1/3 olasılığa karşı biraz daha iyi yapmalıydım...)

5. Düzenli İfadeler

Şimdi daha ilginç bir yazılım geliştirelim. Bu sefer bir dizgenin verilen bir ***şablona*** uyup uymadığını araştıralım:

Bu şablonlar özel anlamları olan bazı karakterler ve karakter birleşimlerinden oluşur:

Düzenli ifade işleçleri ve anlamları

[]	aralık belirtimi (Örn, `[a-z]`, `a` ile `z` arasındaki bir harfi belirtir.)
\w	harf ya da rakam; `[0-9A-Za-z]` ile aynı
\W	ne harf ne de rakam
\s	boşluk karakteri; `[ \t\n\r\f]` ile aynı
\S	boşluklar dışında herhangi bir karakter
\d	rakam; `[0-9]` ile aynı
\D	rakamlar dışında herhangi bir karakter
\b	tersbölü (0x08) (sadece herhangi bir aralık belirtilmişse)
\b	kelime içi sınır belirtimi (aralık belirtiminin dışındayken)
\B	kelime dışı sınır belirtimi

`*`	önceledi̇ği ifadeyi sıfır ya da daha fazla tekrarlar
`+`	önceledi̇ği ifadeyi bir ya da daha fazla tekrarlar
`{m,n}`	önceledi̇ği ifadeyi en az m en çok n kez tekrarlar
`?`	önceledi̇ği ifadeyi en fazla bir kere tekrarlar; `{0,1}` ile aynı
`|`	önündeki veya ardındaki ifade eşleşebilir
`()`	gruplama işleci

Bu ilginç lügat genelde ***düzenli ifadeler*** olarak anılır. Ruby'de, Perl'de de olduğu gibi çift tırnak koymak yerine ters bölü işareti kullanılır. Eğer daha önce düzenli ifadelerle karşılaşmadıysanız muhtemelen *düzenli* hiç birşey göremeyeceksiniz ancak alışmak için biraz zamana ihtiyacınız olduğunu unutmayın. Düzenli ifadeler, metin dizgeleri üzerinde arama, eşleştirme ve bu gibi diğer işlerle uğraşırken sizi baş ağrısından (ve satırlarca koddan) kurtaran gözle görülür bir güce sahiptir.

Örneğin aşağıdaki tanıma uyan bir dizge aradığımızı farzedelim: "Küçük f harfiyle başlayan, ardından bir büyük harf gelen bundan sonra küçük harf haricinde herhangi bir karakterle devam eden" bir dizge. Eğer deneyimli bir C yazılımcısıysanız muhtemelen şimdiden kafanızca binlerce satır kod yazmıştınız, öyle değil mi? Kabul edin, kendinize güçlükle yardım edebilirsiniz. Ancak Ruby'de dizgeyi sadece şu düzenli ifadeyle sınamanız yeterli olacaktır: **`/^f[A-Z](^[a-z])*$/`**.

Köşeli parantezler içindeki bir onaltılık sayıya ne dersiniz? Hiç sorun değil.

```
ruby> def chab(s)    # "parantezler içinde onaltılık içerir"
    |    (s =~ /<0(x|X)(\d|[a-f]|[A-F])+>/) != nil
    | end
   nil
ruby>   chab "Bu değil."
   false
ruby>   chab "Belki bu? {0x35}"    # kaşlı ayraçlar kullanılmamalıydı
   false
ruby>   chab "Ya da bu? <0x38z7e>"    # onaltılık sayı değil
   false
ruby>   chab "Tamam, bu: <0xfc0004>."
   true
```

Düzenli ifadeler başlangıçta alengirli gibi gözükse de kısa süre içinde istediğinizi yapabilme konusunda yol katedeceksiniz.

Aşağıda düzenli ifadeleri anlamanıza yarayacak küçük bir yazılım bulunuyor. **`regx.rb`** olarak kaydedin ve komut satırına **`ruby regx.rb`** yazarak çalıştırın.

```
#ANSI terminal gerektirir!

st = "\033[7m"
en = "\033[m"

while TRUE
  print "str> "
  STDOUT.flush
  str = gets
  break if not str
  str.chop!
  print "pat> "
  STDOUT.flush
  re = gets
  break if not re
```

```
  re.chop!
  str.gsub! re, "#{st}\\&#{en}"
  print str, "\n"
end
print "\n"
```

Yazılım bir tanesi dizge diğeri de düzenli ifade olmak üzere iki girdi alır. Dizge verilen düzenli ifade ile sınanır ve bütün uyuşan sonuçlar listelenir. Şu an ayrıntılara ilgilenmeyin, bu kodun analizini daha sonra yapacağız.

```
str> foobar
pat> ^fo+
foobar
~~~
```

Programın çıktısında gördüğünüz `~~~` ile işaretli parça çıktıda artalan ve önalan renkleri yerdeğiştirmiş olarak çıktılanır.

Bir kaç girdi daha deneyelim.

```
str> abc012dbcd555
pat> \d
abc012dbcd555
   ~~~   ~~~
```

Eğer şaşırdıysanız sayfanın başındaki tabloya tekrar göz atabilirsiniz: **`\d`**'nin **`d`** karakteriyle hiçbir bağlantısı yoktur, sadece bir rakamı eşleştirmekte kullanılır.

Eğer istediğimiz kriterlere uygun birden fazla yol varsa ne olur?

```
str> foozboozer
pat> f.*z
foozboozer
~~~~~~~~
```

Düzenli ifadeler olabilecek en uzun dizgeyi döndürdüğü için **`fooz`**'un yerine **`foozbooz`** eşleştirildi.

Aşağıda iki nokta üstüste işaretiyle sınırlandırılmış bir zaman alanı bulunuyor:

```
str> Wed Feb  7 08:58:04 JST 1996
pat> [0-9]+:[0-9]+(:[0-9]+)?
Wed Feb  7 08:58:04  JST 1996
           ~~~~~~~~
```

"**`=~`**" işleci bulduğu dizgenin konumunu döndüren, aksi halde **`nil`** döndüren düzenli ifadedir.

```
ruby> "abcdef" =~ /d/
  3
ruby> "aaaaaa" =~ /d/
  nil
```

6. Diziler

Ruby'de köşeli parantezler **`[]`** arasına elemanları yazarak ve virgüller yardımıyla ayırarak bir dizi oluşturabilirsiniz. Ruby'de diziler farklı nesne türlerini ayırdedebilecek niteliktedir.

```
ruby> ary = [1, 2, "3"]
  [1, 2, "3"]
```

Diziler de aynı dizgeler gibi birleştirilebilir ya da tekrar edilebilir.

```
ruby> ary + ["foo", "bar"]
  [1, 2, "3", "foo", "bar"]
ruby>  ary * 2
  [1, 2, "3", 1, 2, "3"]
```

Dizinin herhangi bir elemanına ulaşmak için indisleri kullanabiliriz.

```
ruby> ary[0]
  1
ruby> ary[0,2]
  [1, 2]
ruby> ary[0..1]
  [1, 2]
ruby> ary[-2]
  2
ruby> ary[-2,2]
  [2, "3"]
ruby> ary[-2..-1]
  [2, "3"]
```

(Negatif indisler dizinin sonundan başlanmasını sağlar.)

Diziler **join** kullanılarak dizgelere ve dizgeler **split** kullanılarak dizilere dönüştürülebilirler.

```
ruby> str = ary.join(":")
   "1:2:3"
ruby> str.split(":")
   ["1", "2", "3"]
```

Çırpılar

Çırpılar (hash) elemanlarına indisler yerine herhangi bir değer olabilen `anahtarlar` yardımıyla erişilebilen özelleştirilmiş dizilerdir. Böyle dizilere ***çırpı*** dendiği gibi ***isim–değer çiftleri*** de denir; biz ruby dünyasında çırpı deyimini tercih ederiz. Bir çırpı kaşlı ayraçlar arasına yazılarak oluşturulabilir. Dizilerde herhangi bir elemana ulaşmak için indisleri kullandığımız gibi çırpılarda elemana ulaşmak için anahtarları kullanırız.

```
ruby>  h = {1 => 2, "2" => "4"}
   {1=>2, "2"=>"4"}
ruby>  h[1]
   2
ruby> h["2"]
   "4"
ruby> h[5]
   nil
ruby> h[5] = 10      # değer ekleme
   10
ruby>  h
   {5=>10, 1=>2, "2"=>"4"}
ruby>  h.delete 1    # değer silme
   2
ruby>  h[1]
   nil
ruby>  h
   {5=>10, "2"=>"4"}
```

7. Örneklere Dönüş

Şimdi eski örneklere tekrar göz atalım.

Aşağıdakini daha önce *Basit Örnekler* (sayfa: 4) kısmında görmüştük.

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
print fact(ARGV[0].to_i), "\n"
```

Bu bizim ilk örneğimiz olduğu için her satırı teker teker açıklayalım.

Faktöriyeller

```
def fact(n)
```

İlk satırda bir işlev (ya da daha özel olarak bir ***yöntem***; yöntemin ne olduğunu ileriki kısımlarda göreceğiz) tanımlamak için **def** deyimini kullanıyoruz. Burada işlevimiz `fact`'ın `n` adında tek bir argüman aldığını görüyoruz.

```
if n == 0
```

if bir denetim deyimidir. Eğer koşul sağlanıyorsa onu takip eden kod değerlendirilir, aksi taktide **else** kısmına geçilir.

```
1
```

Eğer koşul sağlandıysa **if**'in değeri 1 olacaktır.

```
else
```

Sağlanmadıysa, bu deyimle başlayan **end** ile biten kod parçası değerlendirilir.

```
n * fact(n-1)
```

Yani, eğer koşul sağlanmamışsa sonuç **n** kere **fact(n-1)** olacaktır.

```
end
```

İlk **end**, **if** deyimini kapatmak için kullanılır.

```
end
```

İkinci **end** **def** ifadesini kapatmak için kullanılır.

```
print fact(ARGV[0].to_i), "\n"
```

Bu bizim komut satırından `fact()` işlevini çalıştırmamızı ve sonuçları ekranda görmemizi sağlar.

ARGV komut satırı argümanlarını içeren özel bir dizidir. **ARGV** dizisinin tüm elemanları dizgeler olduğu için, **to_i** yöntemiyle tamsayıya dönüştürmek zorundayız. Ruby Perl'deki gibi dizgeleri tamsayılara kendiliğinden dönüştürmez.

Hmmm... Eğer bu yazılıma negatif bir sayı girersek ne olur? Sorunu görebildiniz mi? Peki düzeltebilir misiniz?

Dizgeler

Dizgeler (sayfa: 6) bölümündeki bulmaca örneğimizi tekrar inceleyelim. Bu sefer biraz daha uzun, kolaylık açısından satırları numaralandırdık.

```
1 words = ['kestane', 'gurgen', 'palamut']
2 secret = words[rand(3)]
3
4 print "tahmin et? "
5 while guess = STDIN.gets
6   guess.chop!
7   if guess == secret
8     print "Bildin!\n"
9     break
10   else
11     puts "Üzgünüm kaybettin.\n"
12   end
13   print "tahmin et? "
14 end
15 puts "Kelime ", secret, ".\n"
```

Bu yazılımda yeni bir denetim yapısı gördük: **while**. Verilen koşul doğru olduğu sürece **while** ve **end** arasındaki kod tekrar tekrar çalıştırılacaktır.

2. satırdaki `rand(3)` işlevi 0 ile 2 arasında rastgele sayı üretir. Bu rastgele sayı `words` dizisinin elemanlarından birini çıkarmak için kullanılır.

5. satırda `STDIN.gets` yöntemiyle standart girdiden bir satır okuduk. Eğer satırı alırken `EOF` (end of file) karakterine rastlanırsa `gets` işlevi **nil** değerini döndürecektir. **while** ile ilişkilendirilmiş kod `^D` (ya da DOS altında ^Z) görene kadar tekrarlanacaktır.

6. satırdaki `guess.chop!` işlevi `guess` değişkeninin sonundaki satırsonu karakterini temizlemeye yarar.

15. satırda gizli kelimeyi yazdırıyoruz. Bunu üç argümanla birlikte bir yazdırma deyimi olarak kullandık (birbiri ardına yazdırılarak) ancak bunu daha verimli hale getirmek için `secret` yerine bir tek argüman alan `#(secret)` yöntemini tanımlayabilirdik:

```
puts "Kelime #{secret}.\n"
```

Birçok yazılımcı, bu yolun çıktı vermek için daha açık olduğunu düşünürler. Tek bir dizge oluşturur ve bu dizgeyi **puts**'a tek bir argüman gibi sunar.

Standart betik çıktısında **puts** kullanma düşüncesini işledik, ancak bu betik 4. ve 13. satırlarda **print**'i de kullanır. Bu ikisi aynı şey değildir. **print** dizgeyi kendisine verildiği gibi çıktılarken, **puts**; aynı zamanda çıktı satırının sonlanmasını sağlar. 4. ve 13. satırlarda **print** kullanıldığı için, işleç bir sonraki satırın başlangıcına geçeceği yerde, işleci ekrana çıktılanan dizgenin yanında bırakır. Bu durum kullanıcı girdisi için tanınabilir bir durumdur. Aşağıdaki dört çıktı çağrısı da aynı sonucu verir:

```
#satırsonu karakteri yoksa, puts tarafından eklenir:
puts  "Tasi delen suyun kuvveti degil, damlalarin sürekliligidir."

#satırsonu karakteri print komutuna verilmelidir:
print "Tasi delen suyun kuvveti degil, damlalarin sürekliligidir.\n"

#çıktıyı + ile birleştirebilirsiniz:
print 'Tasi delen suyun kuvveti degil, damlalarin sürekliligidir.'+"\n"

# ya da birden fazla dizge vererek birleştirebilirsiniz:
print 'Tasi delen suyun kuvveti degil, damlalarin sürekliligidir.', "\n"
```

Düzenli İfadeler

Son olarak *Düzenli İfadeler* (sayfa: 8) bölümündeki yazılımı inceleyeceğiz.

```
1 st = "\033[7m"
2 en = "\033[m"
3
4 while TRUE
5   print "str> "
6   STDOUT.flush
7   str = gets
8   break if not str
9   str.chop!
10   print "pat> "
11   STDOUT.flush
12   re = gets
13   break if not re
14   re.chop!
15   str.gsub! re, "#{st}\\&#{en}"
16   print str, "\n"
17 end
18 print "\n"
```

4. satırda **while**'ın koşulu sonsuz döngüyü sağlamak için **true** yapılmıştır. Ancak döngüden çıkabilmek için 8. ve 13. satırlarda **break** kullandık. Bu iki **break** aynı zamanda **if** deyiminin niteleyicilerinden biridir. Bir *if niteleyicisi* sadece ve sadece koşul sağlandığı zaman sol yandaki terimini çalıştırır.

`chop!` için (9. ve 14 satıra bakın) hakkında söylenecek çok şey var. Ruby'de geleneksel olarak yöntem isimlerinin sonuna '**!**' ya da '**?**' ekleriz. Ünlem işareti (**!** bazen "bang!" diye söylenir) potansiyel olarak yıkıcı bir görev görür, daha da açmak gerekirse; dokunduğu değeri değiştiren bir işleçtir. `chop!` bir dizgeye doğrudan etki eder ancak ünlem işareti olmayan bir `chop` bir kopya üzerinde çalışır. Aşağıda ikisi arasındaki fark görülüyor:

```
 ruby> s1 = "forth"
  "forth"
ruby> s1.chop!        # Bu s1'in değerini değiştirir.
  "fort"
ruby> s2 = s1.chop    # s2'ye değiştirilmiş bir kopyasını koyar.
  "for"
ruby> s1              # ... s1'e dokunmaz.
  "fort"
```

İlerde sonunda soru işareti olan yöntem isimleriyle karşılaşacaksınız (**?** genelde "huh?" şeklinde telaffuz edilir). Bu **true** ya da **false** döndüren bir 'doğrulama' yöntemidir.

15. satırda dikkat edilmesi gereken önemli bir uyarı yer almaktadır. Öncelikle `gsub!`'in başka bir sözde 'yıkıcı' yöntem olduğuna dikkat edelim. `re`'ye uyan her ne varsa `str`'nin yerine koyar (**sub** ikame etmekten (substitute), **g** ise globalden gelir); sadece ilk bulunanı değil, dizgedeki `tüm` eşleşen kısımları değiştirir. Çok iyi, çok güzel; fakat eşleştirilen kısımları neyle değiştireceğiz? 1. ve 2 satırda önalan ve artalan renklerini peşpeşe değiştiren `st` ve `en` adlı iki dizge tanımladık. 15. satırdaysa bunları, olduğu gibi yorumlandıklarından emin olmak için **#{}** arasına yazdık (*değişken isimlerinin yazdırıldığını görmeyiz*). Bunların arasında da "**\\&**" kodunu görüyoruz. Bu küçük bir hiledir. Yer değiştirilen dizge çift tırnak arasında olduğu için bir çift ters bölü işareti tek bir taneymiş gibi yorumlanır. Böylece `gsub!`'in göreceği şey "**\&**" olur ve bu da ilk konumda ne eşleştirildiyse onu gösteren özel bir koda dönüştürülür. Yani yeni dizge eşleşen parçaları farklı renkte gösterilen eski dizge olur.

8. Denetim Yapıları

Bu bölümde Ruby'nin denetim yapılarını açıklayacağız.

case

case deyimini bir dizi koşulu test etmek için kullanırız. Bu yapı, C ve Java 'daki **switch**'e çok benzer ancak birazdan da göreceğiniz gibi ondan biraz daha güçlüdür.

```
ruby> i=8
ruby> case i
    | when 1, 2..5
    |   print "1..5\n"
    | when 6..10
    |   print "6..10\n"
    | end
6..10
   nil
```

2..5, 2 ile 5 arasındaki sayı aralığını ifade eder. Sonraki ifade, **i** değişkeninin bu aralığa düşüp düşmediğini sınar:

```
(2..5) === i
```

case aynı anda birden çok koşulu sınamak için **===** ilişki işleçini kullanır. Ruby'nin nesneye yönelik yapısını korumak için **===** işleci nesneye uygun olarak yorumlanır. Örneğin aşağıdaki kodda ilk **when**'de dizgelerin eşitliği sınanırken ikinci **when**'de düzenli ifadenin eşleşmesi sınanıyor.

```
ruby> case 'abcdef'
    | when 'aaa', 'bbb'
    |   print "aaa or bbb\n"
    | when /def/
    |   print "/def/ icerir\n"
    | end
/def/ icerir
   nil
```

while

Ruby döngü oluşturmak için bir çok yola sahiptir, bununla birlikte ileriki bölümde doğrudan döngü kurmanıza yarayan *yineleyici*leri göreceksiniz.

while bir tekrarlanmış **if**'ten başka birşey değildir. **while**'ı daha önce kelime–tahmin oyunumuzda ve düzenli ifadeler yazılımımızda kullanmıştık (*Düzenli İfadeler* (sayfa: 8) bölümüne göz atın); **while** *koşul*... **end** ile çevrilmiş bir kod bloğu koşul doğru olduğu sürece tekrarlanmaktaydı. Ancak **while** ve **if** ayrı ifadeler olarak da kullanılabilir:

```
ruby> i = 0
   0
ruby> print "Sıfır.\n" if i==0
Sıfır.
   nil
ruby> print "Negatif.\n" if i<0
   nil
ruby> print "#{i+=1}\n" while i<3
1
2
3
   nil
```

Bazen bir sınama koşulunu olumsuzlamak istersiniz. **unless** bir olumsuzlandırılmış **if**, **until** ise olumsuzlandırılmış bir **while**'dır. Bunları deneyip tecrübe etmeyi size bırakıyoruz.

Bir döngüyü içerden kesmek için dört yol vardır. İlki C'deki gibi, döngüden tamamen çıkmamızı sağlayan **break**'tir. İkincisi (C'deki **continue**'ya benzeyen) döngünün başına atlayan **next**'tir. Üçüncüsü o anki yinelemeyi tekrar çalıştıran **redo**'dur. Aşağıda **break**, **next** ve **redo** arasındaki farkı açıklayan bir C kodu bulunuyor:

```
while (koşul) {
 label_redo:
   goto label_next;        /* ruby'nin "next"'i */
   goto label_break;       /* ruby'nin "break"'i */
   goto label_redo;        /* ruby'nin "redo"'su */
   ...
   ...
 label_next:
}
label_break:
...
```

Bir döngünün içinden çıkmak için dördüncü yol, **return**'dür. **return** sadece döngüden değil, döngüyü içeren yöntemden de çıkmamızı sağlar. Eğer bir argüman verildiyse, yöntem çağrısına dönecektir; aksi halde **nil** döndürecektir.

for

C yazılımcıları **for** döngüsünün nasıl yapıldığını merak edeceklerdir. Ruby'nin **for**'u tahmin ettiğinizden biraz daha ilginçtir. Aşağıdaki döngü, her eleman için bir kere döner:

```
for eleman in dizi
  ...
end
```

Elemanlar bir değer aralığı olabilir (bu, döngü denildiğinde çoğu insanın anladığı şeydir):

```
ruby> for num in (4..6)
    |    print num,"\n"
    | end
4
5
6
   4..6
```

Elemanlar dizi gibi başka bir türden değerler olabilir:

```
ruby> for elt in [100,-9.6,"pickle"]
    |    print "#{elt}\t(#{elt.type})\n"
    | end
100     (Fixnum)
-9.6    (Float)
pickle  (String)
   [100, -9.6, "pickle"]
```

Ancak ilerleme kaydediyoruz. **for** aslında **each**'i kullanmanın başka bir yoludur. Aşağıdaki iki biçim de aynı görevi görür:

```
# Eğer daha önce C ya da Java kullandıysanız aşağıdaki gibi
# birşey tercih edersiniz:
for i in collection
  ...
end
```

```
#  Smalltalk yazılımcısıysanız aşağıdaki gibi birşeyi tercih edersiniz:
collection.each {|i|
  ...
}
```

Yineleyiciler sık sık geleneksel döngülere tercih edilir; bir kere onları kullanmaya alıştığınızda ne kadar kolay olduklarını göreceksiniz.

9. Yineleyiciler

Yineleyiciler sadece Ruby'ye özgü bir kavram değildir. Genel olarak çoğu nesneye yönelik yazılım geliştirme dilinde kullanılmaktadır. Lisp'te de yineleyiciler olarak adlandırılmasalar da kullanılmaktadır. Ancak yineleyici kavramı neredeyse her dilde değişik bir anlam kazandığı için önce bu kavramı daha ayrıntılı anlatmaya çalışalım:

Yinelemek sözcüğü aynı şeyi birçok kez tekrarlamak anlamına gelir.

Kod yazarken değişik durumlarda döngülere ihtiyacımız olur. C'de **for** ya da **while** kullanarak işimizi hallederiz. Örneğin,

```
char *str;
for (str = "abcdefg"; *str != '\0'; str++) {
  /* her karakter için işlemler burada */
}
```

C'nin **for(...)** sözdizimi döngünün yaratılmasında soyutlama sağlayarak yardımcı olsa da `*str`'nin bir boş bir karakterle sınanması yazılımcının dizge yapısı hakkında daha ayrıntılı bilgi edinmesini gerektirir. Bu C'nin düşük–seviyeli olduğunu hissettiren nedenlerden biridir. Yüksek seviyeli diller yineleyicilere uyumluluklarıyla ün kazanmışlardır. Aşağıdaki **sh** kabuk betiğini göz önünde bulunduralım:

```
#!/bin/sh

for i in *.[ch]; do
  # ... her dosya icin yapilacak birkaç işlem
done
```

Bulunulan dizindeki tüm C kaynak ve başlık dosyaları çalıştırıldı ve komut satırı ayrıntıları tuttu. C'den daha yüksek seviyeli olduğunu düşünüyorum, öyle değil mi?

Ancak göz önüne alınması gereken başka bir nokta daha var: bir dilin gömülü veri yapıları için yineleyicileri desteklemesi güzel birşey olsa da, geri dönüp kendi veri yapılarımızı tekrarlatacak düşük seviyeli döngüler yazmak hayal kırıklığı yaratacak bir iş olacaktır. Nesneye yönelik yazılım geliştirmede, kullanıcılar çoğu kez ardı ardına veri türleri tanımlarlar ve bu ciddi bir sorun olabilir.

Her nesneye yönelik yazılım geliştirme dili yineleyiciler için kolaylıklar içerir. Bazı diller bu iş için özel sınıflar tanımlarken, Ruby yineleyicileri doğrudan tanımlamayı tercih eder.

Ruby'nin **String** türü bazı yararlı yineleyicilere sahiptir.

```
ruby> "abc".each_byte{|c| printf "<%c>", c}; print "\n"
<a><b><c>
   nil
```

`each_byte`, dizgedeki her karakter için bir yineleyicidir. Her bir karakter yerel bir değişken olan `c`'ye yerleştirilir. Bu daha çok C koduna benzeyen bir şeyle açıklanabilir...

```
ruby> s="abc";i=0
```

```
   0
ruby> while i<s.length
    |    printf "<%c>", s[i]; i+=1
    | end; print "\n"
<a><b><c>
   nil
```

Buna rağmen `each_byte` yineleyicisi hem kabul edilebilir bir basitliktedir hem de **`String`** sınıfı radikal bir değişikliğe uğrasa da çalışmaya devam eder. Yineleyicilerin başka bir yararı da değişiklere karşı sağlam durmasıdır ki bu da iyi bir kodun karakteristik özelliklerinden biridir (evet, sabırlı olun, *sınıflar* hakkında da konuşacağız.).

`String`'in başka bir yineleyicisi de **`each_line`**'dır.

```
ruby> "a\nb\nc\n".each_line{|l| print l}
a
b
c
   nil
```

C'de satır sınırlayıcıları bulmak, alt dizgeler üretmek gibi güç işlerin yineleyicilerle kolayca üstesinden gelinebilir.

Geçen bölümdeki **`for`** döngüsü, **`each`** işlecini kullanarak tekrarlamayı sağlamaktaydı. **`String`**'in **`each`**'i aynı **`each_line`** gibi görev görür, o yüzden yukarıdaki örneği **`for`** ile tekrar yazalım:

```
ruby> for l in "a\nb\nc\n"
    |   print l
    | end
a
b
c
   nil
```

Bulunan yineleyiciyi döngünün başından itibaren tekrar ettirmek için **`retry`** denetim yapısını kullanabiliriz.

```
ruby>  c=0
   0
ruby>  for i in 0..4
    |   print i
    |   if i == 2 and c == 0
    |     c = 1
    |     print "\n"
    |     retry
    |   end
    | end; print "\n"
012
01234
   nil
```

Yineleyici tanımlamasında bazen **`yield`**'le karşılaşırız. **`yield`**, denetimi yineleyiciye parametre olarak geçilen kod bloğuna verir (bu konu *Yordam Nesneleri* (sayfa: 28) bölümünde daha ayrıntılı anlatılacaktır).

Aşağıdaki örnekte, argümanda verilen sayı kadar bir kod bloğunu tekrarlayan **`repeat`** yineleyicisi tanımlanmıştır.

```
ruby> def repeat(num)
    |   while num > 0
    |     yield
    |     num -= 1
```

```
    |   end
    | end
   nil
ruby> repeat(3) { print "foo\n" }
foo
foo
foo
   nil
```

retry ile **while** gibi çalışan ancak hız açısından pek de pratik olmayan bir yineleyici tanımlayabiliriz.

```
ruby> def WHILE(cond)
    |   return if not cond
    |   yield
    |   retry
    | end
   nil
ruby> i=0; WHILE(i<3) { print i; i+=1 }
012  nil
```

Yineleyicinin ne olduğunu anladınız mı? Bir kaç kısıtlama hariç, kendi yineleyicinizı yazabilirsiniz, aslında yeni bir veri türü tanımladığınız zaman ona uygun bir yineleyici tanımlamanız da uygun olacaktır. Yukarıdaki örnekler pek de kullanışlı örnekler sayılmazlar. Sınıfları daha iyi anladığımızda daha uygulanabilir yineleyiciler hakkında konuşabiliriz.

10. Nesne Yönelimli Düşünme

Nesne yönelimlilik kavramı çekici bir kavramdır. Herşeyi nesneye yönelik olarak çağırmak kulağınıza hoş gelebilir. Ruby nesne yönelimli bir betik dili olarak adlandırılır, ancak gerçekte bu "nesne yönelimlilik" kavramı nedir?

Bu soruya aşağı yukarı hepsi aynı kapıya çıkan bir sürü cevap bulunabilir. Çabukça toparlamak yerine, isterseniz öncelikle geleneksel yazılım paradigması üzerinde duralım.

Geleneksel olarak, bir yazılım geliştirme sorunu bazı verilerin gösterimleri ile bu veriler üzerinde çalışan yordamlar olarak karşımıza çıkar. Bu model altında, veri hareketsiz, edilgen ve beceriksizdir; tamamen etkin, mantıksal ve güçlü bir yordamın merhametine kalmıştır.

Bu yaklaşımdaki sorun, yazılımları geliştiren yazılımcıların sadece insan olması ve dolayısıyla bir çok ayrıntıyı sadece bir sefer kafalarında net olarak tutabilmeleridir. Proje genişledikçe, yordamsal öz daha karmaşık ve hatırlaması zor bir noktaya gelir.

Küçük düşünce kusurları ve yazım yanlışlarıyla sonuçta elinizde iyi–gizlenmiş yazılım hataları kalır.

Zamanla yordam çekirdeğinde istenmeyen etkileşimler doğabilir; bu iş dokunaçlarının yüzünüze değmesine izin vermeden sinirli bir mürekkep balığı taşımaya benzer.

Bu geleneksel paradigmalarla yazılım geliştirirken hataları azaltmak ve sınırlamak için kılavuzlar bulunmaktadır, ancak yöntemi kökten değiştirmek daha iyi bir çözüm olacaktır.

Peki nesneye yönelik yazılım geliştirme, mantıksal işin sıradan ve tekrarlayan yönünü verinin kendisine emanet etmemizi mümkün kılmak ve veriyi edilgen durumdan etkin duruma sokmamız için ne yapar? Başka bir açıdan,

- Her veri parçasına, erişip içindekileri etrafa fırlatmamıza izin veren kapağı açık bir kutu gibi davranmayı bıraktık.
- Her veri parçasına kapağı kapalı ve iyi işaretlenmiş düğmeleri bulunan çalışan bir makine gibi davranmaya başladık.

"Makine" olarak tanımladığımız şey çok basit ya da çok karmaşık olabilir ancak bunu dışarıdan bakarak söyleyemeyiz ve makineyi açmayı (tasarımıyla ilgili bir sorun olduğunu düşünmedikçe) istemeyiz. Bu yüzden veriyle etkileşimde bulunmak için düğme çeviriyor gibi işlem yapmamız gerekir. Makine bir kere kurulduğu zaman nasıl çalıştığı hakkında düşünmememize gerek yoktur.

Kendimize iş çıkardığımızı düşünebilirsiniz ancak bu yaklaşımla bazı şeylerin yanlış gitmesini önleyebiliriz.

Şimdi açıklayıcı olması açısından basit ve küçük bir örnek görelim: Arabanızın bir yolmetresi olsun. Görevi yeniden başlatma düğmesine son basıldığından itibaren ne kadar yol katedildiği ölçmektir. Bu durumu bir yazılım geliştirme dilinde nasıl tasarlayabiliriz? C'de yolmetre sadece sayısal bir değişken olmalıdır, muthemelen bir **`float`**. Yazılım bu değişkenin değerini küçük aralıklarla arttıracak, uygun gördüğü zamansa sıfır yapıp yeniden başlatacaktır. Burada yanlış olan nedir? Yazılımdaki bir hata bu değişkene uydurma bir değer atayabilir ve beklenmedik sonuçlar ortaya çıkabilir. C'de yazılım geliştirmiş herhangi biri böylesine küçük ve basit bir hatayı bulmak için saatler ya da günler harcamanın ne demek olduğunu bilir (hatanın bulunma sinyali genelde alında şaklayan bir tokattır).

Aynı problem nesneye yönelik bağlamda da karşımıza çıkabilirdi. Yolmetreyi tasarlayan bir yazılımcının soracağı ilk şeylerden biri tabii ki "hangi veri yapısı bu durum için daha uygundur?" olmayacaktır. Ama "Bunun tam olarak nasıl çalışması gerekiyor?" şeklinde bir soru daha uygun olacaktır. Aradaki fark daha malumatlı olmaktır. Bir kilometre sayacının gerçekte ne işe yaradığına ve dış dünyanın onunla nasıl etkileşimde bulunmayı beklediğine karar vermek için biraz zaman ayırmamız gereklidir. Şimdi arttırabileceğimiz, yeniden başlatabileceğimiz ve değerini okuyabileceğimiz ve başka bir şey yapmayan küçük bir makine yapmaya karar verdik.

Yolmetremize keyfi bir değer atamak için bir yol tanımlamadık; neden? Çünkü yolmetrelerin bu şekilde çalışmadığını biliyoruz. Yolmetreyle yapabileceğiniz pek az şey var, ki bunların hepsini yapmaya izin verdik. Bu şekilde eğer yazılımda herhangi birşey yolmetrenin değerinin yerine geçmeye çalışırsa (örneğin arabanın klimasının derecesi) işlerin yanlış gittiğine dair uyarı alırsınız. Koşan yazılımın (dilin doğasına göre muhtemelen derleme sırasında) yolmetre nesnelerine keyfi değerler atamaya izni olmadığını söyledik. Mesaj tam olarak bu olmayabilir ama buna yakın birşeydir. Ancak hatayı engellemiyor, değil mi? Ancak hatanın yerini kolayca gösterir. Bu nesneye yönelik yazılım geliştirmenin zamanımızı boşa harcamaktan kurtaran birkaç yolundan biridir.

Yukarıda soyutlamanın yalnızca bir adımını yaptık, artık makinelerden oluşan bir fabrika yapmak kolaylaştı. Tek bir yolmetreyi doğrudan oluşturmak yerine, basit bir kalıptan istediğimiz sayıda yolmetre yapmayı tercih etmeliyiz. Kalıp (ya da isterseniz yolmetre fabrikası) "sınıf" olarak adlandırdığımız kavrama, oluşturduğumuz yolmetre de "nesne" olarak tanımladığımız kavrama karşılık gelmektedir. Bir çok nesneye yönelik yazılım geliştirme dili, herhangi bir nesne oluşturmdan önce bir sınıfın tanımlı olmasını gerekli kılar, ancak Ruby'de böyle bir durum sözkonusu değildir.

Bu kullanımın nesneye yönelik tasarımı kuvvetlendirmediğini de not düşelim. Elbette her dilde, anlaşılamayan, hatalı, yarım yamalak kod yazmak mümkündür. Ruby'nin sizin için yaptığı şey (özellikle C++'nın aksine) nesneye yönelik yazılım geliştirme kavramını sindirmenizi sağlayarak, daha küçük bir ölçekte çalışırken çirkin bir kod yazmamak için çaba sarfetmenizi önler. İleriki bölümlerde Ruby'nin takdire şayan diğer özelliklerini açıklayacağız. Hala bizimle misiniz?

11. Yöntemler

Yöntem nedir? Nesneye yönelik yazılım geliştirmede, nesnenin dışından doğrudan veri üzerinde işlem yapmak yerine (*eğer nazikçe böyle yapmalarını söylerseniz*) nesnelerin kendilerini nasıl çalıştıracakları hakkında bilgiye sahip olması tercih edilir. Nesnelere mesajlar gönderdiğimizi ve bunların genelde bir olay tarafından gerçekleştirildiğini ya da anlamlı bir cevapla karşılandığını söyleyebilirsiniz. Bu muhtemelen bizim özellikle bilmemizi gerektiren ya da nesnenin kendi içinde nasıl çalıştığına dikkat etmemizi gerektirmeyen bir olaydır. Bir nesneye gerçekleştirmesi için (*ya da anlayacağı mesajlar göndermemiz için*) izinli olduğumuz görevlere, ***nesnenin yöntemleri*** denir.

Ruby'de bir nesnenin yöntemini nokta gösterimi ile çağırırız (C++ ya da Java'da olduğu gibi).

```
ruby>   "abcdef".length
   6
```

Muhtemelen bu dizgenin *uzunluğunun ne kadar olduğu* soruluyor.

Teknik olarak,`"abcdef"` nesnesi için `length` yöntemini çağırıyoruz.

Diğer nesnelerin `length` yöntemi için biraz farklılıkları olabilir ya da hiç olmayabilir de. Mesajlara nasıl cevap verileceği kararı yazılımın çalıştırılması sırasında verilir ve hangi nesneye başvurduğuna bağlı olarak olay değişebilir.

```
ruby>   foo = "abc"
   "abc"
ruby>   foo.length
   3
ruby>   foo = ["abcde", "fghij"]
   ["abcde", "fghij"]
ruby>   foo.length
   2
```

`length` yönteminin, nesneye göre değişebilmesiyle neyi kastediyoruz? Yukarıdaki örnekte ilk önce `foo`'nun uzunluğunu soruyoruz, basit bir dizgeye başvuruyor ve sadece tek bir mantıklı cevabı olabilir. İkinci sefer `foo` bir diziye başvuruyor ve uzunluğunun 2, 5 ya da 10 olduğunu düşünebilirsiniz ama genelde en kabul edilebilir cevap tabii ki 2 olacaktır.

```
ruby>   foo[0].length
   5
ruby>   foo[0].length + foo[1].length
   10
```

Burada dikkat edilmesi gereken nokta bir dizinin, dizi olmanın ne demek olduğunu bilmesidir. Ruby'de veri parçaları beraberlerinde bilgi taşıdıkları için talepler otomatik olarak algılanabilir ve bir çok yolla gerçekleştirilebilir.

Bu yazılımcıyı spesifik işlev adlarını hatırlamaktan kurtarır, değişik veri türlerine uygulanabilir ve sonuç istediğimiz gibi olur. Nesneye yönelik yazılım geliştirmenin bu özelliği `polimorfizm` olarak adlandırılır.

Bir nesne anlamadığı bir mesaj aldığında bir hata uyanır:

```
ruby>   foo = 5
   5
ruby>   foo.length
ERR: (eval):1: undefined method `length' for 5(Fixnum)
```

Sonuçta bir nesne için hangi yöntemlerın kabul edilebilir olduğunu bilmemiz geretiği halde, yöntemlerin nasıl işlendiğini bilmek zorunda değiliz.

Eğer argümanlar bir yönteme verilecekse genelde parantez içine alınırlar.

```
object.method(arg1, arg2)
```

Ancak belirsizlik ihtiva etmedikleri sürece kullanılmayabilirler de.

```
object.method arg1, arg2
```

Ruby, `self` adında bir nesnenin bir yöntemini çağırdığı zaman başvurulan özel bir değişkene sahiptir. Rahatlık için "`self.`" genelde yöntem çağrılırken kullanılmayabilir:

```
self.method_name(args...)
```

yukarıdaki ifadeyle aşağıdaki aynıdır:

```
method_name(args...)
```

Bir işlev çağrısı sadece **self**'le yöntem çağrımının kısaltılmış şeklidir. Bu da Ruby'yi saf bir nesneye yönelik yazılım geliştirme dili yapan şeydir. Hala işlevsel yöntemler diğer dillerdeki işlevlere çok benzese de aslında işlev çağrılarının Ruby'de gerçek nesne yöntemlerinden başka birşey olmadığını görmeniz gerekir. İstersek hala gerçek nesne yöntemleri değilmiş gibi kabul ederek işlevlerden bahsedebiliriz.

12. Sınıflar

Gerçek dünya sınıflandırabileceğimiz nesnelerle doludur. Örneğin küçük bir çocuk bir köpek gördüğünde, cinsine bakmaksızın "hav hav" demesi gibi biz de dünyadaki nesneleri kategorize ederiz.

Nesneye yönelik terminolojide, "köpek" gibi nesnelerin kategorize edilmiş haline **sınıf**, sınıfın özelleştirilmiş nesnelerinde **örnek** (instance) denir.

Genelde Ruby'de ya da herhangi başka bir nesneye yönelik yazılım geliştirme dilinde nesne yapmak için önce sınıfın karakteristikleri tanımlanır sonra da bir örnek tanımlanır. Bu süreci görebilmek için **Kopek** adında ilk basit sınıfımızı tanımlıyoruz:

```
ruby>   class Kopek
    |    def speak
    |      print "Hav Hav\n"
    |    end
    |  end
   nil
```

Sınıf tanımlaması **class** ile **end** arasında yapılmaktadır. Bu alanda bulunan **def**, önceki bölümlerde açıkladığımız gibi sınıfa bazı özel davranışlar kazandıran yöntemleri tanımlamak için kullanılır.

Artık bir **Kopek** sınıfı tanımladık, öyleyse şimdi bir köpek yapabiliriz:

```
ruby>   kucu = Kopek.new
   #<Kopek:0xbcb90>
```

Kopek sınıfından yeni bir örnek yarattık ve **kucu** adını verdik. **new** yöntemi her sınıf için yeni bir örnek yapmaya yarar. **kucu** sınıf tanımımıza göre bir **Kopek** olduğu için, bir köpekte olmasına karar verdiğimiz tüm özellikleri taşır. **Kopek** sınıfımız çok basit olduğu için **kucu**'dan yapmasını istediğimiz küçük bir hile var.

```
ruby>   kucu.konus
Hav Hav
   nil
```

Bir sınıftan yeni bir örnek yaratmak bazen örnekleme olarak adlandırılır. Köpeğimizin havlamasını test etmek için öncelikle bir köpeğimizin olması lazım, **Kopek** sınıfından bizim için havlamasını isteyemeyiz.

```
ruby>   Kopek.konus
ERR: (eval):1: undefined method `konus' for Kopek:class
```

Diğer taraftan, duygusal olarak bağlanmamış bir köpeğin sesini duymak istersek, geçici bir köpek yaratabilir ve kaybolmadan önce bizim için küçük bir ses çıkarmasını isteyebiliriz.

```
ruby>   (Kopek.new).konus   # ya da daha genel olarak, Kopek.new.konus
Hav Hav
   nil
```

"Bekle" diyebilirsiniz, "bu kerata nereye kayboldu böyle?" Bu doğru: eğer ona bir isim vermezseniz (**kucu**'da yaptığımız gibi) Ruby'nin otomatik çöp toplama mekanizması devreye girer ve bunun istenmeyen aylak bir köpek olduğuna karar verir ve merhametsizce yok eder. Gerçekten, sorun yok; biliyorsunuz ki tüm köpekleri istediğimizi söyleyebiliriz.

13. Miras

Gerçek hayatta yaptığımız sınıflandırmalar son derece hiyerarşiktir. Örneğin bütün kedilerin memeli olduğunu ve bütün memelilerin hayvan olduğunu biliriz. Küçük sınıflar mensup oldukları büyük sınıfların karakteristik özelliklerini miras alırlar. Eğer bütün memeliler nefes alabiliyorsa, bütün kediler de nefes alabiliyor demektir.

Bu durumu Ruby'de aşağıdaki gibi açıklayabiliriz:

```
ruby>   class Memeli
    |    def nefes
    |      print "nefes al, nefes ver\n"
    |    end
    |  end
   nil
ruby>   class Kedi<Memeli
    |    def konus
    |      print "Miyauvvvv\n"
    |    end
    |  end
   nil
```

Örneğin bir **Kedi**'nin nasıl nefes alması gerektiğini belirtmediğimizi farz edelim. Bu durumda her kedi **Kedi** sınıfı **Memeli** sınıfının bir alt sınıfı olarak tanımlanmışsa, bu davranışı **Memeli** sınıfından miras olarak alacaktır. (Nesneye yönelik terminolojide küçük sınıf `alt sınıf` , büyük sınıfsa `süper sınıf` olarak isimlendirilir.) Programcının bakış açısına göre, kediler nefes alma yeteneğini bağımsız olarak almıştır, eğer bir **konuşma** yöntemi de eklersek, artık kedilerimiz hem nefes alabilme hem de konuşabilme yeteneğine sahip olurlar.

```
ruby>   pisi = Kedi.new
   #<Kedi:0xbd80e8>
ruby>   pisi.nefes
nefes al, nefes ver
   nil
ruby>   pisi.konus
Miyauvvvv
   nil
```

Bazen süper sınıfta olması gereken ancak alt sınıf tarafından miras alınması istenmeyen bir özellik olabilir. Örneğin genel olarak kuşların uçmayı bildiğini ancak penguenlerin, kuşların uçamayan bir alt sınıfı olduğunu kabul edelim.

```
ruby>   class Kus
    |    def gagala
    |      print "Tüylerimi temizliyorum."
    |    end
    |    def uc
    |      print "Uçuyorum."
    |    end
    |  end
   nil
ruby>   class Penguen<Kus
    |    def uc
```

```
    |       fail "Üzgünüm, yüzmeyi tercih ederim."
    |     end
    |   end
   nil
```

Her yeni sınıfın her özelliğini ayrı ayrı tanımlamak yerine, sadece her alt sınıfla onun süper sınıfı arasındaki farklılıkları eklemek ya da yaniden tanımlamak daha iyidir. Miras'ın bu kullanımına bazen ***farksal yazılım geliştirme*** denir. Bu nesneye yönelik yazılım geliştirmenin en yararlı özelliklerinden biridir.

14. Yöntemleri Yeniden Tanımlama

Bir alt sınıfın davranışlarını, süper sınıfın yöntemlerini yeniden tanımlayarak değiştirebiliriz.

```
ruby>  class Insan
    |   def tanimla
    |     print "Ben bir insanim.\n"
    |   end
    |   def tren_bileti(yas)
    |     if yas < 12
    |       print "Indirimli ucret.\n";
    |     else
    |       print "Normal ucret.\n";
    |     end
    |   end
    | end
  nil
ruby>  Insan.new.tanimla
Ben bir insanim.
  nil
ruby>  class Ogrenci1<Insan
    |   def tanimla
    |     print "Ben bir ogrenciyim.\n"
    |   end
    | end
  nil
ruby>  Ogrenci1.new.tanimla
Ben bir ogrenciyim.
  nil
```

Örneğin süper sınıfın **tanimla** yöntemini tamamen yeniden tanımlamak yerine geliştirmek istediğimizi düşünelim. Bunun için **super**'i kullanıyoruz.

```
ruby>  class Ogrenci2<Insan
    |   def tanimla
    |     super
    |     print "Ben bir ogrenciyim, aynı zamanda.\n"
    |   end
    | end
  nil
ruby>  Ogrenci2.new.tanimla
Ben bir insanim.
Ben bir ogrenciyim, aynı zamanda.
  nil
```

super bizim orijinal yönteme argüman geçmemize izin verir. Bazen iki tür insan olduğunu söylerler...

```
ruby>  class Sahtekar<Insan
```

```
    |   def tren_bileti(yas)
    |     super(11) # ucuz tarife istiyoruz.
    |   end
    | end
  nil
ruby>  Sahtekar.new.tren_bileti(25)
Indirimli ucret.
  nil

ruby>  class Durust<Insan
    |   def tren_bileti(yas)
    |     super(yas) # verilen argümanı gecelim
    |   end
    | end
  nil
ruby>  Durust.new.tren_bileti(25)
Normal ucret.
  nil
```

15. Erişim Denetimi

Daha önce Ruby'nin işlevlere değil sadece yöntemlere sahip olduğunu söylemiştik. Ancak sadece tek bir tür yöntem yoktur. Bu bölümde ***erişim yöntemleri***'nden bahsedeceğiz.

Bir yöntemi, bir sınıf tanımlamasının içinde değil de, en üstte tanımladığımızı farz edelim. Bunun C gibi daha geleneksel bir dildeki `işlev`lerle aynı işi yapan bir yöntem olduğunu düşünürüz.

```
ruby>   def square(n)
    |    n * n
    |  end
   nil
ruby>   square(5)
    25
```

Yeni yöntemimiz hiç bir sınıfa bağlı değil gibi gözüküyor, ama aslında Ruby bu yöntemi tüm sınıfların süper sınıfı olan **`Object`** sınıfına verir. Sonuç olarak her nesne bu yöntemi nasıl kullanacağını bilir. Bu durum doğru gibi gözükebilir ama burada küçük bir nokta vardır: bu yöntem her sınıfın `private` yöntemidir. Bunun ne anlama geldiğinden bahsedeceğiz fakat bu durumun sonuçlarından bir tanesi de aşağıdaki gibi sadece işlev tarzında çağırabilmemizdir:

```
ruby>   class Foo
    |    def dorduncu_kuvvet(x)
    |      square(x) * square(x)
    |    end
    |  end
   nil
ruby>   Foo.new.dorduncu_kuvvet 10
  10000
```

Bir nesnenin, yöntemini açıkça çağırmasına izin verilmez:

```
ruby>   "fish".square(5)
ERR: (eval):1: private method `square' called for "fish":String
```

Bu durum daha geleneksel bir dildeki gibi işlev yazmamızı sağlarken, Ruby'nin saf 'nesneye yönelik' yapısını korumasına yardımcı olur (işlevler halen nesnelerin yöntemleridir, sadece alıcı üstü kapalı olarak **`self`**'tir.)

Önceki bölümlerde de vurguladığımız gibi nesneye yönelik yazılım geliştirmenin genel mantığı, belirtim ile gerçekleştirimi birbirinden ayırmak ya da bir nesnenin hangi görevleri yapmak istediği ve bunu nasıl yapabileceğiyle ilgilenmektir.

Bir nesnenin dahili işleri genelde kullanıcıdan saklanmalı, kullanıcı yalnızca neyin gidip geldiğiyle ilgilenmeli ve nesnenin kendi içinde neyi nasıl yaptığını bildiğine dair güvenmelidir.

Genelde nesnenin dış dünya tarafından görülmeyen ancak dahili olarak kullandığı yöntemlere sahip olması yararlı bir şeydir (ve bu durum kullacının nesneleri görme biçimi değiştirilmeksizin yazılımcının isteğine göre değiştirebilir).

Aşağıdaki basit örnekte **motor** sınıfının görünmediğini ama dahili olarak çalıştığını varsayalım.

```
ruby>   class Deneme
    |    def iki_kati(a)
    |      print a," kere iki ",motor(a),"\n"
    |    end
    |    def motor(b)
    |      b*2
    |    end
    |    private:motor  # motoru kullanıcılardan saklar
    |  end
    Deneme
ruby>   deneme = Deneme.new
   #<Deneme:0x4017181c>
ruby>   deneme.motor(6)
ERR: (eval):1: private method motor' called for #<Deneme:0x4017181c>
ruby>   deneme.iki_kati(6)
6 kere iki 12.
   nil
```

deneme.motor(6)'nın 12 değerini döndürmesini umuyorduk ancak bir Deneme nesnesi gibi davrandığımızda **motor**'un erişilemez olduğunu gördük. Yalnızca **iki_kati** gibi diğer **Deneme** yöntemleri **motor**'a ulaşma hakkına sahiptir. Böylece öncelikle **iki_kati** yöntemini içeren genel arayüze gitmek zorunda bırakıldık. Yazılımcı, kullancının **Deneme** nesnelerinin birbirini nasıl etkilediğine bakmaksızın **motor** yöntemini (bu örnekte muhtemelen başarım açısından **b*2**'yi **b+b** ile) değiştirebilir. Bu örnek erişim denetimlerini anlatmak için tabii ki çok basit bir örnektir ancak daha karmaşık ve ilginç sınıflar üretmeye başlayınca erişim denetiminin ne kadar yararlı bir kavram olduğunu anlayacaksınız.

16. Tekil Yöntemler

Bir örneğin davranışı ait olduğu sınıf tarafından belirlenir. Ancak bazen belirli bir örneğin özel bir davranışı olmasını isteyebiliriz. Çoğu yazılım geliştirme dilinde, sadece bir kere kullanacağımız bir sınıf tanımlamak gibi karmaşık bir yol seçebiliriz. Ruby'de her nesneye kendine ait yöntemler verebiliriz.

```
ruby>   class TekilDeneme
    |    def boyut
    |      print "25\n"
    |    end
    |  end
   nil
ruby>   dnm1 = TekilDeneme.new
   #<TekilDeneme:0xbc468>
ruby>   dnm2 = TekilDeneme.new
   #<TekilDeneme:0xbae20>
ruby>   def dnm2.boyut
```

```
    |    print "10\n"
    |  end
   nil
ruby>   dnm1.boyut
   25
   nil
ruby>   dnm2.boyut
   10
```

Yukarıdaki örnekte, **dnm1** ve **dnm2** aynı sınıfa mensup olmalarına rağmen, **dnm2**'nin **boyut** yöntemi yeniden tanımlandığı için farklı davranır. Sadece tekil bir nesneye verilen yönteme `tekil yöntem` (*singleton method*) denir.

Tekil yöntemler genelde grafik arayüzü elemanlarında (GUI) değişik düğmelerin değişik eylemler yapması gerektiğinde kullanılır.

Tekil yöntemler CLOS, Dylan vb. yazılım geliştirme dillerinde olduğu üzere Ruby'ye özgü değildir. Self ve NewtonScript gibi bazı diller ise sadece tekil yöntemlerden oluşmuştur. Bu tür diller `prototip tabanlı` diller olarak anılırlar.

17. Modüller

Ruby'de modüller sınıflara benzer özellikler gösterirler:

- Modülün örneği yoktur.
- Modülün alt sınıfı yoktur.
- Modül **module ... end** şeklinde tanımlanır.

Aslında modülün Module sınıfı, sınıfın Class sınıfının bir süper sınıfıdır. Anladınız mı? Hayır? O zaman devam edelim.

İki tip modül kullanımı bulunur. Bir tanesi ilişkili yöntemleri ve sabitleri merkezi bir yerde toplar. Ruby'nin standart kitaplığındaki **Math** modülü böyle bir rol oynar:

```
ruby>   Math.sqrt(2)
   1.41421
ruby>   Math::PI
   3.14159
```

:: işleci Ruby yorumlayıcısına bir sabit için hangi modülü yükleyeceğini söyler. (örneğin **Math** için bir anlam ihtiva eden birşey **PI** için başka bir anlama gelebilir). Eğer bir yöntem ya da sabitin, **::** kullanmadan doğrudan modüle başvurmasını istiyorsak bu modülü **include** ile ekleyebiliriz.

```
ruby>   include Math
   Object
ruby>   sqrt(2)
   1.41421
ruby>   PI
   3.14159
```

Diğer bir kullanım da ***karışım*** (mixin) olarak adlandırılır. Bazı nesneye yönelik yazılım geliştirme dili, C++ da dahil, birden fazla süper sınıftan miras almamızı sağlayan `çoklu miras` kavramına izin verir. Bunun gerçek dünyadaki örneği çalar saatler olabilir. Çalar saatleri hem `saat` sınıfına hem de `alarm` sınıfına sokabilirsiniz.

Ruby direkt olarak gerçek çoklu mirası desteklemez ancak **karışım** tekniği iyi bir alternatiftir. Modüllerin örneklenemeyeceğini ve alt sınıflanamayacağını hatırlayın, ancak bir modülü bir sınıf tanımlamasının içine **include** ile eklersek bu yöntemi sınıfa 'karıştırmış' ya da eklemiş oluruz.

Karışım stratejisi, ek olarak sınıfımıza hangi özellikleri istediğimizi belirtmenin başka bir yoludur. Örneğin eğer bir sınıfın çalışan bir **each** yöntemi varsa– bunu Enumerable standart kütüphanesine eklemek size **sort** ve **find** yöntemlerini bedava verir.

Modüllerin bu kullanımı işlevel bir çoklu miras kullanımı sağlarken– aynı zamanda basit bir ağaç yapısıyla sınıf akrabalıklarını temsil eder, böylece dil gerçekleştirimini basitleştirir (benzer bir dil gerçekleştirimi Java tasarımcıları tarafından da yapılmıştı).

18. Yordam Nesneleri

Beklenmeyen durumlara cevap verebilme genelde istenen bir durumdur. Eğer diğer yöntemlere kod bloklarını argüman olarak geçebilirsek yani koda bir veriymiş gibi davranabilirsek bu işi oldukça kolaylaştırmış oluruz.

Yeni bir ***yordam nesnesi***, **proc** kullanılarak oluşturulur:

```
ruby>   guguk = proc {
    |    print "GUGUKGUGUKGUGUK!!!\n"
    |  }
   #<Proc:0x4017357c>
```

Artık **guguk** bir nesne belirtiyor ve onun da diğer nesneler gibi istenebilir davranışları vardır. **call** yöntemi sayesinde bu davranışları talep edebiliriz:

```
ruby>   guguk.call
GUGUKGUGUKGUGUK!!!
   nil
```

Peki tüm bunlardan sonra, **guguk** bir yöntem argümanı gibi kullanılabilir mi? Tabii ki.

```
ruby>   def run( p )
    |    print "Bir yordamı çağırıyoruz...\n"
    |    p.call
    |    print "Bitti.\n"
    |  end
   nil
ruby>   run guguk
Bir yordamı çağırıyoruz...
GUGUKGUGUKGUGUK!!!
Bitti.
   nil
```

trap yöntemi aldığımız cevabı tercihimize göre istediğimiz sistem sinyaline atamamıza izin verir.

```
ruby>   inthandler = proc{ print "^C basıldı.\n" }
   #<Proc:0x401730a4>
ruby>   trap "SIGINT", inthandler
   #<Proc:0x401735e0>
```

Normalde ^C'ye basmak yorumlayıcıdan çıkmamızı sağlar. Ancak şimdi bir ileti yazıldı ve yorumlayıcı hala çalışmaya devam ediyor, böylece yaptığınız işi kaybetmemiş oluyorsunuz. (Yorumlayıcıda sonsuza dek kapana kısılmadınız, hala **exit** yazarak ya da ^D'ye basarak çıkabilirsiniz.)

Başka konulara geçmeden önce son bir not: bir yordamı bir sinyale bağlamadan önce ona illa ki isim vermek gerekli değildir. Anonim bir yordam nesnesi aşağıdaki gibi olabilir:

```
ruby>   trap "SIGINT", proc{ print "^C basıldı.\n" }
   nil
```

daha kısa şekilde,

```
ruby>   trap "SIGINT", 'print "^C basıldı.\n"'
   nil
```

Bu kısaltılmış biçim, küçük anonim yordamlar yazdığınızda okunabilirlik sağlar.

19. Değişkenler

Ruby biri sabit, ikisi de yarı–değişken olmak üzere üç çeşit değişkene sahiptir. Değişkenlerin ve sabitlerin türü yoktur. Türü olmayan değişkenler sakıncalı olsa bile, Ruby'nin `kolay ve hızlı` felsefesine uygun olarak bir çok avantaj sağlar.

Bir çok yazılım geliştirme dilinde değiştirilebilirliğini sağlamak (sabit olsalar bile) ve etki alanını belirlemek için değişkenlerin türü belirtilerek bildirilmesi gerekir. Ancak değişken türleri önemli olmadığından ve aşikar çözümün göreceğiniz gibi değişken isminden halledilebilediğinden beri, Ruby'de değişken bildirimlerine ihtiyacımız yoktur.

Değişken isminin ilk karakteri sayesinde bir bakışta türünü anlamak mümkündür:

$	genel değişken
@	örnek değişken
[a-z] ya da _	yerel değişken
[A-Z]	sabit

Bunların dışında tek istisna Ruby'nin yarı–değişkenleridir: daima o an çalışmakta olan nesneyi ifade eden **self** ve ilklendirilmemiş değişkenlere atanan anlamsız değer olan **nil**. Her ikisi de yerel değişkenler gibi tanımlanmış olsalar da, **self** yorumlayıcı tarafından saklanan bir genel değişken ve **nil** de gerçekte bir sabittir. Bunlar sadece iki istisna olduğu için üzerlerinde fazla durmayacağız.

self'e ya da **nil**'e değer atamamalıyız. **main**, bir **self** değeri olarak üst nesneyi ifade eder:

```
ruby>   self
   main
ruby>   nil
   nil
```

20. Genel Değişkenler

Genel değişkenler isimlerinin başında birer **$** işareti bulundururlar. Genel değişkenlere yazılımın her hangi bir yerinden başvurulabilir. İlklendirilmeden önce **nil** değerine sahiptirler.

```
ruby>   $foo
   nil
ruby>   $foo = 5
   5
ruby>   $foo
   5
```

Genel değişkenler dikkatli kullanılmalıdırlar. Her yerden yazılabildikleri için tehlikelidirler. Genel değişkenlerin aşırı kullanılması yanlışları izole etmede zorluk çıkarabildiği gibi yazılımın iyice düşünülmeden tasarlandığına dikkat çeker. Genel değişken kullanmayı uygun gördüğünüz zaman, onlara anlaşılabilir isimler verdiğinizden emin olun (**$foo** gibi birşeyi çağırmak oldukça kötü bir fikir değil mi?).

Genel değişkenlerin güzel bir özelliği de izlenebilir olmalarıdır; bir değişkenin değeri ne zaman değişirse o zaman çağrılan bir yordam belirleyebilirsiniz.

```
ruby>   trace_var :$x, proc{print "$x şimdi ", $x, "\n"}
   nil
ruby>   $x = 5
$x şimdi 5
   5
```

Bir global değişken, değiştiği zaman bir yordamı çalıştırmak için kullanılıyorsa, ***etkin değişken*** olarak da anılır.

Aşağıda **$** işaretini takiben tek bir karakter daha içeren bir dizi özel değişken bulunuyor. Örneğin **$$** Ruby yorumcusunun süreç numarasını içerir ve sadece okunabilirdir. Aşağıda önemli sistem değişkenleri ve anlamları bulunuyor: (ayrıntılar için Ruby Başvuru Kılavuzu[B10]na bakınız):

$!	son hata iletisi
$@	hatanın konumu
$_	**gets** tarafından okunan son dizge
$.	yorumlayıcı tarafından son okunan satır numarası
$&	regexp tarafından son bulunan dizge
$~	alt ifade (*subexpression*) dizisi olarak regexp tarafından bulunan son ifade
$n	son bulunan n'inci alt ifade (**$~[**n**]** ile aynı)
$=	büyük–küçük harfe duyarsız bayrak
$/	girdi kaydı ayracı (*input record separator*)
$	çıktı kaydı ayracı (*output record separator*)
$0	ruby betik dosyasının adı
$*	komut satırı argümanları
$$	yorumcunun süreç numarası (PID)
$?	son işletilen çocuk sürecin çıkış durumu

$_** ve **$~ için etki alanı yereldir. Her ne kadar isimleri gereği genel değişkenler olmaları gerekiyorsa da böyle daha kullanışlıdırlar.

21. Örnek Değişkenler

Örnek değişken **@** ile başlayan bir ada sahiptir ve etki alanı **self** nesnesi ile sınırlıdır. Aynı sınıfa dahil olan iki aynı nesne için iki değişik örnek değişken tanımlamak mümkündür.

Örnek değişkenler, yazılımcı hangi yöntemi tanımlarsa tanımlasın bir nesnenin dışından değiştirilemez (Ruby'nin örnek değişkenleri hiçbir zaman *genel* olamaz). Genel değişkenlerde olduğu gibi, örnek değişkenler de başlangıç değeri atanmazsa **nil** değerine sahip olurlar.

Ruby'de örnek değişkenleri bildirmeye gerek yoktur. Bu durum nesnelerin yapısına esneklik kazandırır. Aslında, her örnek değişken, nesnedeki ilk kullanımında kendiliğinden oluşturulur.

```
ruby>   class OrnekDeneme
    |     def set_foo(n)
    |       @foo = n
    |     end
    |     def set_bar(n)
    |       @bar = n
    |     end
    |   end
   nil
ruby>   i = OrnekDeneme.new
   #<OrnekDeneme:0x83678>
```

```
ruby>   i.set_foo(2)
   2
ruby>   i
   #<OrnekDeneme:0x83678 @foo=2>
ruby>   i.set_bar(4)
   4
ruby>   i
   #<OrnekDeneme:0x83678 @foo=2, @bar=4>
```

i'nin **set_bar** yöntemi çağrılmadan **@bar**'ın hiçbir değer belirtmediğine dikkat edin.

22. Yerel Değişkenler

Yerel değişkenler küçük harfle ya da _ karakteriyle başlayan isimlere sahiptirler. Yerel değişkenler genel ya da örnek değişkenlerde olduğu gibi, başlangıçta **nil** değerine sahip değildirler.

```
ruby>   $foo
   nil
ruby>   @foo
   nil
ruby>   foo
ERR: (eval):1: undefined local variable or method `foo' for main(Object)
```

Yerel bir değişkene yaptığınız ilk atama onu bildirmekle aynı şeydir. Eğer başlangıç değeri olmayan bir yerel değişkene başvurursanız, Ruby yorumlayıcısı bunun bir yöntemi çalıştırma girişimi olduğunu düşünür ve aşağıdaki gibi bir hata verir.

Genelde yerel bir değişkenin etki alanı aşağıdakilerden biridir:

- **proc{** ... **}**
- **loop{** ... **}**
- **def** ... **end**
- **class** ... **end**
- **module** ... **end**
- yazılımın tamamı (yukarıdakilerden herhangi biri yoksa)

Aşağıdaki örnekte görülen **defined?** işleci bir belirtecin tanımlanıp tanımlanmadığına bakar. Eğer tanımlanmışsa bir açıklama döndürür; tanımlanmamış ise **nil** değerini döndürür. Gördüğünüz gibi **bar** döngüde yerel, döngüden çıkınca tanımsızdır.

```
ruby>   foo = 44; print foo, "\n"; defined? foo
44
   "local-variable"
ruby>   loop{bar=45; print bar, "\n"; break}; defined? bar
45
   nil
```

Yordam nesneleri yynı etki alanındaki yerel değişkenleri paylaşırlar. Örnekte yerel değişken **bar**, **main** ve yordam nesneleri **p1** ve **p2** tarafından paylaşılmaktadır:

```
ruby>   bar=nil
   nil
ruby>   p1 = proc{|n| bar=n}
   #<Proc:0x8deb0>
ruby>   p2 = proc{bar}
   #<Proc:0x8dce8>
```

```
ruby>   p1.call(5)
   5
ruby>   bar
   5
ruby>   p2.call
   5
```

Baştaki "`bar=nil`"'ın çıkarılamayacağına dikkat edin; bu atama **`bar`**'ın **`p1`** ve **`p2`** tarafından kuşatılacağını garanti eder. Öteki türlü **`p1`** ve **`p2`** kendi yerel **`bar`** değişkenlerini sonlandırır ve **`p2`**'yi çağırmak "undefined local variable or method" hatasına neden olabilir.

Yordam nesnelerinin güçlü bir özelliği de argüman olarak aktarılabilme yetenekleridir: paylaşımlı yerel değişkenler özgün etki alanının dışından değer aktarıldığında bile geçerli kalırlar.

```
ruby>   def kutu
    |    icerik = 15
    |    getir = proc{icerik}
    |    ata = proc{|n| icerik = n}
    |    return getir, ata
    |  end
   nil
ruby>   okur, yazar = kutu
   [#<Proc:0x40170fc0>, #<Proc:0x40170fac>]
ruby>   okur.call
   nil
ruby>   yazar.call(2)
   2
ruby>   okur.call
   2
```

Ruby etki alanı konusunda bir parça akıllıca davranır. Örneğimizde **`icerik`** değişkeni **`okur`** ve **`yazar`** tarafından paylaşılıyordu. Aynı zamanda yukarıda tanımladığımız kutumuzdan birden çok okur–yazar çifti oluşturabilir ve her çiftin aynı sabiti paylaşmasını sağlayabiliriz.

```
ruby>   okur_1, yazar_1 = kutu
   [#<Proc:0x40172820>, #<Proc:0x4017280c>]
ruby>   okur_2, yazar_2 = kutu
   [#<Proc:0x40172668>, #<Proc:0x40172654>]
ruby>   yazar_1.call(99)
   99
ruby>   okur_1.call
   99
ruby>   okur_2.call
   nil
```

23. Sınıf Sabitleri

Bir sabit büyük harfle başlayan bir ada sahiptir. Sabitlere bir kere değer ataması yapılmalıdır. Ruby'nin şu anki uygulamasına göre, sabitlere yeniden değer ataması yapmak hata değil uyarı ile sonuçlanır (eval.rb'nin ANSI olmayan sürümü uyarı değil hata raporlar):

```
ruby>  fluid=30
   30
ruby>  fluid=31
   31
ruby>  Solid=32
```

```
   32
ruby>  Solid=33
   (eval):1: warning: already initialized constant Solid    33
```

Sabitler sınıflarla beraber tanımlanabilirler ancak örnek değişkenlerin aksine sınıfın dışından da erişilebilir durumdadırlar.

```
ruby>   class SabitSinifi
    |    C1=101
    |    C2=102
    |    C3=103
    |    def goster
    |      print C1," ",C2," ",C3,"\n"
    |    end
    |  end
   nil
ruby>   C1
ERR: (eval):1: uninitialized constant C1
ruby>   SabitSinifi::C1
   101
ruby>   SabitSinifi.new.goster
101 102 103
   nil
```

Sabitler aynı zamanda modül içinde de tanımlanabilirler.

```
ruby>   module SabitModulu
    |    C1=101
    |    C2=102
    |    C3=103
    |    def sabitleriGoster
    |      print C1," ",C2," ",C3,"\n"
    |    end
    |  end
   nil
ruby>   C1
ERR: (eval):1: uninitialized constant C1
ruby>   include SabitModulu
   Object
ruby>   C1
   101
ruby>   sabitleriGoster
101 102 103
   nil
ruby>   C1=99  # pek iyi bir fikir değil
   99
ruby>   C1
   99
ruby>   SabitModulu::C1  # modülün sabiti rahatsız edilmemiş...
   101
ruby>   SabitModulu::C1=99   # önceki sürümlerde buna izin verilmez
   (eval):1: warning: already initialized constant C1    99
ruby>   SabitModulu::C1  # sen iste yeter ki...
   99
```

24. Hata İşleme: `rescue` deyimi

Çalıştırılan bir yazılım beklenmeyen sorunlar doğurabilir. Okunmaya çalışılan bir dosya mevcut olmayabilir ya da veri kaydetmemek istediğimiz disk dolu olabilir yada kullanıcı beklenmeyen bir girdi yapabilir.

```
ruby>   file = open("bir_dosya")
ERR: (eval):1:in `open': No such file or directory - bir_dosya
```

Güçlü bir yazılım bu gibi durumları hassasiyetle yakalayacaktır. C yazılımcılarından, hata doğurabilecek her sistem çağrısının sonucunu kontrol etmeleri ve anında ne yapılacağına ilişkin karar vermeleri beklenir:

```
FILE *file = fopen("bir_dosya", "r");
if (file == NULL) {
  fprintf( stderr, "Dosya mevcut değil.\n" );
  exit(1);
}
bytes_read = fread( buf, 1, bytes_desired, file );
if (bytes_read != bytes_desired ) {
  /* hata giderme işlemleri... */
}
...
```

Bu yazılımcıları dikkatsizliğe ve ihmalciliğe iten, üstelik hataları tam olarak yakalayamayan bir yazılım yazmanıza yol açan sıkıcı bir uygulamadır. Öte yandan, işi doğru düzgün yapmak, yakalanabilecek bir çok hata olduğu için yazılımın okunabilirliğini oldukça zorlaştıracaktır.

Güncel bir çok dilde olduğu gibi Ruby'de de, yazılımcıyı ya da sonradan kodumuzu okuyan kişileri sıkıntıya sokmadan, sürprizleri kod bloklarından soyutlayan bir yolla yakalayabiliriz.

begin ile işaretlenmiş kod bloğu bir istisnayla karşılaşana dek çalıştırılır, hata durumunda denetimi **rescue** ile işaretlenmiş kod bloğuna verir. Eğer hiçbir istisnayla karşılaşılmazsa **rescue** kodu kullanılmaz. Aşağıdaki yöntem bir metin dosyasının ilk satırını döndürür, bir istisna ile karşılaşırsa **nil** değerini:

```
def first_line( filename )
  begin
    file = open("bir_dosya")
    info = file.gets
    file.close
    info  # Değerlendirmeye alınan son şey dönüş değeri
  rescue
    nil   # Dosyayı okuyamıyor musunuz? O zaman bir dizge dönmez.
  end
end
```

Bir problemle yaratıcı bir biçimde ilgilenmek istediğimiz zamanlar olacaktır. Örneğin dosyaya erişmek mümkün değilse standart girdi yerine başka bir şey kullanmak isteyebiliriz:

```
begin
  file = open("bir_dosya")
rescue
  file = STDIN
end

begin
  # ... girdiyi değerlendir ...
rescue
  # ... burada diğer istisnalarla uğraş.
end
```

begin kodunu tekrar çalıştırmak için **rescue**'nun içinde **retry**'ı kullanabiliriz. Bu bize önceki örneğimizi daha kısa şekilde yazmamıza izin verir:

```
fname = "bir_dosya"
begin
  file = open(fname)
  # ... girdiyi degerlendir ...
rescue
  fname = "STDIN"
  retry
end
```

Ancak burada bir kusur bulunmaktadır. Hiç olmayan bir dosya bu kodun sonsuz bir döngüde kendisini tekrar etmesini sağlayacaktır. **retry**'ı kullanırken bu tür durumlara dikkat etmelisiniz.

Her Ruby kütüphanesi, sizin de kendi kodunuzda yapabileceğiniz gibi, herhangi bir hata karşısında bir istisna doğurur. Bir istisnayı çıkarmak için **raise** kullanılır. **raise** tek argüman olarak istisnayı açıklayan bir dizge alır. Bu argüman isteğe bağlıdır ancak atlanmaması gereken bir husustur. Özel değişkenlerden olan **$!** ile sonradan ulaşılabilir.

```
ruby>   raise "deneme hatası"
   deneme hatası
ruby>   begin
    |     raise "dnm2"
    |   rescue
    |     print "Bir hata meydana geldi: ",$!, "\n"
    |   end
Bir hata meydana geldi: dnm2
   nil
```

25. Hata İşleme: ensure deyimi

Bazen bir yöntem işini bitirdikten sonra temizlik yapılması gerekebilir. Örneğin açılmış olan bir dosyanın kapatılması ya da bir veri için ayrılan bellek gözesinin boşaltılması gerekebilir. Eğer her yöntem için her zaman tek bir çıkış noktası olsaydı temizleme kodumuzu tek bir yere koyardık ve çalıştırılacağından emin olurduk. Ancak yöntem bir çok yere geri dönebilir ve temizlik kodumuz beklenmeyen istisnalardan dolayı atlanabilir.

```
begin
  file = open("/tmp/bir_dosya", "w")
  # ... dosyaya yaziliyor...
  file.close
end
```

Ayrıca eğer kodun dosyaya yazdığımız kısmında bir istisna meydana gelirse o zaman dosya açık bırakılabilir. Ve böyle bir fazlalığa gitmek istemeyiz:

```
begin
  file = open("/tmp/bir_dosya", "w")
  # ... dosyaya yazılıyor ...
  file.close
rescue
  file.close
  fail # istisna yakalanıyor
end
```

Bu hantal bir yöntemdir; her **return** ve **break** ile ilgilenmek zorunda kalınca işler çığrından çıkar.

Bu yüzden "**begin...rescue...end**" şemasına **ensure** adında başka bir anahtar kelime daha ekleriz. **ensure** kodu **begin** kodunun başarılı olup olmadığına bakmaksızın çalıştırılır.

```
begin
  file = open("/tmp/bir_dosya", "w")
  # ... dosyaya yazılıyor ...
rescue
  # ... istisnalar yakalanıyor...
ensure
  file.close   # ...her zaman yapılması gerekir
end
```

ensure kodunu **rescue** olmadan da kullanmak mümkündür ya da tam tersi; ancak aynı **begin...end** bloğunda birlikte kullanılıyorlarsa **rescue**, **ensure**'den önce gelmelidir.

26. Erişgeçler

Geçtiğimiz bölümlerde örnek değişkenlerden kısaca bahsettik ancak henüz işimiz bitmedi. Bir nesnenin örnek değişkenleri onun kendisine ait olan ve aynı sınıfa ait diğer nesnelerden ayıran özniteliklerdir.

Bu öznitelikleri okuyabilmek ve yazabilmek önemlidir; bu yüzden ***öznitelik erişgeçleri*** denilen yöntemi kullanırız. Bir kaç dakika sonra erişgeç yöntemlerini her zaman açıkça yazmak zorunda olmadığımızı göreceksiniz ancak şimdilik tüm devinimlere bakalım. Erişgeçler iki çeşittir: *yazıcılar* ve *okuyucular*.

```
ruby>   class Meyve
    |    def cesit_ata(k)  # bir yazıcı
    |      @kind = k
    |    end
    |    def ne_cesit     # bir okuyucu
    |      @kind
    |    end
    |  end
   nil
ruby>   f1 = Meyve.new
   #<Meyve:0xfd7e7c8c>
ruby>   f1.cesit_ata("seftali")  # yazıcıyı kullan
   "seftali"
ruby>   f1.ne_cesit           # okuyucuyu kullan
   "seftali"
ruby>   f1                    # nesneyi yokla
   #<Meyve:0xfd7e7c8c @kind="seftali">
```

Yeterince basit; baktığımız meyve hakkında istediğimiz bilgiyi yerleştirebilir ya da erişebiliriz. Ama yöntem isimlerimiz biraz uzun. Aşağıdaki daha kısa ve daha uzlaşımsal:

```
ruby>   class Meyve
    |    def cesidi=(k)
    |      @cesidi = k
    |    end
    |    def cesidi
    |      @cesidi
    |    end
    |  end
   nil
ruby>   meyve = Meyve.new
   #<Meyve:0xfd7e7c8c>
ruby>   meyve.cesidi = "muz"
   "muz"
ruby>   meyve.cesidi
```

```
   "muz"
```

`inspect` yöntemi

Küçük bir uzlaşma sağlanmıştır. Bir nesneye doğrudan ulaşmak istediğimizde **#<birNesne:0x83678>** gibi şifreye benzer birşeyle karşılaştığımıza dikkat edin. Bu öntanımlı bir davranıştır ve istediğiniz gibi değiştirebilirsiniz. Yapmanız gerek tek şey **inspect** yöntemini eklemektir. **inspect** yöntemi, nesneyi birkaç ya da bütün örnek değişkenleri de içeren ve mantıklı bir şekilde tanıtan bir yöntemdir.

```
ruby>   class Meyve
    |    def inspect
    |      @kind + "bir meyve çeşididir"
    |    end
    |  end
   nil
ruby>  meyve
   "muz bir meyve çeşididir"
```

Benzer bir yöntem de ekrana bir nesne yazdıracağımız zaman kullandığımız **to_s** (dizgeye dönüştürür) yöntemidir. Genel olarak **inspect** yöntemini yazılım geliştirirken ve hata ayıklarken kullandığınız bir araç olarak, **to_s**'yi de yazılımın çıktısını düzeltmek için kullandığımız bir yol düşünebilirsiniz.

eval.rb sonuçları görüntülemek için her zaman **inspect** yöntemini kullanır.

p yöntemini, yazılımlarınızdan hata ayıklama çıktısı almak için kullanabilirsiniz.

```
# Bu iki satır eşdeğerdir:
p birNesne
print birNesne.inspect, "\n"
```

Erişgeçleri kolay hale getirmek

Her örneğin bir erişim yöntemine ihtiyacı olmasına rağmen, Ruby standart tarzlar için daha elverişli bir yol sunar.

Kısayol	*Etkisi*
attr_reader :v	**def v; @v; end**
attr_writer :v	**def v=(value); @v=value; end**
attr_accessor :v	**attr_reader :v; attr_writer :v**
attr_accessor :v, :w	**attr_accessor :v; attr_accessor :w**

Şimdi bunun avantajlarından faydalanalım ve bilgimizi tazeleyelim. Öncelikle otomatik olarak oluşturulmuş bir okuyucu ve yazıcı olup olmadığına bakarız ve yeni bilgiyi **inspect**'in içine dahil ederiz:

```
ruby>   class Meyve
    |    attr_accessor :nitelik
    |    def inspect
    |      "#{@nitelik} bir #{@cesit}"
    |    end
    |  end
   nil
ruby>   meyve.nitelik = "olgun"
   "olgun"
ruby>   meyve
   "olgun bir muz"
```

Meyveyle biraz daha eğlence

Eğer kimse olgunlaşmış meyvemizi yemezse, parasını almak için beklemeliyiz.

```
ruby>   class Meyve
    |    def durumu
    |      @nitelik = "çürük"
    |    end
    |  end
   nil
ruby>   meyve
   "olgun bir muz"
ruby>   meyve.durumu
   "curuk"
ruby>   meyve
   "çürük bir muz"
```

Ancak buralarda oynarken, küçük bir sorunla karşılaştık. Üçüncü bir meyve yaratmaya çalıştığımızda ne olur? Örnek değişkenlerin onlara değer atanmadan var olmadıklarını hatırlayın.

```
ruby>   f3 = Meyve.new
ERR: failed to convert nil into String
```

Burada yakınılan **inspect** yöntemidir ve geçerli bir sebebimiz var. **f3**'e özellik atamadan bir meyvenin çeşidi ve niteliği hakkında bir rapor istedik. Eğer istersek, **inspect** yönteminin **defined?** yöntemini de kullanarak sadece tanımlanmış meyveleri rapor etmesini sağlayabiliriz ancak bu iş hala kullanışsız olur, çünkü her meyvenin bir çeşidi ve niteliği olduğuna göre bu ikisinin her zaman tanımlı olduğundan emin olmamız gerekir. Bu ileriki bölümün konusudur.

27. Nesnenin İlklendirilmesi

Geçen bölümdeki Meyve sınıfı, biri meyvenin çeşidini diğeri de niteliğini açıklayan iki örnek değişkene sahipti. Bunu yapmamızın nedeni bir kaç meyve için karakteristiğin önemli olmamasıydı. Genel bir **inspect** yöntemi oluşturmaktı. Ruby örnek değişkenlerin her zaman hazırlanmış olduğunu garanti eden bir yol sunuyor.

`initialize` yöntemi

Ruby yeni bir nesne yaratıldığı zaman **initialize** denen bir yöntem arar ve çalıştırır. Yapabileceğimiz basit şeylerden biri her örnek değişkene öntanımlı bir **initialize** yöntemi koymak ve böylece **inspect** yöntemine söyleyebilecek bir şeyler sağlamaktır.

```
ruby>   class Meyve
    |    def initialize
    |      @cesit = "elma"
    |      @nitelik = "olgun"
    |    end
    |  end
   nil
ruby>   f4 = Meyve.new
   "olgun bir elma"
```

Öntanımlı değerlerin değiştirilmesi

Bazen öntanımlı değerlerin pek de anlamlı olmadığı zamanlar olabilir. Öntanımlı bir meyve çeşidi gibi birşey olabilir mi? Her meyvenin yaratıldığı zaman kendi çeşidini belirlemesi daha tercih edilebilir bir durumdur. Bunu yapmak için **initialize** yöntemine bir argüman ekleriz. Burada bahsetmeyeceğimiz nedenlerden dolayı **new**'e verdiğiniz her argüman **initialize** yöntemi tarafından alınmış olur.

```
ruby>   class Meyve
    |    def initialize( k )
    |      @cesit = k
    |      @nitelik = "olgun"
    |    end
    |  end
   nil
ruby>   f5 = Meyve.new "mango"
   "olgun bir mango"
ruby>   f6 = Meyve.new
ERR: (eval):1:in `initialize': wrong # of arguments(0 for 1)
```

Esnek ilklendirme

Yukarıda gördüğümüz gibi bir argümanı **initialize** yöntemi ile ilişkilendirirseniz boş değer vermeniz durumunda hatayla karşılaşırsınız. Daha düşünceli davranmak istersek, değer verildiği zaman o değeri kullanabilir, verilmediği zamansa öntanımlı bir değer atayabiliriz.

```
ruby>   class Meyve
    |    def initialize( k="elma" )
    |      @cesit = k
    |      @nitelik = "olgun"
    |    end
    |  end
   nil
ruby>   f5 = Meyve.new "mango"
   "olgun bir mango"
ruby>   f6 = Meyve.new
   "olgun bir elma"
```

Öntanımlı değerleri sadece **initialize** için değil tüm yöntemler için uygulayabilirsiniz.

Bazen bir nesneyi hazırlamak için birçok yol hazırlamak yararlı olabilir. Bu kılavuzun kapsamının dışında olmasına rağmen Ruby, yöntemleri aşırı yüklemeyi sağlayan nesne yansıtmaya (*object reflection*) ve değişken uzunluklu argüman listesine izin verir.

28. Ivır Zıvır

Bu bölüm pratik bir kaç konuyu kapsar.

Deyim sınırlayıcılar

Bazı diller deyimleri sonlandırmak için, noktalı virgül **;** gibi noktalama işaretleri gerektirir. Ruby bunun yerine **sh** ve **csh**'ın geleneğini takip eder. Birden fazla deyim noktalı virgülle ayrılmalıdır ancak bir satırın sonuna noktalı virgül koymanız gerekmez, satırsonu karakteri bir noktalı virgülmüş gibi davranır. Eğer bir satır tersbölü (****) ile biterse o zaman onu takip eden satırsonu karakteri dikkate alınmaz, bu da çok sayıda satıra bölünmüş tek bir satır oluşturmanızı sağlar.

Yorum Satırları

Niçin yorum yazmalıyız? İyi yazılmış bir kodun kendi kendini belgelemesi yanında başkalarının da çiziktirdiğimiz koda bakabileceği ihtimalini göz ardı etmemeliyiz. Öte yandan siz ve kendiziniz iki gün önceki sizle farklı kişilersiniz; hangimiz bir bölüm sonra durup, "bunu yazdığımı hatırlıyorum, ama ne cehennemi kastettim!" dememişizdir ki?

Bazı deneyimli yazılımcılar, güncelliğini yitirmiş ve tutarsız yorumların hiç yorum yazmamaktan daha kötü olduğuna dikkat çekerler. Tabii ki yorumlar okunabilir bir kodun vekili olmamalıdırlar; eğer kodunuz yeterince açık değilse muhtemelen yanlışlarla doludur. Ruby'yi öğrenirken yorum yazmaya sık sık ihtiyacınız olacak–ancak daha iyi duruma geldiğiniz zaman basit, şık ve okunabilir kodlar yazmaya başladığınızda daha az yorum satırına ihtiyaç duyacaksınız.

Ruby yorum satırını belirtmek için **#** işareti kullanarak betik dillerinin izlediği geleneksel yolu izler. **#** ile başlayan her satır yorumlayıcı tarafından gözardı edilir.

Ayrıca büyük yorum bloklarını mümkün kılmak için Ruby yorumlayıcısı "**=begin**" ve "**=end**" arasındaki satırları da gözardı eder.

```
#!/usr/bin/env ruby

=begin
**********************************************************************
  Bu bir yorum öbeği. Daha sonra kodunuzu okuyanların (kendiniz de dahil)
  rahatlıla kodunuzu anlayabilmeleri için bir şeyler yazabilirsiniz.
  Yorumlayıcı bu öbeği görmezden gelir. Her satır başında '#' işaretine
  ihtiyacımız yok.
**********************************************************************
=end
```

Kodunuzu düzenlemek

Ruby yorumlayıcısı kodları okuduğu gibi işletir. Derleme aşaması gibi bir şey söz konusu değildir; eğer birşey henüz okunmamışsa basitçe *tanımsızdır*.

```
#sonuç bir tanımsız yöntem ("undefined method") hatası olacaktır:

print successor(3),"\n"

def successor(x)
  x + 1
end
```

Bu, ilk bakışta öyle gibi gözükse de, kodunuzu baştan aşağı bir tasarım harikasına dönüştürmeniz için zorlamaz. Yorumlayıcı, bir yöntem tanımlamasıyla karşılaştığında tanımlanmamış başvuruları da rahatlıkla ekleyebilir ve yöntem çağrıldığında tanımlı olabileceğinden emin olabilirsiniz:

```
# fahrenhaytı santigrata çevirir,
# iki adımdan olusur.

def f_to_c(f)
  scale(f - 32.0)
end

def scale(x)
  x * 5.0 / 9.0
end

printf "%.1f iyi bir sıcaklık.\n", f_to_c(72.3)
```

Perl ya da Java'da alıştığınızdan daha az elverişli olsa da prototip yazmadan C yazmaktan daha az kısıtlayıcıdır. Bir kaynak kodunun altındaki kodu en üste koymak her zaman çalışır. Üstelik başlangıçta göründüğünden daha az can sıkıcıdır. Tanımlamak istediğiniz davranışı yaptırmak için en ağrısız ve uygun yol dosyanın başında bir **main** işlevi tanımlamak ve en altta çağırmaktır.

```
#!/usr/bin/env ruby

def main
  # Ana mantığı burada açıklayın...
end

# ... destek kodunu buraya koyun ...

main # ... burada çalıştırmaya başlayın.
```

Bu yöntem ayrıca Ruby'nin karmaşık yazılımları, okunabilir, tekrar kullanılabilir ve mantıksal ilişkilendirilmiş parçalara bölünmesini desteklemesine yardımcı olur. Daha önce modüllere ulaşmak için **include** özelliğinin kullanımını görmüştük. Ayrıca **load** ve **require** özellikleri de yararlı olabilir. **load** başvurduğu dosya kopy-alanıp yapıştırılmış gibi işlev görür (C'deki #include ön işlemcisine benzer). **require** biraz daha yetenekli bir özelliktir: kodun bir kere ve ihtiyaç duyulduğu zaman yüklenmesini sağlar. **load** ve **require** arasındaki diğer farklılıklar için Ruby SSS'a bakın.

İşte bu kadar!

Bu kılavuz Ruby'de yazılım geliştirmeye başlamanıza yetecek kadar bilgi içerir. Eğer başka sorularınız olursa Ruby Başvuru Kılavuzu[B11]na göz atabilirsiniz. Ruby SSS[B12] ve Ruby Başvuru Kütüphanesi[B13] de yararlı kaynaklardan bir kaç tanesidir.

Şansınız bol olsun, iyi kodlamalar!

29. Kılavuz Hakkında

Bu kılavuz çeşitli yerlerde yayınlanmış ve bir çok da çevirisi vardır. Güncel ingilizce sürüm rubyist.net[B14]'de bulunur. Eğer güncelliğini kaybetmiş bir sürümle karşılaşırsanız, yansının yöneticisini lütfen uyarın.

Belge Geçmişi

- Özgün Japonca sürüm Yukihiro Matsumoto <matz (at) netlab.co.jp> tarafından yazılmıştır.
- İlk İngilizce çeviri[B15] GOTO Kentaro <gotoken (at) notwork.org> ve Julian Fondren tarafından yapıldı.
- Mark Slagell <slagell (at) ruby–lang.org> tarafından belge yeniden çevrildi ve bazı eklemeler yapıldı.

GNU Free Documentation License

Version 1.2, November 2002

1. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world–wide, royalty–free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front–matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front–Cover Texts or Back–Cover Texts, in the notice that says that the Document is released under this License. A Front–Cover Text may be at most 5 words, and a Back–Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine–readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ascii without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard–conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally

available, and the machine–generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

3. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front–Cover Texts on the front cover, and Back–Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine–readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer–network location from which the general network–using public has access to download using public–standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front–matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—-for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front–Cover Text, and a passage of up to 25 words as a Back–Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front–Cover Text and one of Back–Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10. TERMINATION

    You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

11. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
  Copyright (C)  year  your name.
  Permission is granted to copy, distribute and/or modify this document
  under the terms of the GNU Free Documentation License, Version 1.2
  or any later version published by the Free Software Foundation;
  with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
  A copy of the license is included in the section entitled "GNU
  Free Documentation License".
```

If you have Invariant Sections, Front–Cover Texts and Back–Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front–Cover Texts being list, and with the Back–Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Notlar

a) Belge içinde dipnotlar ve dış bağlantılar varsa, bunlarla ilgili bilgiler bulundukları sayfanın sonunda dipnot olarak verilmeyip, hepsi toplu olarak burada listelenmiş olacaktır.

b) Konsol görüntüsünü temsil eden sarı zeminli alanlarda metin genişliğine sığmayan satırların sığmayan kısmı ¬ karakteri kullanılarak bir alt satıra indirilmiştir. Sarı zeminli alanlarda ¬ karakteri ile başlayan satırlar bir önceki satırın devamı olarak ele alınmalıdır.

(1) Unix altında çalışıyorsanız zaten uçbirim ANSI uyumludur; DOS altında bunu sağlamak için ANSI.SYS veya ANSI.COM yüklemelisiniz.

(B4) http://www.ruby-«doc.org/docs/UsersGuide/rg/eval.rb

(B10) http://www.ruby-«lang.org/en/man-«1.4/

(B11) http://www.ruby-«lang.org/en/man-«1.4/

(B12) http://dev.rubycentral.com/faq/rubyfaq.html

(B13) http://dev.rubycentral.com/ref/

(B14) http://www.rubyist.net/~slagell/ruby/

(B15) http://www.math.sci.hokudai.ac.jp/~gotoken/ruby/ruby-«uguide/

Bu dosya (ruby–ug.pdf), belgenin XML biçiminin TEXLive ve belgeler-xsl paketlerindeki araçlar kullanılarak PDF biçimine dönüştürülmesiyle elde edilmiştir.

20 Nisan 2007