Giriş

```
puts 'Merhaba Dünya'
```

Programlama

- MİB
- Bellek
- Giriş/Çıkış

- Programı belleğe yükle (İşletim Sistemi)
- Denetimi programa ver (İşletim Sistemi)
- Bellekte sırayla çalışan buyruklar

- Sınırlı sayıda buyruklar $\rightarrow$ Buyruk kümesi (instruction set)
- Buyruğu veya işlem sonucunu tutan kayıt alanları $\rightarrow$ Kaydediciler (registers)
- Aritmetik ve Mantıksal işlemleri yerine getiren birim $\rightarrow$ ALB (ALU)

Oyuncak Makine

- Kaydediciler: sadece 1 tane $\rightarrow$ Akümülatör (Birikeç)
- Buyruk kümesi: 14 buyruk

İki sayıyı topla

```
start  load this
       add result
       store result
       load that
       add result
       store result
       load result
       print
       stop
this   3
that   5
result 0
```

Kaynak kod

Problemin çözümünü ilgili programlama dilinin sözcük ve kurallarıyla anlatan tarif

MİB'nin anladığı tek dil: makine dili

- Programın çalıştırılması: Kaynak kodla yapılan tarifin MİB'nin dilindeki buyruklara dönüştürülmesi

Tarifin hayata geçirilmesi ("programın çalıştırılması")

- Önce kaynak kodun tamamını makine diline çevir $\rightarrow$ Derleme (compile)
- Kaynak kodu (tarifi) bir programa girdi olarak vererek tarifteki her cümlenin gereğinin MİB'ne bu program tarafından yaptırılmasını sağla $\rightarrow$ Yorumlama (interprete)

- Kaynak kod bir tarifin hayata geçmesi için tek başına yeterli değil
- Sadece makine dilinde yazılan bir tarif doğrudan yeterli (ki onda bile bir tür işlemeye ihtiyaç var, bk. örnekte yapılan bellek ilklendirmeleri)
- Bir derleyiciye veya bir yorumlayıcıya ihtiyaç var

Derleme

(Aşırı basitleştirme içerir)

- Kaynak kodu hedef MİB'in buyruklarından oluşan makine diline çevir
- Bu işlem program çalıştırılmadan önce **bir seferliğine** yapılır
- Derlenmiş biçimdeki program çalıştırılır
- Bu modelde program işletim sistemi tarafından doğrudan yüklenerek çalıştırılıyor

```c
#include <stdio.h>

static int this   = 3;
static int that   = 5;
static int result = 0;

int main()
{
    result = this + that;

    printf("%d\n", result);

    return 0;
}
```

Nesne kodu

Object code

- Derleme sonucunda elde edilen imajı (ör. çalıştırılabilir kipte bir ikili program dosyası) anlatır
- Kaynak kodun devamında yer alan bir terim
- Terimde geçen nesneyi "Nesne Yönelimli"deki (Object Oriented) nesne ile karıştırmayın

Yorumlama

(Aşırı basitleştirme içerir; derlemeye göre daha da aşırı)

- "Yorumlayıcı" programı belleğe yükle
- Yorumlayıcı kaynak kodu okur; artık denetim yorumlayıcı programda
- Yorumlayıcı, kaynak koddaki anlamlı çalıştırma cümlelerini (ör. satırlar) sırayla yorumlar
- Yorumlama? Cümleyi anlamlandır ve gereğini MİB'ne (onun anladığı buyruklarla) yaptır
- Bu modelde program işletim sistemi tarafından yüklenenen bir yorumlayıcının aracılığıyla çalıştırılıyor

```
this = 3
that = 5

result = this + that

puts result
```

Çalışma zamanı

Önemli bir terim: "çalışma zamanı" $\rightarrow$ **runtime**

- Programın çalıştırılması süresince geçen zaman dilimini anlatıyor
- Derlenen programlarda, derlenmiş program imajının belleğe yüklenip MİB tarafından çalıştırılmaya başlandığı andan, sonlandığı ana kadar geçen süre
- Yorumlanan programlarda, kaynak kodun yorumlayıcı tarafından çalıştırılmaya başlandığı andan, sonlandığı ana kadar geçen süre

Dinamik programlama dilleri

- Kaynak kod üzerinde çalışma zamanı dışında yapılan başka işlemler de var
- Bu süreçler de farklı şekilde adlandırılabiliyor, ör. derleme zamanı (compile time)
- Yorumlanan bir program dilinde kararlar çalışma zamanında dinamik olarak alındığından bu dillere "dinamik program dilleri" de deniliyor
- "Dinamik" teriminin karşı tarafındaki terim: "Statik"
- Bu nedenle kaynak kod üzerinde çalışma zamanı dışında gerçekleşen süreçler genel olarak "statik" terimiyle vasıflandırılıyor
- Örnek: Statik kod çözümlemesi

Yüksek/alçak seviye diller

Bilinmesinde yarar olan bir terim çifti

- Bir programlama dilinde sunulan soyutlamalarla ifade kabiliyeti ne kadar yüksek ise dil de o kadar "yüksek seviye" (high-level) bir dil oluyor
- Karşısındaki terim "alçak seviye" (low-level); soyutlamalar daha az, donanıma daha yakın (ve bir o kadar da denetim olanağı)
- Yüksek/alçak diyerek dilin kalitesine ilişkin bir sıfat oluşturmuyoruz, bu teknik bir tartışma
- Bunlar göreceli terimler, mutlak anlamda kullanmayın
- Örnek: Go, Ruby'ye göre alçak-seviye bir dildir, ama C'ye göre yüksek-seviyelidir
- Yorumlanan (dinamik) diller derlenen dillere göre hemen hemen daima yüksek-seviyeli

Derleme/Yorumlama

- "Hesaplama" (computing) süreçlerini anlamak için yararlı
- Günümüzde artık çok anlamlı terimler değil (bk. JIT, bytecode, garbage collector)
- Pek çok gerçeklemede "yorumlama" sürecinde bir tür derleme yapılıyor (çalışma zamanında)
- Derleme bazen doğrudan MİB'i hedeflemiyor, sanal bir MİB hedefleniyor (ör. Java sanal makinesi)

- Bu terimler programlama dilinin gerçeklemesiyle ilişkili; programlama diline ilişkin mutlak bir özelliği anlatmıyor
- Bir programlama dili, en azından kuramsal olarak, hem derlenen hem yorumlanan biçimde gerçeklenebilir
- Fakat dil (ortaya çıkışında belirlenmiş) doğası itibarıyla bir tür gerçeklemeyi daha etkin kılar veya bir tür gerçeklemeyi teknik olarak çok zorlaştırır
- "Derlenen/yorumlanan dil" yerine "Kaynak kodun derlenerek/yorumlanarak çalıştırılması öngörülen dil"
- Ör. Ruby, Python, Javascript yorumlanarak çalıştırılması öngörülen diller
- Ör. C, Go, Rust derlenerek çalıştırılması öngörülen diller

Derlenen dil

Avantajlar

- Çalışma zamanında yorumlama olmadığından (veya minimize edildiğinden) çok daha hızlı
- Bellek kullanımı daha az
- Sorunlar program çalışmadan önce (derleme aşamasında) yakalanabilir
- Lojistiği daha kolay; hedef platform için derlenmiş programın kurulumu yeterli, ayrıca bir yorumlayıcı kurmanıza gerek yok

Dezavantajlar

- Yazılması daha maliyetli (derleyiciyi mutlu etmek zorundasınız, tip bildirimleri gibi daha ayrıntılı tarifler gerekiyor)
- Çalışma zamanı üzerinde denetiminiz olmadığından "dinamik" işler çeviremezsiniz
- (C gibi en azından bir kısım dilde) Çalışma zamanında güvenlik açıkları

Yorumlanan dil

Avantajlar

- Geliştirme süresi daha kısa (arada zeki bir yorumlayıcı var, daha kısa lafla çok iş)
- Çalışma zamanı denetlenebildiğinden "dinamik" işler çevrilebilir
- Çalışma zamanı denetlenebildiğinden basit güvenlik açıkları yaşanmaz
- Daha "taşınabilir" (portable); yazdığınız kodun ilgili platformda çalışması için yorumlayıcının o platformda kurulu olması yeterli (fakat bk. lojistik)

Dezavantajlar

- Daha yavaş
- Daha fazla bellek tüketimi
- Çalışma zamanında yaşanan sürpriz hatalar (derlenebilseydi çalıştırmadan önce yakalanabilirdi)
- Artan lojistik yük (yorumlayıcı kurulumu gerekiyor)

Yorumlanması öngörülen bir dilde programın çalışma süresi ve bellek tüketimini artırmak pahasına, programın geliştirme süresini azaltıyoruz

- Birim zamanda daha fazla iş
- Daha çabuk hayata geçen fikirler

(Ama ile başlayacak eleştirilere açık bir yargı)

Dinamik (yorumlanan) bir dilde geliştirici konforu hedeflenir

- Sistem kaynaklarını (MİB, bellek vs) daha konforsuz bir durumda tutmak pahasına

Günümüz trendleri

- Ayrım yine korunmakla birlikte her iki türün en iyi özellikleri dillere eklenebiliyor
- Yorumlanan dillerde tip bildirimleri
- Derlenen dillerde çalışma zamanını denetleyen eklemeler (ör. çöp toplayıcı)
- Teknik olarak geçerli, fakat pratikte hatalı kod parçalarını geliştirme aşamasında yakalayan zengin statik çözümlemeler ("lint"leme)

Ruby

Değişken

İsimlendirilmiş bellek hücresi

- Bellek hücresinde (bir tür) veri var
- Veriye anlamlı bir isimle erişiyoruz

```
kur = 8.96
dolar = 100.0

tl = kur * dolar
```

```
oran = 18.0 / 100
fiyat = 100.0

kdv = fiyat * oran
```

İsimlendirme

Söz dizimi (sentaks) kuralları

- İlk karakter İngilizce alfabedeki küçük/büyük harflerden biri veya alt tire (_) olmalı
- Varsa devam eden karakterlerde ilkine ilave olarak rakamlar kullanılabilir (ama ilk karakter rakam olamaz)

- Sadece değişkenler değil, metot adları, sabitler, sınıf/modül adları da (Ruby'de bunlar da birer sabit isim) isimlendirmenin kapsamında
- Bu isimlere genel olarak "tanımlayıcı" (identifier) deniliyor
- İsimlendirme söz dizimi kuralları $\rightarrow$ Tanımlayıcı söz dizimi kuralları

Uygun isimlendirme kod okunurluğunu çok artırır

- Her program bir öykü veya (uzunluğuna göre) bir roman
- İsimler bu öykünün kahramanları
- Anlamlı isimler öykünün okunmasını kolaylaştırıyor

Türkçe karakterler?

ç	ğ	ı	ö	ş	ü
Ç	Ğ	İ	Ö	Ş	Ü

- `ı` ve `İ`'ye dikkat! (`i` ve `I` Türkçe'ye özgü değil)
- Değişken adlarında Türkçe karakter çoğu durumda kullanabiliriz, ama kullanmamalıyız

- Programlama evrensel bir etkinlik
- Programlama dillerinin anahtar kelimeleri de İngilizce
- İsimlendirmeleri enternasyonal yapmakta yarar var; özellikle her dilden geliştiricinin katkı sunabileceği açık kaynak projelerde

```
exchange_rate = 8.96
usd = 100.0

tl = exchange_rate * usd

tax_rate = 18.0 / 100
price = 100.0

tax = price * tax_rate
```

İfadeler

- Değerlendirmeye (evaluation) konu ögeler
- Değerlendirme? Hesaplama, değer verme
- Örnek: `exchange_rate * usd`
- Bu bir aritmetik ifade
- Örnek: `usd = exchange_rate * usd`
- Bu (aritmetik ifade içeren) bir "atama" (assignment) ifadesi

Her ifade bir değer döner (değerlendirme sonrası)

- Ruby'de ilkel değerlerin bizzat kendisi de ifade
- Örnek: `100.0`
- `usd = 100.0` atama ifadesinde önce sağ taraf değerlendirilir, dönen değer (`100.0`) sol taraftaki değişkene atanır

Ruby'de her şey bir ifadedir

- IRB'de girilen bir satır `enter` tuşu ile yorumlayıcıya gönderilir
- IRB, satırı bir bütün halde ifade olarak yorumlar
- İfadenin döndüğü değer `#=>` ile belirtilir

Aritmetik operatörler

- Operatör $\rightarrow$ İşleç
- Sayısal türde değerleri operatörlerle düzenleyerek dönüş değeri yine sayısal türde olan aritmetik ifadeler kurabiliyoruz
- Sayısal tür? Tam sayı, Gerçel sayı, Rasyonel sayı
- Aritmetik operatörler beklediğiniz gibi: +, −, ∗, /
- Ayrıca iki operatör: % modülüs ve ∗∗ üs alma operatörleri

Sayısal tür

- Tam sayı ve gerçel sayılar
- Gerçel sayılarla kurulan ifadelere dikkat! `18.0 / 100` yerine `18 / 100` yazılırsa?

Ruby'de Rasyonel sayıların gösterimi için özel bir söz dizimi kullanılıyor

```
tax_rate = 18/100r
```

- Daha okunur
- Bunu nasıl kullanacağız? Göründüğü gibi, ör. `18/100r * 100.0`

Tür dönüşümleri yapılabilir

- Değer nesneleri üzerinde çalıştırılacak iki metot: `to_i` ve `to_f`
- Bir değeri tamsayıya çevirmek için `to_i`, ör. `18.9.to_i #=> 18`
- Bir değeri gerçel sayıya çevirmek için `to_f`, ör. `18/100r.to_f #=> 0.18`
- Değer bu dönüşümü desteklemeli

Fonksiyonlar

```
value = Math.sin(0.5236) # 0.5236 ~ Pi/6 ~ 30 derece
```

- Matematikte aşina olduğumuz bir trigonometrik fonksiyon: `sin`
- `<fonksiyon>(girdi listesi) → çıktı`
- Fonksiyonlara giriş değerlerini argümanlar yoluyla iletiyoruz, örnekte 30 derece `sin` fonksiyonuna iletiliyor
- Fonksiyon (isminin yansıttığı) hesaplamayı yapıp bir değer dönüyor, örnekte `0.5`
- Ruby'de bir fonksiyona geçirilen argümanlar etrafında parantez kullanmanız her zaman gerekmiyor

Matematiksel fonksiyonlardan bir parça farklı olarak programlamada yazacağınız fonksiyonlar:

- Hiç argüman istemeyebilir
- Birden fazla argüman isteyebilir
- Bir değer dönmeyebilir
- Dönecekse sadece tek bir değer döner (bazı dillerde, ör. Go, birden fazla değer dönülebilir)

Fonksiyon veya Metot

Ruby gibi Nesne Yönelimli dillerde fonksiyon yerine metot adlandırması tercih ediliyor

- Bu bir isimlendirme inceliği (bazı nedenleri var, gelecekte daha ayrıntılı değineceğiz)
- Bundan sonra fonksiyon değil metot diyeceğiz

Metotlarla ilk karşılaşmamız:

```
puts 'Merhaba Dünya'
```

- `puts` bir metot (yani fonksiyon)
- Metotlara genel olarak bir nesne üzerinde `.` operatörüyle erişiyoruz
- Fakat bu örnekte metot bir nesne üzerinden değil doğrudan çağrılıyor
- Bu konuya gelecekte değineceğiz

Nesne Nokta Metot notasyonu

Notasyona dikkat edin! `18.to_f #=> 18.0`

- Noktanın solunda bir değer: `18`, sağında ise bir metot: `to_f` bulunuyor
- Noktanın solundaki "değer" aslında bir "nesne" (object)
- Nesnelere `.` operatörü yoluyla bir mesaj iletiyoruz
- Mesaj $\rightarrow$ Metot
- Nesne mesajın gereğini yerine getiriyor (ilgili metot çağrılıyor)

Ruby'de hemen her şey bir nesne

- Nesneleri `.<metot>` söz dizimiyle uyarıyoruz

İlkel veri türleri

- Sayısal türler ilkel (primitive) veri türlerinin en yaygın örneği
- Pek çok programlama dilinde bir diğer önemli veri türü: “dizgi” (string)

Dizgi

```
message = 'Merhaba Dünya'

puts message
```

- Örnekteki `'Merhaba Dünya'` değeri bir dizgi (string)
- Çift tırnak veya tek tırnak kullanabiliriz

```ruby
who = 'Dünya'
message = "Merhaba #{who}"

puts message
```

- Çift tırnakta Ruby dizgi değerini özel olarak yorumlar → "Dizgi Enterpolasyonu" (String Interpolation)
- #{} arasına istediğiniz karmaşıklıkta bir Ruby kodu yazabilirsiniz
- Yorumlayıcı #{} arasındaki kodu bir ifade olarak değerlendirir ve dönüş değerini yerine koyar
- Bu örnekte tek tırnak kullanılsaydı `message` dizgisi olduğu gibi (literal) yorumlanacaktı

Dizgiler programlama dillerinde çok temel bir veri türü

- Her bir tespih tanesi bir “karakter” (character, char) olan bir tespih gibi

Karakter

- Dizgilerin yapıtaşları; kabaca harfler, rakamlar ve noktalama işaretleri
- Bunlara ilave kontrol karakterleri var: boşluk, satır sonu, sekme gibi
- Karakterler belirli sayıda bitlik bir bilgiyle kodlanıyor
- En bilineni 7 bitlik ASCII: American Standard Code for Information Interchange
- Türkçe gibi dile özgü karakterler ASCII tabloda yok
- Bunun yerine günümüzde UTF-8 gibi daha evrensel kodlama standartları kullanılıyor
- Yine de ASCII tabloya hakim olmalısınız (örneğin UTF-8 ASCII'nin bir tür üst sürümü)

ASCII Tablo

Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@
1	SOH (start of heading)	33	!	65	A
2	STX (start of text)	34	"	66	B
3	ETX (end of text)	35	#	67	C
4	EOT (end of transmission)	36	$	68	D
5	ENQ (enquiry)	37	%	69	E
6	ACK (acknowledge)	38	&	70	F
7	BEL (bell)	39	'	71	G
8	BS (backspace)	40	(	72	H
9	TAB (horizontal tab)	41	)	73	I
10	LF (NL line feed, new line)	42	*	74	J
11	VT (vertical tab)	43	+	75	K
12	FF (NP form feed, new page)	44	,	76	L
13	CR (carriage return)	45	-	77	M
14	SO (shift out)	46	.	78	N
15	SI (shift in)	47	/	79	O
16	DLE (data link escape)	48	0	80	P

Ruby'de karakterler özel bir veri türü değildir

- Ama örneğin C gibi bazı programlama dillerinde çoğunlukla `char` adında özel bir veri türüdür (Ruby'den farklı olarak C programlama dilinde dizgi veri türü yoktur)

Ruby'de bir karakterin ASCII tablodaki onluk tabanda kodunu öğren: `.ord`

```
'a'.ord
' '.ord
"\n".ord
"\t".ord
```

Onlu tabanda verilen bir kodu karakteri içeren dizgiye çevir: `.chr`

```
97.chr
```

Özel karakterler

- `"\n"` $\rightarrow$ Satır sonu
- `"\t"` $\rightarrow$ Sekme
- Bunlar en yaygınları, bunların dışında ters bölü karakteriyle nitelendirilen başka kodlar da var

Beyaz boşluk (whitespace)

- Kabaca; boşluk, satır sonu ve sekme karakterlerine deniliyor (ama başkaları da var)
- Dizgi içinde kullanılmadığında, kaynak kod ayrıştırılırken bu karakterler göz ardı edilir veya kodun söz dizimsel olarak farklı parçalarını birbirinden ayırır

Temel bazı dizgi metotları

```ruby
string = gets

puts string.size # string.length
puts string.empty?
puts string.chomp
puts string.chop
puts string.upcase
puts string.downcase
puts string.capitalize
puts string.tr '_', ' '
puts string.delete '_'
puts string.strip
puts string.start_with? '2021'
puts string.end_with? '2021'
puts string.delete_prefix '2021'
puts string.delete_suffix '.rb'
```

String birleştirme (“concatenate”)

```
string = ''

string << 'Cezmi'
string << ' '
string << 'Seha'

puts string
```

`# frozen_string_literal: true`?

- Bu bir pragma
- Kaynak koddaki tüm dizgi literallerini öntanımlı olarak "değiştirilemez" yapıyor
- Bu sayede aynı dizgi literali için bellek ayırmak gerekmiyor
- Yorumlayıcıya verdiğiniz açık sözün denetlenmesi sağlanıyor (hata yakalama)

```
city = 'Samsun'.freeze
city << '55' # hata
```

yerine

```
# frozen_string_literal: true

city = 'Samsun'
city << '55' # hata
```

`# frozen_string_literal: true` yapılırsa birleştirmeler nasıl?

- IRB'de sorunu görmeyebilirsiniz (görmek için `RUBYOPT="--enable-frozen-string-literal" irb`)

```ruby
string = String.new '' # string = '' yerine
```

(Dikkat! `String.new`'i argümansız çalıştırırsanız karakter kodlaması ASCII oluyor)

Akış denetimi

- Kod akışını farklı kod yollarına bölen koşul deyimleri
- Kod bloklarının etkinleştirilmesi belirli koşulların sağlanmasına bağlanıyor

Örnek: Katsayıları verilen kuadratik (İkinci derece) bir denklemde çözüm var mı?

Diskriminant pozitif olmalı (alan bilgisi)

```
a, b, c = 1.0, 0.0, 1.0

delta = b ** 2 - 4 * a * c

if delta >= 0.0
  puts 'Çözüm var'
end
```

- `if`, `end` birer anahtar kelime
- Koşul ifadesi: `delta >= 0.0` aritmetik karşılaştırma içeren bir mantık (lojik) ifade
- Aritmetik karşılaştırma operatörü >= "büyük veya eşit"
- Gerçel sayı karşılaştırmalarını böyle yapmayın, sorunu görebiliyor musunuz?
- İlk satırda paralel atama yapılıyor (kötüye kullanmayın)

Gövdesi tek satır olan `if` deyimlerini tek satırda yazabiliyoruz

```
a, b, c = 1.0, 0.0, 1.0

delta = b ** 2 - 4 * a * c

puts 'Çözüm var' if delta >= 0.0
```

```ruby
a, b, c = 1.0, 0.0, 1.0

delta = b ** 2 - 4 * a * c

puts 'Çözüm var' unless delta < 0.0
```

- Yeni anahtar kelime: `unless`
- Negatif lojik için kullanılıyor
- Özellikle `!` değillemeleri içeren basit ifadelerde yararlı
- Okunurluğu (yerine göre) bir parça arttırıyor

Örnek: Katsayıları verilen kuadratik denklemin gerçel kökleri neler?

```
a, b, c = 1.0, 0.0, 1.0

delta = b ** 2 - 4 * a * c

if delta >= 0.0
  delta_sqrt = Math.sqrt(delta)

  p, q = (-b - delta_sqrt) / 2 * a, (-b + delta_sqrt) / 2 *

  puts "Kökler: (#{p}, #{q})"
else
  puts 'Çözüm yok'
end
```

- Yeni anahtar kelime: `else`

Örnek: Verilen 3 sayı geçerli bir üçgenin kenar uzunlukları mı?

Üçgen kuralı (alan bilgisi): Sayılardan herhangi ikisinin toplamı üçüncüden **daima** büyüktür

```
a, b, c = 3, 4, 5

if a + b > c && a + c > b && b + c > a
  puts "Geçerli üçgen"
else
  puts "Geçerli üçgen değil"
end
```

- Koşulda mantıksal (lojik) bir ifade, önermeler && "ve" mantık operatörüyle bağlanmış
- Önermelerin her biri aritmetik karşılaştırma, > "büyüktür"

Örnek: Kullanıcıdan bir tam sayı iste

```ruby
print 'Lütfen bir sayı girin: '
string = gets.chomp

if string == ''
  puts 'Hiç bir şey girmediniz.'
elsif (number = Integer(string, exception: false))
  puts "Girdiğiniz sayı #{number}"
else
  puts "Geçersiz sayı girdiniz: #{string}"
end
```

- Yeni anahtar kelime: `elsif`, çoklu koşul deyimleri
- `Integer(string, exception: false)` hatalı dönüşümde `nil` değeri dönüyor
- Koşul ifadesi içinde atama yapabilirsiniz (kötüye kullanmayın), yeter ki parantezlerle niyetinizi açık hale getirin
- Boş dizgi denetimi daha deyimsel nasıl yapılabilir?

`nil`

Düpedüz yokluğu veya geçerli bir değerin yokluğunu anlatan "sözde değer"

- Mantıksal bağlamda `false` ile benzer sonuçlar üretiyor
- Yani bu bir "falsy" değer
- Diğer dillerde de kısmen benzer değerler var; ör. C, C#, Java'da `null`

Doğruluk/Yanlışlık

Basit iki kural

1. **Ruby'de değeri `false` ve `nil` olan her ifade yanlıştır**
2. **Yanlış olmayan her şey doğrudur**

```ruby
number = Integer('geçersiz', exception: false) #=> nil

if number
  puts 'Doğru'
else
  puts 'Yanlış'
end
```

Bazen `nil` değerini açıkça denetlemeniz gerekebilir

```ruby
number = Integer('geçersiz', exception: false) #=> nil

if number.nil?
  puts 'Evet: nil'
end
```

Yeri gelmişken... "Ruby'de her şey bir ifade" demiştik, örneği inceleyelim

```
flag =
  if x == 0
    false
  else
    true
  end
```

- Herşeyin ifade olmadığı pek çok dilde böyle bir kod yazamazsınız (`if` pek çok dilde bir ifade değil bir deyimdir)

```
flag =
  if x == 0
    false
  else
    true
  end
```

Bu kodu basitçe şöyle yazabilirdiniz

```
flag = x == 0
```

Metot

İsmiyle çağrılarak çalıştırılabilir (bir veya çoğunlukla birden fazla satırlık) kod parçası

- Farklı girdilerle tekrar tekrar yapılan hesaplamalar için her seferinde aynı kodu yazmanız gerekmiyor
- Hesaplama girdileri çağırma zamanında verilen parametrelerle değiştirilebilir

Örnek: Katsayıları verilen kuadratik (İkinci derece) bir denklemin gerçel köklerini bul

```
def calculate_roots(a, b, c)
  delta = b ** 2 - 4 * a * c

  if delta >= 0.0
    delta_sqrt = Math.sqrt(delta)

    p, q = (-b - delta_sqrt) / 2 * a, (-b + delta_sqrt) / 2

    puts "Kökler: (#{p}, #{q})"
  else
    puts 'Çözüm yok'
  end
end

a, b, c = 1.0, 0.0, 1.0

calculate_roots(a, b, c)
```

- Yeni anahtar kelime: `def`
- `a`, `b` ve `c` metot argümanları
- Çağırma zamanında metoda bu argümanlarla değerleri geçiriyoruz

- Metot argümanlarıyla çağırma zamanında kullanılan değişkenlerin aynı isimde olması gerekmiyor

```
a2, a1, a0 = 1.0, 0.0, 1.0

calculate_roots(a2, a1, a0)
```

- Değerleri hiç bir değişken kullanmadan da geçirebiliriz

```
calculate_roots(1.0, 0.0, 1.0)
```

Örnek: Verilen 3 sayı geçerli bir üçgenin kenar uzunlukları mı?

```ruby
def validate_triangle(a, b, c)
  if a + b > c && a + c > b && b + c > a
    puts "Geçerli üçgen"
  else
    puts "Geçerli üçgen değil"
  end
end

validate_triangle(3, 4, 5)
```

Metotlar çoğunlukla bir hesap yaptıktan sonra bize bir sonuç döner

- Her iki örnekte de bir sonuç dönmedik
- Son örnekte aşama aşama giderek gösterelim

```ruby
def validate_triangle(a, b, c)
  if a + b > c && a + c > b && b + c > a
    return true
  else
    return false
  end
end

if validate_triangle(3, 4, 5)
  puts "Geçerli üçgen"
else
  puts "Geçerli üçgen değil"
end
```

- Yeni anahtar kelime: `return`
- Kullanıldığı noktada metotu sonlandırarak verilen değeri çağıran tarafa dönüyor

Her metot tek bir iş yapmalı

- İlk örnekte bu kural nasıl ihlal edilmiş?

- Ruby zaten `true`/`false` hesabını yapıyor, biz ayrıca neden hesap ediyoruz?
- Ruby'de metottan çıkarken etkin olan son satır aynı zamanda dönüş değeridir
- Çoğu zaman `return` ile açık dönüş yapmamız gerekmez
- Ruby'de `return` deyimini "erken çıkış"lar için kullanın

```ruby
def validate_triangle(a, b, c)
  a + b > c && a + c > b && b + c > a
end

if validate_triangle(3, 4, 5)
  puts "Geçerli üçgen"
else
  puts "Geçerli üçgen değil"
end
```

Örnek: Kullanıcıdan bir tam sayı iste

```ruby
def getnum
  print 'Lütfen bir sayı girin: '
  string = gets.chomp

  if string.empty?
    puts 'Hiç bir şey girmediniz.'
  elsif (number = Integer(string, exception: false))
    puts "Girdiğiniz sayı #{number}"
  else
    puts "Geçersiz sayı girdiniz: #{string}"
  end

  number
end
```

İsimlendirmeler çok önemli

- Ruby'de metot adlarının sonunda `?` ve `!` karakterlerini kullanabilirsiniz
- `true` veya `false` değer dönen metotlara "predicate method" diyoruz
- `?` sonlandırma karakteri bir metotun "predicate" olduğunu nitelendirmekte kullanılan bir konvansiyon
- Bu sadece bir konvansiyon, metot adının sonunda `?` karakteri olunca sihirli bir işlem gerçekleşmiyor
- İsimlendirmeleri çok daha anlamlı yapıyor

Örnek: Katsayıları verilen kuadratik (İkinci derece) bir denklemde çözüm var mı?

Diskriminant pozitif olmalı (alan bilgisi)

```
def has_solution?(a, b, c)
  (b ** 2 - 4 * a * c) >= 0.0
end

if has_solution?(1.0, 0.0, 1.0)
  puts "Çözüm var"
else
  puts "Çözüm yok"
end
```

```ruby
def triangle?(a, b, c)
  a + b > c && a + c > b && b + c > a
end

if triangle?(3, 4, 5)
  puts "Geçerli üçgen"
else
  puts "Geçerli üçgen değil"
end
```

Üçlü operatörü

Ternary operatörü

```
def has_solution?(a, b, c)
  (b ** 2 - 4 * a * c) >= 0.0
end

puts "Çözüm #{has_solution?(1.0, 0.0, 1.0) ? 'var' : 'yok'}

def triangle?(a, b, c)
  a + b > c && a + c > b && b + c > a
end

puts "Geçerli üçgen#{triangle?(3, 4, 5) ? '' : ' değil'}"
```

Kapsam

```
a, b, c = 1.0, 0.0, 1.0

def calculate_roots(a, b, c)
  delta = b ** 2 - 4 * a * c

  if delta >= 0.0
    delta_sqrt = Math.sqrt(delta)

    p, q = (-b - delta_sqrt) / 2 * a, (-b + delta_sqrt) / 2

    puts "Kökler: (#{p}, #{q})"
  else
    puts 'Çözüm yok'
  end
end

calculate_roots(a, b, c)

puts delta #=> ?
```

Metotlar dışarıya kapalı bir kutu gibi davranır

- Metot gövdesi bir kapsam ("scope") belirler: yerel kapsam ("local scope")
- Yerel kapsamdaki bir değişken dışarı sızmaz (ör. `delta`)
- Benzer şekilde metot dışındaki hiç bir değer argümanlar yoluyla verilmedikçe içeri sızmaz
- Metodun dış dünyayla yegane kontak noktaları: giriş argümanları ve dönüş değeri

İsimlendirilmiş argümanlar

```ruby
def calculate_roots(a:, b:, c:)
  delta = b ** 2 - 4 * a * c

  if delta >= 0.0
    delta_sqrt = Math.sqrt(delta)

    p, q = (-b - delta_sqrt) / 2 * a, (-b + delta_sqrt) / 2

    puts "Kökler: (#{p}, #{q})"
  else
    puts 'Çözüm yok'
  end
end

calculate_roots(a: 1.0, b: 0.0, c: 1.0)
```

- Veriliş sırasıyla anlamlandırılan argümanlar: “pozisyonel argümanlar”
- Argümanları veriliş sırasıyla değil de isimleriyle belirtsek?
- Özellikle birden fazla sayıda argüman geçirmemiz gerektiğinde yararlı
- Neyin ne olduğunu çağırma zamanında karıştırmamış oluyoruz

Öntanımlı argümanlar

```ruby
def calculate_roots(a: 0.0, b: 0.0, c: 0.0)
  delta = b ** 2 - 4 * a * c

  if delta >= 0.0
    delta_sqrt = Math.sqrt(delta)

    p, q = (-b - delta_sqrt) / 2 * a, (-b + delta_sqrt) / 2

    puts "Kökler: (#{p}, #{q})"
  else
    puts 'Çözüm yok'
  end
end

calculate_roots(a: 1.0, c: 1.0)

calculate_roots
```

Öntanımlı argümanlar "pozisyonel argümanlar" için de geçerli

```
def calculate_roots(a = 0.0, b = 0.0, c = 0.0)
  delta = b ** 2 - 4 * a * c

  if delta >= 0.0
    delta_sqrt = Math.sqrt(delta)

    p, q = (-b - delta_sqrt) / 2 * a, (-b + delta_sqrt) / 2

    puts "Kökler: (#{p}, #{q})"
  else
    puts 'Çözüm yok'
  end
end

calculate_roots

calculate_roots(1.0, 0.0, 1.0)
```

- İlk çağrıda işe yaradı

İkinci derece denklem örneğinde bir sorun daha var

- “Bir metot tek bir iş yapmalı” kuralı ihlal edilmiş
- Bunu düzeltmek şu aşamada zor
- Ruby’de metotlar sadece tek bir değer dönebilir
- Birden fazla değeri tek bir değer halinde dönmek gerekiyor
- Bunun yolu? Diziler

Döngü

Bilgisayarın en temel kabiliyeti: bir işlemi tekrar tekrar yapabilmek

Örnek: Kullanıcıdan geçerli bir tamsayı al

```
01  def getnum
02    print 'Lütfen bir sayı girin [ENTER sonlandırır]: '
03
04    while !(string = gets.chomp).empty?
05      number = Integer(string, exception: false)
06      if number
07        return number
08      end
09
10      print "Geçersiz sayı: '#{string}'.  Lütfen tekrar g
11    end
12
13    nil
14  end
```

- Yeni anahtar kelime: `while`
- Çoğu durumda “... oldukça/olmadıkça” veya “... olduğu sürece/olmadığı sürece” gibi okuyabilirsiniz

Sözde kod

```
Girdi al, bu boş bir dizgi olmadığı sürece
  dizgiyi tamsayıya çevir
  eğer dönüşüm geçerli ise tamsayıyı dön

  hata görüntüle
```

```ruby
def getnum
  print 'Lütfen bir sayı girin [ENTER sonlandırır]: '

  until (string = gets.chomp).empty?
    number = Integer(string, exception: false)
    return number if number

    print "Geçersiz sayı: '#{string}'.  Lütfen tekrar girin
  end

  nil
end
```

- Yeni anahtar kelime: `until`
- `if`/`unless` ilişkisine benzer şekilde `while`/`until`
- Olumsuz lojik için kullanılıyor
- Basit ifadeler kullanıldığı sürece okunurluğu bir parça arttırıyor

Örnek: Sayı tahmini

Verilen bir aralık içinde belirlenmiş bir tamsayıyı tahmin et

İyileştirmeler

- Kod tekrarını nasıl önleriz?
- Döngü üzerinde tam denetim nasıl kurarız?
- Kullanıcıya ipucu verebilir miyiz? “Büyük/küçük” gibi
- Maksimum deneme sayısını sınırlayabilir miyiz?
- Sayı aralığını değişken yapabilir miyiz?
- UX: Her tahminde deneme sayısını da kullanıcıya bildirebilir miyiz?
- Sayı aralığı ve maksimum deneme sayısını program çalıştırılırken girebilir miyiz?

Yeni

- Yeni anahtar kelime: `loop`: açık uçlu döngüler kurmaya yarıyor
- Yeni anahtar kelime: `break`: döngü sonlandırmaya yarıyor

- `break` anahtar kelimesi `while`, `until` ve gelecekte göreceğiniz tüm döngülerde, döngüyü kırmak için kullanılır
- `loop`'a özgü değil

- Sabitler: büyük harf ile başlayan (ve çoğunlukla hepsi büyük harften oluşan) tanımlayıcılar
- `ARGV`: komut satırı argümanlarını tutan (sabit isimli bir) dizi
- `||=` öntanımlı değer atama özdeyişi
- `STDIN`: standart girdiya karşı düşen sabit
- `STDIN.gets`: daima standart girdiden (ör. klavye) okuma yapan özdeyiş

Diziler

Birbiriyle ilişkili (bir küme oluşturan) değerleri barındıran veri türü

```
days = ['pazartesi', 'salı', 'çarşamba', 'perşembe', 'cuma
```

- Kümedeki her değere tek bir tanımlayıcı üzerinden erişiyoruz, nasıl?
- İndis kullanarak, ilk değerin indisi 0, her seferinde 1 artıyor

```
days[0]   #=> 'pazartesi'
days[1]   #=> 'salı'
days[6]   #=> 'pazar'
days[7]   #=> nil
days[100] #=> nil
```

Pek çok programlama dilinde saymaya `0`'dan başlanır, **`1`'den değil**

- Bu durumda son elemanın indisi ne oluyor? `<dizi uzunluğu> - 1`

Söz dizimi

- Birden fazla satıra yayabiliriz

```
days = [
  'pazartesi',
  'salı',
  'çarşamba',
  'perşembe',
  'cuma',
  'cumartesi',
  'pazar',
]
```

- Son elemandan sonra da , kullanmaya izin var (ama topluluk stilinde hoş bakılmıyor)
- Özel olarak dizgilerden oluşan dizileri ilklendirmek için %w[] kullanabiliriz

```
days = %w[pazartesi salı çarşamba perşembe cuma cumarte
```

Değer atama

Basitçe ilgili indisteki elemana değer atamamız yeterli

```
days[0] = 'monday'
```

Temel metotlar

- Dizi uzunluğu: `days.size` veya `days.length` (eşdeğer)
- İlk eleman: `days.first`
- Son eleman: `days.last`

Negatif indisler

Diziye sondan erişmek için kullanılıyor

```
days[-1]           #=> 'pazar'
days[-2]           #=> 'cumartesi'
days[-7]           #=> 'pazartesi'
days[-days.size]   #=> 'pazartesi'
days[-8]           #=> nil
days[-100]         #=> nil
```

Dizi sonuna yeni eleman eklemek

Çok sık yaptığımız bir işlem

```
a = []
a << 3
a << 5
a << 9
a #=> [3, 5, 9]
```

Dizinin başına (veya seçilen bir indisten önce/sonraya) elema eklemek? Düşündüğünüz kadar sık ihtiyaç duyulacak bir işlem değil

```
a = []
a.unshift 3
a.unshift 5
a.unshift 9
a #=> [9, 5, 3]
```

Dizgiden diziye, Diziden dizgiye

`split` ve `join`: bu metot çiftine özel bir yer ayırıyoruz

- `split`: bir dizgiden (tekil, skalar) bir dizi (çoğul, vektör) üretiyor
- `join` : bir diziden (çoğul, vektör) bir dizgi (tekil, skalar) üretiyor

```
name   = 'Ahmet Yılmaz'
names  = name.split       #=> ['Ahmet', 'Yılmaz']
dashed = names.join('-') #=> 'Ahmet-Yılmaz'
```

Dizide dolaşmak

En sık yapılan işlem, bir tür döngü

- Şu ana kadar öğrendiğiniz döngü deyimleriyle yapılabilir
- Ama bunu yapmayın, Ruby'de dizilerde dolaşmak için çok daha güçlü yöntemler var

each

Ruby'de `Enumerable` modülünde tanımlı olan ve dizi türündeki tüm nesnelerin cevap verdiği bir metot

```
days.each do |day|
  puts day
end
```

Ruby'de dizilerde dolaşmak için daima `each` ve daha sonra gösterilecek `Enumerable` metotlarını kullanın

- `while`, `until`, `loop` (ve anlatmaya gerek görmediğimiz `for`) gibi döngü deyimlerini **kullanmayın**
- Bu deyimlere çoğunlukla dizi içermeyen düzensiz döngülerde ihtiyaç duyacaksınız
- Bu uyarı özellikle diğer programlama dillerinden bilgi transfer edeceklere önemli

each_with_index

Dolaşırken indis bilgisine ihtiyacımız varsa?

```
days.each_with_index do |day, i|
  puts "#{i}: #{day}"
end
```

Bloklar

Metotlara geçirilen eylemler

- Daha teknik bir anlatımla isimsiz (anonim) işlevler
- Ruby'nin çok önemli bir özelliği

Blokları anlayabilmemiz için hikayede biraz geriye gitmek zorundayız

- Ruby'de hemen her şey akıllı bir "nesne"
- Uygun metotlarla uyararak nesnelerin istediğiniz davranışı göstermesini sağlayabilirsiniz
- Örneğin dizgiler birer nesne

```
irb(main):001:0> 'This is a test'.length
=> 14
irb(main):002:0> 'This is a test'.upcase
=> THIS IS A TEST
```

- **Sayılar da öyle**

```
irb(main):003:0> 3.times { puts 'Test' }
Test
Test
Test
=> 3
```

Son örneğe yoğunlaşalım

```
3.times { puts 'Test' }
```

- 3 bir tamsayı nesnesi, `times` bu nesnenin bir metotu
- Öyle ki bu metota hangi eylemi tekrarlayacağını bildirebiliyorsunuz
- Nasıl? Eylemi gerçekleyen bir kod bloğuyla

  ```
  { puts 'Test' }
  ```

- Kod bloklarını `{...}` yerine `do ... end` ile de yazabiliriz

```
3.times do
  puts 'Test'
end
```

- Stil olarak tek satırlık bloklarda kıvrık parantezler, birden fazla satıra yayılan kod blokları için `do ... end` tercih ediyoruz

Metafor

- İngiltere kraliçesi Türkiye'ye resmi ziyaret yapacak; Kraliçenin ülkede bulunduğu sürece yemekleri nasıl olacak?
- 1: İngiliz protokolü Dışişleri protokolüne kraliçenin ülkedeyken yiyeceği yemeklerin listesini veya tarifini iletebilir
- 2: Kraliçe yemekleri yapacak özel ahçısını bizzat yanında getirebilir
- İlki klasik yöntem, bir metota (ör. `yemek_hazırla`) veri girilmesi (yemek listesi veya tarifler)
- İkincisinde ise metota bir eylem veriliyor, ahçının eylemleri

Bir işleve eylemde ihtiyaç duyacağı bilgileri argümanlar üzerinden geçirebiliriz.

```
puts 'Test'
```

- `puts`: Neyi görüntüleyeyim?
- Çağıran: "Test" dizgisini

Aynı diyaloğu `times` için kurgulayalım.

- `times`: **kaç** defa **ne** yapacağım?

Ama bu soru hatalı.

- `times` metodu uyarılırken **kaç** defa bilgisini zaten alıyor, nasıl?

  ```
  3.times ...
  ```

- Diyalog `times` ile değil 3 tamsayı nesnesi arasında gerçekleşmeli

Diyalog:

- 3: Ne istiyorsun?
- Çağıran: Sen **defa** (yani 3 defa) bir şey yapmanı.
- 3: Tamam, ben **defa** ne yapacağım?
- Çağıran: `puts 'Test'` ('Test' dizgisini görüntüle).

Sonuçlar:

- Nesnelere sadece veri değil eylem de bildirilebiliyor
- Bu sayede çeşitlenebilir davranışlar elde edebiliyoruz
- Tekrarlama eylemiyle (`times`), tekrarlanacak eylemi (`puts "Test"`) ayırıyoruz
- `times` bir metot, `puts 'Test'` ise bu metota geçirilen bir blok

Bloklar metotumsu (veya fonksiyonumsu) şeyler

- Blok argümanları (varsa) |...| karakterleri arasında (metotlardaki parantezler)
- Blok dönüş değeri metotlardaki gibi etkin olan son satır
- Dikkat! Bloğun dönüş değerini bloğu alan metotun dönüş değeriyle karıştırmayın
- Nihai "dönüş değeri"ni bloğun verildiği metot belirliyor

```
3.times { 'Merhaba' } #=> 3 döner
```

- Bloklarda da erken çıkış için (`return` yerine) `break` veya `next` tercih ediyoruz

```ruby
%w[samsun istanbul izmir adana].each do |city|
  next if city.include? 'a'

  puts city
end
```

```ruby
%w[samsun istanbul izmir adana].each do |city|
  break unless city.include? 'a'

  puts city
end
```

Blok kapsamı

"Bloklar metotumsu (veya fonksiyonumsu) şeyler" demiştik

- Blok içinde tanımlanan bir değişken bloğa özgüdür, blok dışına çıkamaz

```
%w[samsun istanbul izmir adana].each do |city|
  next if city.include? 'a'

  puts city
end

puts city
```

- Bu kod neden hata veriyor?
- Düzeltmek için ne yapılabilir?

- Bazen bir blokta üretilen bir değeri dış kapsama taşımak isteyebiliriz

```
cities_with_a = []

%w[samsun istanbul izmir adana].each do |city|
  cities_with_a << city if city.include? 'a'
end

puts cities_with_a
```

- Sonuç: dış kapsama taşınacak değeri tutacak değişkeni bloktan önce tanımlayın (`nil` gibi bir değerle de olsa)
- Fakat bunu yapmanın hemen hemen daima daha iyi bir yolu vardır (örnek için `select` veya `collect`)

Sayılabilirler modülü

`Enumerable` modülü

- Koleksiyonlar üzerinde yapılabilecek pek çok işlemi barındıran bir "katıştırma" (mixin) modülü
- Koleksiyon? Genel olarak Dizi ve daha sonra görülecek olan Sözlük veri yapıları
- Koleksiyon? Daha teknik bir anlatımla `each` metodunu sağlayan tüm nesneler
- İşlemler? Sıralama, seçme, eşleme, arama

- En temel metot: `each`, koleksiyonlarda dolaşmak için
- Fakat "sonuç üreten" dolaşmalarda `each` yerine kullanmanızın daha uygun olacağı metotlar var
- `Enumerable` modülü bu metotları sağlıyor
- Yeter ki nesne `each` metotunu gerçeklesin (bu metotu `Enumerable` sunmuyor, sadece yararlanıyor)

Seçme (Süzme): `select`

- `N` boyutlu bir dizide belirli bir koşulu sağlayan ögeleri seçiyoruz
- Koşul bir "blok"la ifade ediliyor: "blok" `true` değer üretmişse ise seç, aksi halde bırak
- Seçilen ögelerle `M` boyutlu yeni bir dizi oluşturuyoruz; öyle ki `M <= N`

```ruby
[1, 2, 3, 4, 5].select { |num| num.even? } #=> [2, 4]

%w[foo bar].select { |str| str == 'foo' } #=> ['foo']
```

- `select` metotunun diğer adları: `filter`, `find_all`

- Seçme işlemi yeni bir dizi üretir
- Yeni dizinin boyutu özgün diziden büyük olamaz, genellikle daha küçüktür

Diziler üzerinde seçme yaparken `each` değil `select` tercih edin

- `Enumerable` modülünü de barındıran Standart kitaplığı iyi tanıyın
- Her iş için en uygun çözümü kullanın
- Aksi halde kendi "kötü" çözümünüzü geliştirme tehlikesi söz konusu

İlişkili metot: `reject`

```ruby
[1, 2, 3, 4, 5].reject { |num|  num.even? } #=> [1, 3, 5]
```

İlişkili metot: `find`

```
[1, 2, 3, 4, 5].find { |num|  num.even? } #=> 2
```

- Bu metotun daima tek bir değer döndüğüne dikkat edin ("find" yapamamışsa `nil`)
- Amacınız koşul sağlandığında devam etmeden ilgili ögeyi dönmek ise daima `find` kullanın
- Diğer adı: `detect`

Eşleme: `map`

- `N` boyutlu bir dizideki her ögeyi, çoğunlukla o ögeyi bir işlemden geçirerek, yeni bir ögeyle eşleştiriyoruz
- İşlem bir "blok"la ifade ediliyor: "blok"un (girdi olarak verilen ögeyle) ürettiği değerle eşleme yap
- Eşleşen ögelerle yine aynı boyutta (`N`) yeni bir dizi oluşturuyoruz

```ruby
[1, 2, 3, 4, 5].map { |num|  num ** 2 } #=> [1, 4, 9, 16, 2
```

```ruby
%w[foo bar].map { |str| str.upcase } #=> ['FOO', 'BAR']
```

- map metotunun diğer adı: `collect`

- Eşleme işlemi yeni bir dizi üretir
- Yeni dizinin boyutu özgün diziye daima eşittir (daha büyük veya küçük olamaz)

Diziler üzerinde eşleme yaparken `each` değil `map` tercih edin

- `Enumerable` modülünü de barındıran Standart kitaplığı iyi tanıyın
- Her iş için en uygun çözümü kullanın
- Aksi halde kendi "kötü" çözümünüzü geliştirme tehlikesi söz konusu

Koleksiyon predikatörleri

`all?`, `any?`, `one?`, `none?`

```
%w[ant bear cat].all?  { |word| word.length >= 3 } #=> tru
%w[ant bear cat].any?  { |word| word.length >= 3 } #=> tru
%w{ant bear cat}.one?  { |word| word.length == 4 } #=> tru
%w{ant bear cat}.none? { |word| word.length == 5 } #=> tru
```

Diziler üzerinde mantıksal sonuç üreten bir değerlendirme yaparken `each` veya `select` değil bu metotları tercih edin

- `Enumerable` modülünü de barındıran Standart kitaplığı iyi tanıyın
- Her iş için en uygun çözümü kullanın
- Aksi halde kendi "kötü" çözümünüzü geliştirme tehlikesi söz konusu

İyi

```ruby
ok = original.all?  { |word| word.length >= 3 }

if ok
  ...
end
```

Kötü

```
original = %w[ant bear cat]
longest_than_two = original.select { |word| word.length >=

if longest_than_two.length == original.length
  ...
end
```

Çirkin

```
original = %w[ant bear cat]

ok = true
original.each do |word|
  if word.length < 3
    ok = false
    break
  end
end

if ok
  ...
end
```

Sıralama: `sort`

- `N` boyutlu bir diziyi "belirli bir kurala" göre sıralıyoruz
- Kural bir "blok"la ifade ediliyor: verilen öge çiftini karşılaştırarak "küçük, eşit, büyük" sonucu üret
- Sıralanan yine aynu boyutta (`N`) yeni bir dizi oluşturuyoruz

```ruby
[1, 2, 3, 4, 5].sort { |a, b| b <=> a } #=> [5, 4, 3, 2, 1
```

```ruby
%w[foo bar].sort #=> ['bar', 'foo']
```

- Sıralama kuralı zorunlu değil: verilmemesi halinde öntanımlı kurallar uygulanıyor
- Öntanımlı kuralllar? Dizgilerin alfabetik sıralaması, sayıların büyüklüğüne göre sıralanması

- Sıralama işlemi yeni bir dizi üretir
- Yeni dizinin boyutu özgün diziye daima eşittir (daha büyük veya küçük olamaz)

Diziler üzerinde sıralama yaparken `each` değil `sort` tercih edin

- `Enumerable` modülünü de barındıran Standart kitaplığı iyi tanıyın
- Her iş için en uygun çözümü kullanın
- Aksi halde kendi "kötü" çözümünüzü geliştirme tehlikesi söz konusu

Karşılaştırma işlemi

- Sıralama yaparken her seferinde iki ögeyi karşılaştırıyoruz: ör. a ve b
- İşlem başarımını eniyileştirmek için öge çiftlerinin nasıl seçileceği önemli bir "algoritmik problem"
- Buna "sıralama algoritması" diyoruz

- Literatürde “quick, merge, heap, shell, bubble” sort gibi iyi bilinen bazı algoritmalar var
- Standart kitaplıktaki metotları kullanarak sıralama yaparken “sıralama algoritması”yla ilgilenmiyoruz
- Bu bir gerçekleme detayı: Ruby sizin için en uygun (optimum) veya ona yakın algoritmayı seçiyor
- Bizim ilgilendiğimiz kraşılaştırma işlemi; ki çoğu durumda onu bile vermemiz gerekmiyor

Karşılaştırma işlemi 3 ihtimallı bir bilgiyi üretmeli: “küçüktür”, “eşittir”, “büyüktür”

- “Küçüktür”: negatif bir sayı, çoğunlukla `-1`
- “Eşittir”: `0`
- “Büyüktür”: pozitif bir sayı, çoğunlukla `1`

```ruby
[1, 2, 3, 4, 5].sort do |a, b|
  if b < a
    -1 # herhangi bir negatif değer de olabilir
  elsif b > a
    1 # herhangi bir pozitif değer de olabilir
  else
    0
  end
end #=> [5, 4, 3, 2, 1]
```

Örnekte dizideki öge çiftinin `a, b` sırasıyla verildiğine fakat karşılaştırmanın `b, a` sırasıyla yapıldığına dikkat edin: "Ters sıralama"

<=>: “Spaceship” işleci

- Örnekteki karşılaştırmayı her seferinde yapmanız gerekmiyor
- <, > ve == karşılaştırmaları ilkel nesnelerin zaten cevap verdiği metotlar (bu bilgi nesne de zaten var)
- O halde örnekteki karşılaştırmaları <=> isimli özel bir metotta toplayabiliriz

(Dikkatli bakılırsa cepheden bir tür “uzay gemisi” görülebilir)

```ruby
def <=>(other)
  if self > other
    1 # herhangi bir negatif değer de olabilir
  elsif self < other
    1 # herhangi bir pozitif değer de olabilir
  else
    0
  end
end
```

- Böyle bir <=> metotu `Integer`, `String` gibi ilkel veri türü nesnelerine ekleyebiliriz
- Bu yapıldığında `sort` metotuna karşılaştırma işlemi vermek bile gerekmez
- Neden? Ruby "türdeş" dizide dolaşırken ilgili türün <=> metotunu uyarır

`sort` metotunu genellikle hiç bir blok tanımlamadan kullanıyoruz

- Öntanımlı davranışın var olmadığı veya uygun olmadığı durumlarda bir blok lullanıyoruz
- Fakat çoğu durumda ilgili nesneye ait sınıfta özel bir <=> metotu yazmak daha doğru

Bir diziyi ters sıralamak için her seferinde önceki örneklerdeki gibi mi kod yazacağız?

```
[1, 2, 3, 4, 5].sort.reverse #=> [5, 4, 3, 2, 1]
```

- Önce sırala, sonra tersle
- Dikkat! Sadece `reverse` diziyi sıralamadan tersler

Bir dizgi dizisini dizgi uzunluklarına göre sıralamak istersek?

İlişkili metot: `sort_by`

```
%w[apple pear fig].sort_by { |word| word.length }  #=> ['f
```

- Karşılaştırma işlemini ögelerin kendisiyle değil bir özelliği üzerinden yapıyoruz
- "Blok" nesnenin kendisinden bu özelliğe ait değeri üretiyor
- Öyle ki karşılaştırma bu değerlere ait öntanımlı <=> işleciyle gerçekleşiyor

Ünlemli metotlar

Verilen koleksiyonla eş boyutta bir koleksiyon üreten metotları ele alalım: `map`, `sort`, `reverse`

- Bu metotlar her seferinde yeni bir dizi üretiyor
- Örneğin bellekte büyük yer kaplayan 100,000 ögeli bir diziyi sıraladığımızda bellekte bir o kadar yer işgal eden yeni bir dizi üretiyoruz
- Sıralanmamış özgün diziye sonraki aşamalarda çoğunlukla ihtiyacımız yok
- Bellekten tasarruf için yerinde sıralasak? Yani sıralanmış dizi eskisine ayrılan bellek alanına kayıtlansa?

```
> a = [1, 2, 3, 4, 5] #=> [1, 2, 3, 4, 5]
> b = a.reverse       #=> [5, 4, 3, 2, 1]
> a                   #=> [1, 2, 3, 4, 5]
```

yerine

```
> a = [1, 2, 3, 4, 5]
> a.reverse! #=> [5, 4, 3, 2, 1]
> a          #=> [5, 4, 3, 2, 1]
```

Tek yapmamız gereken metotu sonunda ! olan çeşidiyle değiştirmek

```
> a = [1, 2, 3, 4, 5]
> a.reverse!
> a #=> [5, 4, 3, 2, 1]
> a.sort!
> a #=> [1, 2, 3, 4, 5]
> a.map! { |i| i ** i }
> a #=> [1, 4, 9, 16, 25]
```

Ters sıralamada daima ünlemli metotları kullanın

İyi

```
a.sort!.reverse!
```

Kötü

```
a.sort.reverse
# veya
a.sort.reverse!
```

- Ruby'de sonunda ? karakteri bulunan metotları görmüştük
- ? "doğru mu, değil mi?"yi çağrıştırıyordu
- Sondaki ! ise genel olarak "dikkatli ol, bu metotun bir yan etkisi var" mesajını iletiyor
- Örneklerdeki yan etki? Özgün dizinin artık yok olması, değişime uğraması
- Bu yan etki bazen istemediğiniz bir şey olabilir
- Örneğin: "Sıralanmamış diziye de ihtiyacım var; isteğe göre farklı kurallarla birden fazla sıralama yapmam gerekiyor"

Bu bir konvansiyon

- Her metotun ünlemli bir karşılığı yok, sadece anlamlı durumlarda var
- Sona ! koyduğunuzda sihirli bir şey olmuyor, **bu sadece bir konvansiyon**
- Bu konvansiyon, (daima zor olan) isimlendirmelerde kullanacağınız bir enstrüman
- Mevcut bir ismi "recycle" ederek yeniden kullanabiliyorsunuz, üstelik daha anlamlı bir seçenek olarak
- Ünlemli metotu, eğer bu anlamlıysa, siz gerçekleyeceksiniz
- Bunu yaparken "dikkat, yan etkisi var" semantiğine sadık kalın

Standart kitaplık

Standart kitaplık bir hazine

- Kıymetli pek çok mücevher (metot) var
- Bunları tanıyın (aksi halde ne olacağını biliyorsunuz)

Diğer “sayılabilirler” metotları

- `max`, `min`, `max_by`, `min_by`, `sum`
- `uniq`, `zip`, `tally`

Diğer dizi metotları

- `compact`, `flatten`
- `sample`, `rotate`
- `permutation`, `combination`
- `product`, `transpose`
- `intersection`, `union`, `difference`