



# Ruby 101

Uğur “vigo” Özyılmazel

---

# Table of Contents

Giriş	1.1
İnceleme	1.2
Bölüm 1	1.3
Ruby Hakkında	1.3.1
Kurulum	1.3.2
İnteraktif Kullanım	1.3.3
Ruby Komutu ve Parametreleri	1.3.4
Bölüm 2	1.4
Syntax (Söz Dizimi) ve Rezerve Kelimeler	1.4.1
Değişkenler	1.4.2
Ön Tanımlı ve Pseudo (Gerçek Olmayan) Değişkenler	1.4.3
Operatörler	1.4.4
Global Constants (Genel Sabitler)	1.4.5
Bölüm 3	1.5
Methods (Fonksiyonlar)	1.5.1
Blocks (Bloklar)	1.5.2
Proc ve Lambda	1.5.3
Conditional Statements (Koşullar)	1.5.4
Bölüm 4	1.6
Object	1.6.1
Number	1.6.2
String	1.6.3
Array	1.6.4
Hash	1.6.5
Symbol	1.6.6
Class	1.6.7
Module	1.6.8
Bölüm 5	1.7
Enumeration ve Iteration	1.7.1
Ranges	1.7.2
File System ve IO (Dosya Sistemi)	1.7.3
Exception Handling	1.7.4

---

Kernel Modülü	1.7.5
Bölüm 6	1.8
Monkey Patching	1.8.1
Regular Expressions	1.8.2
Time ve Date Nesneleri	1.8.3
Ruby Paketleri: RubyGems	1.8.4
Paket Yöneticisi: Bundler	1.8.5
Komut Satırı (Command-Line) Kullanımı	1.8.6
Meta Programming	1.8.7
Kod Yazma Tarzı (Style Guide)	1.9
Gerçek Hayat Ruby Örnekleri	1.10
Neden Ruby?	1.10.1
Ruby ve TDD/BDD/CI	1.10.2
Kendi Rubygem'imizi yapalım!	1.10.3
Sinatra ve Web	1.10.4
Yazar Hakkında	1.11

---

# Ruby 101 Kitabı

book passing genel 77%

## Önsöz

Kitap yazmak hep hayalini kurduğum bir şeydi. Hem kendi işime yarayacak hem de başkalarının işini görecek bir kitap olmalıydı. Aslında bir sene önce bu işe soyundum ama bir türlü fırsat bulamadım.

Kafamda kabaca planlar yaptım hep ama son noktayı bir türlü koyamadım. [Gitbook.io](https://gitbook.io) bu konuda çok işime yaradı. Hem beni fişekledi hem de [GitHub](https://github.com) ile kolay entegre olması kendimi organize etmem açısından çok rahat oldu.

Hep [O'Reilly](https://oreil.ly)'nin **Pocket** yani cep kitaplarına bayılmışımdır. Hem boyut itibariyle hem de içerik anlamında. Sürekli yanınızda taşıyabileceğiniz, içinde konusuyla ilgili her şeyin kompakt bir şekilde bulunduğu kaynak.

Amacım, bu kitaplar tadında, her zaman yanınızda bulunabilecek, tabiri caizse **başucu** kitabı hazırlamak.

Kitabı hazırlarken en çok zorlandığım kısım İngilizce'den anlamlı Türkçe metinler çıkartmak oldu. Bazı şeyleri İngilizce olarak ifade etmek çok kolay, fakat bazı durumlarda tam Türkçe anlamlı karşılık bulmak gerçekten zor oluyor.

Prensip olarak **Developer** (*Yazılım Geliştiren Kişi*) denen insanın **default** olarak İngilizce bilmesi gerektiğine inanıyorum. Neden? Örneğin milyonlarca açık-kaynak projenin bulunduğu GitHub'da herkes İngilizce konuşuyor.

Takıldığınız bir konuda, GitHub'da yorumları okumanız gerekecek. Hatta bazen siz bir şey soracaksınız. **Issue**'lara bakacaksınız, **Pull Request** yapacaksınız. Gördüğünüz gibi bir cümlede iki tane İngilizce terim. Bunlar evrensel. Bilmemiz gerekiyor yoksa çuvallarız :)

Özellikle pek çok şeyi olduğu gibi İngilizce olarak kullanmak istedim. Tabii ki Türkçe anlamını da yazdım fakat, genel olarak kullandığım terminoloji Ruby ve yazılım geliştirme terminolojisi.

Örneğin **Constant** dediğimde bunun ne anlama geldiğini anlamış olmanız gerekiyor. Ya da **Instance** dediğimde, bunun sınıftan oluşturulmuş bir nesne olduğunu anlamamız gerekiyor.

Yazılım dünyası ne yazık ki İngilizce ve tüm kaynaklar da İngilizce. Bu bakımdan orijinal kelimeleri ve terminolojiyi öğrenmemiz, bilmemiz şart :)

# Lisans Mevzusu

Prensip olarak, GitHub'a **Public** olarak koyduğum her şey, herkes tarafından her türlü şekilde kullanılabilir. Bence Public olarak sürülen bir şey (*ki ben eski Amiga'cı Public Domain'ci biriyim*) herkesin tepe tepe kullanabilmesi amaçlı olmalıdır.

Her türlü lisans olayına karşıyım. Eğer bir tür lisanslama yapacaksanız kendinize saklayın :)

Bu kitap ananızın ak sütü gibi hepinize helal olsun. Umarım işinize yarar!

Kitabı online olarak okumak için:

<https://vigo.gitbooks.io/ruby-101/content/>

## Tamamlanma Durumu (%78)

### Bölüm 1 (%94)

- Ruby Hakkında
- Kurulum
- İnteraktif Kullanım
- Ruby Komutu ve Parametreleri (%98)

### Bölüm 2 (%100)

- Syntax (Söz Dizimi) ve Rezerve Kelimeler
- Değişkenler
- Ön Tanımlı ve Pseudo (Gerçek Olmayan) Değişkenler
- Operatörler
- Global Constants (Genel Sabitler)

### Bölüm 3 (%100)

- Methods (Fonksiyonlar)
- Blocks (Bloklar)
- Proc ve Lambda
- Conditional Statements (Koşullar)

### Bölüm 4 (%100)

- Object
- Number
- String
- Array

- Hash
- Symbol
- Class ve Module

## **Bölüm 5 (%58)**

- Enumeration ve Iteration
- Ranges
- File System ve IO (Dosya Sistemi) (%90)
- Exception Handling (--)
- Kernel Modülü (--)

## **Bölüm 6 (%50)**

- Monkey Patching
- Regular Expressions (--)
- Time ve Date Nesneleri (%50)
- Ruby Paketleri: RubyGems
- Paket Yöneticisi: Bundler (--)
- Komut Satırı (Command-Line) Kullanımı (--)
- Meta Programming

## **Kod Yazma Tarzı (*Style Guide*) (%98)**

- Değişken Tanımlamaları
- Blok Tanımlamaları
- Syntax Tanımlamaları
- Semantik Tanımlamalar
- Yanlış ve Doğru Kullanımlar
- İsimlendirmeler

## **Gerçek Hayat Ruby Örnekleri (%25)**

- Neden Ruby?
- Ruby ve TDD/BDD/CI (--)
- Kendi Rubygem'imizi yapalım! (--)
- Sinatra ve Web (--)

# İnceleme

Bu kısma Technical Review gelecek...

# Bölüm 1

Bu bölümde Ruby'nin kısaca tarihçesine bakacağız. Buna ek olarak farklı işletim sistemlerinde Ruby kurulumuna kısaca değineceğiz. Eğer **OSX** ya da **Linux** türevi işletim sistemi kullanıyorsanız işiniz kolay :)

Windows kullanıcıları için özel bir **Installer** var.

Bu bölümde son olarak da **IRB** ve komut satırı üzerinden Ruby çalıştırmayı göreceğiz.

- [Ruby Hakkında](#)
- [Kurulum](#)
- [İnteraktif Kullanım](#)
- [Ruby Komutu ve Parametreleri](#)



# Ruby Hakkında

1990'lı yılların ortalarında (1995) **Yukuhiko "Matz" Matsumoto** tarafından geliştirilen Ruby, günümüzde en çok kullanılan [açık-kaynak](#) yazılımların başında geliyor.

Üretkenlik (az kod, çok iş) ve basitliğe odaklı, dinamik, açık-kaynak programlama dili.  
Okuması ve yazması kolay, anlaşılabilir nitelikte!

Dilin en büyük esin kaynakları tabii ki yine varolan diller. Bunlar; Perl, Smalltalk, Eiffel, Ada ve Lisp dilleri.

İlk kararlı (stable) sürümü 1995'de yayınlanan Ruby'nin geliştiricilerin tam anlamıyla dikkatini çekmesi **2006** yılına kadar sürdü. Keza ilk versiyonları gerçekten çok yavaş ve sıkıntılıydı.

Ruby en büyük patlamasını [Ruby on Rails](#) framework'ü ile yaptı. Danimarkalı yazılımcı [@dhh](#)'in (*David Heinemeier Hansson*) yayınladığı bu framework ne yazık ki Ruby dilinin önüne bile geçti.

Kitabı yazdığım an itibariyle (*13 Temmuz 2014, Pazar*) Ruby'nin en son sürümü [2.1.2](#) (Stabil sürüm)

- Güncelleme: Ruby versiyon [2.1.3](#) oldu. (*26 Ekim 2014, Pazar*)
- Güncelleme: Ruby versiyon [2.1.5](#) oldu. (*6 Aralık 2014, Pazar*)
- Güncelleme: Ruby versiyon [2.2.0](#) oldu. (*25 Aralık 2014*)
- Güncelleme: Ruby versiyon [2.2.1](#) oldu. (*3 Mart 2015*)
- Güncelleme: Ruby versiyon [2.2.2](#) oldu. (*1 Mayıs 2015*)
- Güncelleme: Ruby versiyon [2.3.0](#) oldu. (*24 Aralık 2015*)

Ruby'nin en önemli özelliği her şeyin bir nesne yani **Object** olmasıdır. Nesneyi bir tür paket / kutu gibi düşünebilirsiniz. Doğal olarak, **Object** yani nesne olan bir şeyin, action'ları / method'ları da olur.

Ruby'nin yaratıcısı **Matz** şöyle demiş:

**Perl** dilinden daha güçlü, **Python** dilinden daha object-oriented bir script dili olmasını istedim.

Pek çok programlama dilinde sayılar primitive (ilkel/basit) tiplerdir, nesne değildirler. Halbuki Ruby'de sayılar dahil herşey nesnedir. Yani sayının method'ları vardır :)

[Örnek](#)

```
class Numeric
  def topla(x)
    self.+(x)
  end
end

5.topla(6) # => 11
5.topla(16) # => 21
```

Sayılara (yani *Numeric* tipine) `topla` diye bir method ekledik...

Block ve Mixin ise Ruby'nin yine öne çıkan özelliklerindendir. Block denen şey aslında [closure](#)'dur. Herhangi bir method'a block takılabilir:

```
search_engines = %w[Google Yahoo MSN].map do |engine|
  "http://www." + engine.downcase + ".com"
end

search_engines # => ["http://www.google.com", "http://www.yahoo.com", "http://www.msn.com"]
```

Bu örnekte `%w` string'i boşluklarından ayırarak bir array (dizi) formatına çeviri. Yani sonuçta `%w[Google Yahoo MSN]` dediğimizde elimize `["Google", "Yahoo", "MSN"]` dizisi gelir. `map` metodu bize bu diziden bir `Enumerator` dönecektir ve `do/end` kısmı ise bizim `block` kısmımızdır. Bu konuda daha açıklayıcı bilgiyi 3. bölümdeki Bloklar başlığı altında bulacaksınız.

Ruby'de bir **Class** (sınıf) sadece tek bir sınıftan türeyebilir. Yani A class'ı B'den türer ama aynı anda hem B'den hem C'den türeyemez. Bu Python'da mümkün olan bir şeydir. Ruby'de ise bunun üstesinden gelmek için class'lar **Module**'leri kullanır. Bir Class N tane Module içerebilir, işte bu tür nesnelere **Mixin** denir:

```
class MyArray
  include Enumerable
end
```

Ortak kullanılacak metodları ya da değişkenleri ayrı bir Module olarak tasarlayıp, gerektiği yerde `include` ederek Class + Module karışımından oluşan Mixin'ler ortaya çıkar.

Diğer dillerdeki gibi Exception Handling, Garbage Collector özelliklerinin yanı sıra, C-Extension'ı yazmak diğer dillere göre daha kolaydır. İşletim sisteminden bağımsız **threading** imkanı sunmaktadır. Pek çok işletim sisteminde Ruby kullanmak mümkündür: Linux / Unix / Mac OS X / Windows / DOS / BeOS / OS/2 gibi...

Ruby'den türemiş farklı Ruby uygulamaları da var:

[JRuby](#), [Rubinius](#), [MacRuby](#), [mruby](#), [IronRuby](#), [MagLev](#), [Cardinal](#)

Son olarak, **Test Driven Development** yani test'le yürüyen geliştirme mentalitesinin en iyi oturduğunu düşündüğüm dillerden biri Ruby'dir. Çok güzel test kütüphaneleri var ve nasıl kullanabileceğimize dair tonlarca blog/video sitesi de mevcut!

| Ruby ile programlamak eğlencelidir!

diyor Matz, haydi o halde biz de bu eğlenceye katılalım :)

# Kurulum

## OSX

Eğer Mac OSX kullanıyorsanız ilk etapta hiçbir şeye ihtiyacınız yok, çünkü Mac OSX'de Ruby hazır kurulu olarak geliyor.

OSX Mavericks (10.9.4) Ruby sürümü : `ruby 2.1.0p0 (2013-12-25 revision 44422) [x86_64-darwin13.0]`

## Linux

[Debian](#) ve [Ubuntu](#) kullanan okuyucularımız

```
sudo apt-get install ruby # ya da
sudo aptitude install ruby
```

[CentOS](#), [Fedora](#), ya da [RedHat](#) kullananlar:

```
sudo yum install ruby
```

[Gentoo](#) kullananlar;

```
sudo emerge dev-lang/ruby
```

## Kaynaktan Kurulum

Ruby'nin sitesinden [tar dosyasını](#) indirip;

```
./configure
make
sudo make install
```

şeklinde de kurulum yapabilirsiniz.

## Windows

Bu siteden özel Windows için hazırlanmış Ruby kurulum paketini indirip klasik "next" > "next" diyerek kurulum yapabilirsiniz.

## Ruby Versiyon Yöneticileri

Bazen, kullandığınız hazır kütüphanelerin destekledikleri Ruby versiyonlarındaki kısıtlamalar ya da kişisel tercihiniz gibi, farklı nedenlerle birden fazla Ruby sürümü ile çalışmak isteyebilirsiniz. Projelerinizden biri, örneğin `ruby 1.9.3` kullanırken, diğer bir projeniz `ruby 2.1.0` kullanıyor olabilir. Bu anlarda kullandığınız Ruby versiyonunu kolayca değiştirmek, aslında aktive etmek de diyebiliriz, için 2 adet popüler versiyon yöneticisi bulunmaktadır.

### Rbenv

Rbenv meşhur [37 Signals](#)'ın. Aslında orada çalışan [Sam Stephenson](#) tarafından geliştirilmiş bir araç.

Eğer OSX ve [Homebrew](#) kullanıyorsanız kurulum çok kolay:

```
brew install rbenv ruby-build
```

Eğer farklı bir işletim sistemi kullanıyorsanız (Linux/Unix tabanlı)

```
git clone https://github.com/sstephenson/rbenv.git ~/.rbenv

# sonra PATH'e ekleyin
export PATH="$HOME/.rbenv/bin:$PATH"

# açılışa burada ekleyin
# hangisini kullanıyorsanız (.bashrc, .profile ya da .bash_profile)
eval "$(rbenv init -)"
```

Kurulumdan sonra istediğini Ruby versiyonu için;

```
# kurulabilecek versiyonları göster
rbenv install -l

# ruby 2.3.0'ı kuralım
rbenv install 2.3.0
```

Kurulan Ruby'yi

- Sistem genelinde `rbenv global`
- Sadece bulunduğumuz dizin içinde (Uygulamaya Özel) `rbenv local`
- Anlık, sadece Shell'de `rbenv shell`

aktive etme opsiyonlarımız var. Örneğin proje dizinin içine `.ruby-version` dosyası koyar ve içine de hangi versiyonu kullandığımızı yazarsak o dizine geçtiğimiz an Ruby versiyonu değişir.

Yani **A** projesinde versiyon `2.1.1` , **B** projesinde version `1.9.3` kullanmak için;

```
cd ~/projelerim/A/  
echo "1.9.3" > .ruby-version  
  
cd ~/projelerim/B/  
echo "2.1.1" > .ruby-version  
  
# bakalım hangi versiyonu aktive etmişiz?  
rbenv version
```

## RVM

Adından da anlaşılacağı gibi **Ruby Version Manager** yani **RVM** de aynı Rbenv gibi Ruby versiyonlarını kolay yönetmeyi sağlıyor. Ruby dünyasından Rbenv'ciler ve RVM'ciler olarak iki kanat olduğunu söyleyebilirim.

Kurulumu da zor değil:

```
gpg --keyserver hkp://keys.gnupg.net --recv-keys D39DC0E3  
\curl -sSL https://get.rvm.io | bash -s stable
```

Rbenv'den en büyük farklığı **Gem Set** yani proje bazlı Ruby paketi yönetimi özelliği.

Ben Rbenv'ci olduğum için RVM kullanmıyorum. Özellikle yeni başlayanlar için RVM'i öneriyorum, Rbenv'e göre daha kolay kurulumu ve kullanımı var.

# İnteraktif Kullanım

Ruby'nin (ve benzer dillerin) en çok hoşuma giden özelliği İnteraktif Shell özelliği olmasıdır. Aynı eskiden Commodore 64 günlerindeki gibi, shell'i açıp Ruby yazmaya başlayabiliriz.

Genel olarak bu interaktif kullanım **REPL** olarak geçer. REPL aslında **R**ead **E**valuate **P**rint **L**oop'un baş harfleridir. Yani, kullanıcıdan bir input (*Read*) gelir, bu girdi çalıştırılır (*Evaluate*), sonuç ekrana yazdırılır (*Print*) ve son olarak başa döner ve yine input bekler (*Loop*).

REPL olayı, Python ve PHP'de de var.

## IRB

Ruby kurulumu ile beraber gelir. Yapmanız gereken Terminal'i açıp `irb` yazıp `enter` a basmak.

```
irb
irb(main):001:0> print "Merhaba Dünya"
Merhaba Dünya=> nil
irb(main):002:0>
```

Örnekleri yaparken çok sık kullanacağız bu komutu. Keza, daha da geliştirilmiş bir versiyon olan `pry` gem'ini de göreceğiz ilerleyen bölümlerde.

## Shell

**Shebang** dediğimiz yöntemli Linux/Unix tabanlı işletim sistemlerinde Ruby dosyalarını aynı bir uygulama çalıştırır gibi kullanabilirsiniz.

`test.rb` dosyası olduğunu düşünün; bu dosya

```
#!/usr/bin/env ruby
puts "Merhaba dünya"
```

şeklinde olsun. Bu dosyayı çalıştırmak için ya

```
ruby test.rb
```

ya da, dosyanın Execute flag'ini aktif hale getirerek

```
chmod +x test.rb  
./test.rb
```

çalıştırabilirsiniz. Eğer Execute flag'ini aktif hale getirmez iseniz işletim sistemi size aşağıdaki gibi bir hata dönecektir.

```
permission denied: ./test.rb
```

Bunun sebebi dosyanızın Execute edilebilmesi için izninin bulunmamasıdır. Konuya daha yakından bakmak için `test.rb` dosyasının bulunduğu dizinde `ls -l` komutunu çalıştıralım.

```
$ ls -l  
total 8  
-rw-r--r-- 1 kullanıcı staff 42 May 20 23:48 test.rb
```

Burada görüleceği gibi `-rw-r--r--` dosyanızın sadece okuma ve aktif kullanıcı için yazma izni bulunmakta. Eğer yukarıdaki gibi Execute flag'ini aktif hale getirirseniz dosyanızın son hali aşağıdaki gibi olacaktır.

```
$ ls -l  
total 8  
-rwxr-xr-x 1 kullanıcı staff 42 May 20 23:48 test.rb
```

Şu anda dosyanız tüm kullanıcılarda okunabilir ve çalıştırılabilir durumda.



# Ruby Komutu ve Parametreleri

Kurulum işlemleri bittikten sonra ya da kullandığınız **OS** (*İşletim Sistemi*) ön tanımlı Ruby ile geliyorsa hemen aşağıdaki testi yapabilirsiniz:

```
ruby --help
```

```
Usage: ruby [switches] [--] [programfile] [arguments]
```

ruby 'yi çağırırken, her shell aracı gibi (*binary mi diyim, executable mı diyim karar veremedim!*) Ruby de çeşitli parametreler alabiliyor. Bu kısma dikkat edelim çünkü burada bahsi geçecek **Switch**'ler 2.Bölümde göreceğimiz **Ön Tanımlı Değişkenler** ile çok alakalı.

Switch	Açıklaması
-O[octal]	<code>\$/</code> değeridir. ( <i>Ön tanımlı değişkenlerde göreceğiz</i> ) <b>octal</b> yani 8'lik sistemde değer atanır. Örneğin <code>ruby -0777</code> şeklinde çalıştırılsa, Ruby, dosya okuma işlemleri sırasında tek seferde dosyayı okur ve tek <b>String</b> haline getirir.
-a	<code>-n</code> ve <code>-p</code> ile birlikte kullanılınc <b>autosplit mode</b> olarak çalışır. Yani <code>\$F =&gt; \$_.split</code> şeklinde işler.
-C	<b>Syntax Check</b> yani dosya içindeki kodu çalıştırmadan, sadece söz dizimi kontrolü yapar ve çıkar.
-Cdirectory	Ruby önce belirtilen <b>directory</b> 'ye <code>cd</code> ( <i>Shell'de bir dizine geçiş yapmak</i> ) yapar ve daha sonra kodu çalıştırır. <code>ruby -C/tmp/foo</code> gibi..
-d, --debug	Debug modda çalıştırır. Bu esnada <code>\$DEBUG</code> değişkeni de <code>true</code> değerini alır. Yani eğer kodunuzun içinde <code>if \$DEBUG</code> gibi bir ifade kullanabilir ve sonucunu görebilirsiniz.
-e 'command'	Komut satırından tek satırda Ruby kodu çalıştırmak için. <code>ruby -e 'puts "hello"'</code> gibi.
-Eex[:in], --encoding=ex[:in]	Varsayılan karakter encoding'i ( <i>Internal ve External için...</i> )
--external-encoding=encoding	<code>-E</code> gibi
--internal-encoding=encoding	<code>-E</code> gibi
-Fpattern	<b>auto split</b> için ve <code>split()</code> için varsayılan regex pattern'i. <code>\$;</code> değeri.
-i	<b>in-place-edit</b> mod. Lütfen örneğe bakın!
-I	<code>\$LOAD_PATH</code> 'e ilave path ekleme.
	Japonca ( <i>KANJI</i> ) encoding belirtilir. <code>UTF-8</code> için <code>-K u</code>

	kullanılabilir.
-l	Otomatik satır sonu ( <i>Line Ending</i> ) işlemi. -n ve -p ile çalışır. Önce \$\ değişkenine \$/ değeri atanır, chop! method'u her satıra uygulanır.
-n	Komut satırındaki sed -n ya da awk gibi çalışır. Sanki kodun etrafında loop varmış gibi davranarak süzgeçten geçirir.
-p	-n gibi çalışır, farkı \$_ den gelen değeri döngünün sonunda print eder.
-r	require komutunun yaptığı gibi verilen değeri require eder. (require komutunu ileride göreceğiz)
-s	Komut satırı argümanlarını parse etme (işleme) özelliğini aktive eder.
-S	\$PATH çevre değişkenini bulmayı force eder. Örneğe bakınız!
-T	Güvenlik seviyesi, tainted kontrolünü devreye sokmak. \$SAFE değişkenine -T ile geçilen değer atanır.
-v, --verbose	Önce versiyon numarasını yazar sonra da verbose (Ayrıntılı çalıştırma) modu aktive eder. Yani \$VERBOSE true olur.
-w	-v ile aynı işi yapar sadece versiyon numarasını yazmaz.
-W	Uyarı seviyesini belirler (Warning Level). 0 Sessiz, 1 Orta şekerli, 2 Verbose!
-x	Shebang'den önceki yazıyı siler atar ve alternatif olarak ilgili dizine cd yapar.
--copyright	Ruby'e ait telif bilgisini yazar. ruby - Copyright (C) 1993-2013 Yukihiro Matsumoto
--enable=feature[,...], --disable=feature[,...]	Örneğin kodun RubyGem'lerini kullanmasını istemiyorsanız --disable-gems şeklinde, ya da \$RUBYOPT çevre değişkenini devre dışı bırakmak için --disable-rubyopt gibi. --disable-all her ek özelliği devre dışı bırakır. --enable-all ya da devreye sokar.
--version	Versiyon numarasını yazar.
--help	Yardım sayfasını gösterir.

ruby --help dışında daha detaylı bilgi man ruby yani man pages'da bulmak mümkündür, ben de pek çok şeye oradan baktım.

## -i örneği:

Önce içinde düz metin olan bir dosya oluşturalım:

```
echo vigo > /tmp/test.txt
cat /tmp/test.txt
# vigo
```

sonra;

```
ruby -p -i.backup -e '$_.upcase!' /tmp/test.txt  
cat /tmp/test.txt.backup  
# VIGO
```

Ne oldu? Amacımız, `/tmp/test.txt` dosyasında, satır satır okuyup her satırda yazan metni **uppercase** yani büyük harfe çevirmek. Normalde bu işlemi;

```
ruby -p -e '$_.upcase!' /tmp/test.txt
```

şeklinde komut satırından yapabiliyoruz. Ama **in-place-edit** mod ve **extension** özelliği ile, çalıştırılmış kod çıktısını başka bir dosyada görüntüleyebiliriz. Bu noktada `-i` devreye giriyor. `-i.backup` sonucun görüntülendiği dosya oluyor.

## -n örneği:

Loop'tan kastım, sanki;

```
while gets  
  # kod...  
end
```

çevreler.

## -s örneği:

`example_s.rb` adında bir dosyamız olsun ve içinde;

```
print "xyz argümanı kullanıldı\n" if $xyz
```

yazsın. Bu dosyayı çalıştırın;

```
ruby example_s.rb
```

Hiçbir çıktı görmezsiniz. Eğer şu şekilde çalıştırırsanız;

```
ruby -s example_s.rb -xyz
```

şu çıktıyı alırsınız:

```
xyz argümanı kullanıldı
```

### -s örneği:

Bazı işletim sistemlerin **Shebang** sorunu olabilir. `#!/usr/bin/env ruby` Bu gibi durumlarda;

```
#!/bin/sh  
exec ruby -S -x $0 "$@"
```

şekinde, `bash` üzerinden Ruby script'i'ni çalıştırabiliriz.

wip...

## Bölüm 2

Bu bölümde;

- [Syntax](#) (*Söz Dizimi*)
- [Comments](#) (*Yorum satırı*)
- [Rezerve Edilmiş Kelimeler](#)
- [Değişken Tanımlama ve Türleri](#)
- [Duck Typing](#)
- [Ön Tanımlı Değişkenler](#)
- [Pseudo Değişkenler](#)
- [Operatörler](#)
- [Global Constants](#) (*Genel Sabitler*)

konularını işleyeceğiz.

# Syntax (Söz Dizimi) ve Rezerve Kelimeler

## Syntax (Söz Dizimi)

Genel olarak çok kolay ve anlaşılır bir **syntax**'a sahiptir. Sanki İngilizce okur/konuşur gibi söz dizimi bulunur. Örneğin,

Eğer a'nın değeri 5'ten büyükse ekrana "Merhaba" yaz

demek istiyorsak;

```
puts "Merhaba" if a > 5
```

şeklinde yazabiliriz.

Diğer dillerden farklı olarak, Ruby'de fonksiyon (method) çağırırken **parantez** kullanmak zorunluğu yoktur. Bu ilk etapta kafa karıştırıcı gibi dursa da, alışınca ne kadar kolay okunabilir olduğunu görüyorsunuz. Mecburi değil, yani parantez kullanmanızda sorun yok. Parantezli kullanım;

```
def greet_user(user_name)
  puts "Merhaba #{user_name}"
end

greet_user("Uğur") # Merhaba Uğur
```

Eğer parantez kullanmazsak;

```
def greet_user user_name
  puts "Merhaba #{user_name}"
end

greet_user "Uğur" # Merhaba Uğur
```

şeklinde olur. Keza, pek çok dilde, fonksiyon eğer bir şey dönerse geriye, mutlaka `return` komutu kullanılır. Ruby'de buna da gerek yok. Çünkü her method (*yani Fonksiyon*) mutlaka default olarak bir şey döner. Hiçbir şey dönmese bile `nil` döner. Bu bakımdan da;

```
def greet_user user_name
  "Merhaba #{user_name}"
end

puts greet_user "Uğur" # Merhaba Uğur
```

şeklinde kullanabiliriz. Korkmayın, kafalar karışmasın. Detaylara ileride gireceğiz.

## Comments (Yorum Satırları)

Her dilde olduğu gibi **Comment out** yani "işaretli kısmı çalıştırma" demek için kullandığımız şey Ruby'de de var.

**Comment** için `#` işareti kullanılıyor. Genelde `line-comment` yani satır bazlı, ve `block-comment` yani kod bloğu bazlı yorum yapma şekilleri var.

```
# Bu satır line-comment, yani tek satırlı yorum

# Bu kısım block-comment
#
# def test_user
#   true
# end

# ya da

=begin
Bu yorum...
Bu da yorum...
Hatta bu da...
=end
```

Gördüğünüz gibi `block-comment` için ilave olarak `=begin` ve `=end` kelimeleri kullanılabiliyor.

## Rezerve Edilmiş Kelimeler

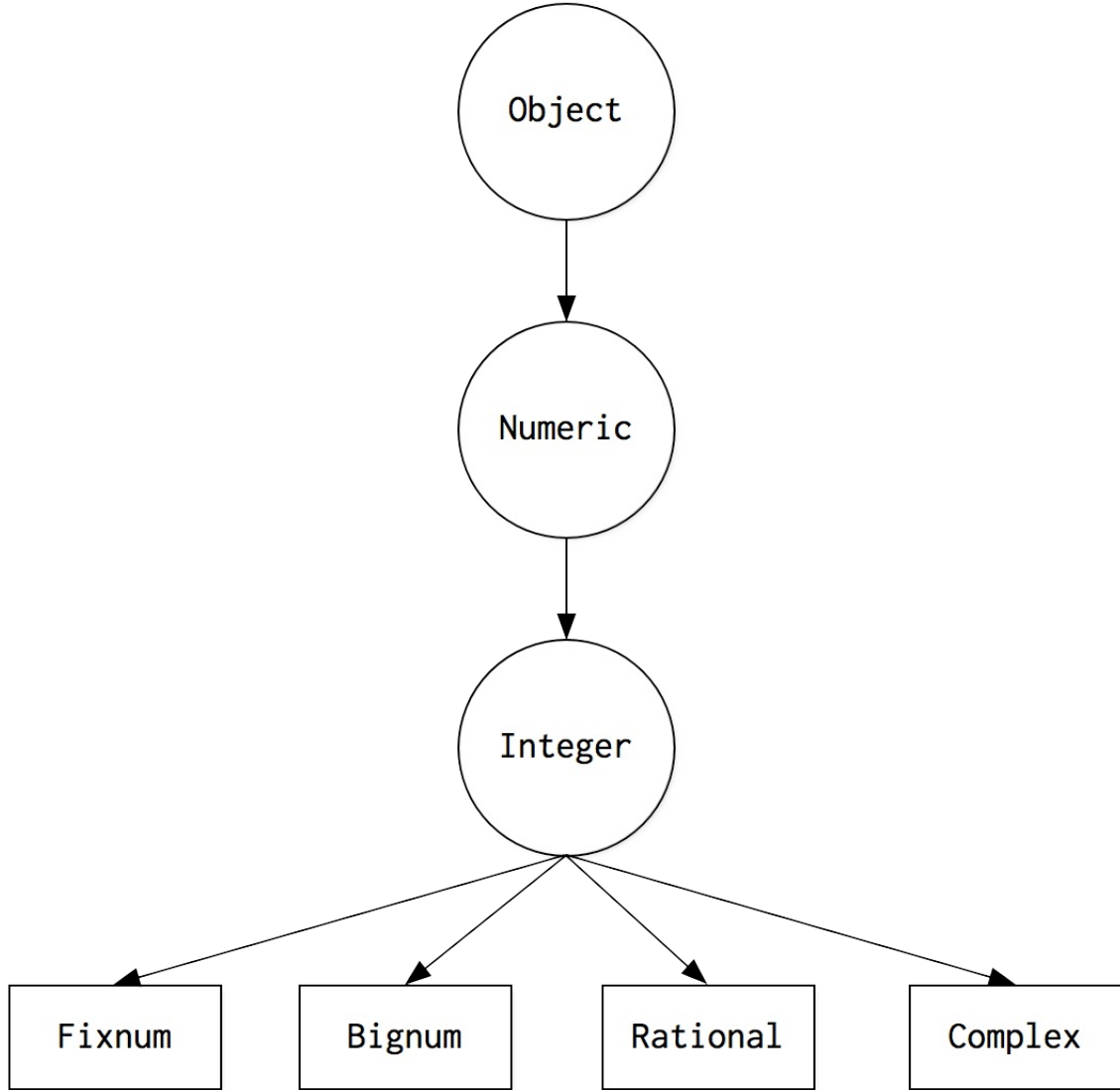
Her dilde olduğu gibi Ruby'de de daha önceden rezerve edilmiş bazı kelimeler bulunmaktadır. Bu kelimeleri değişken ya da method adı olarak kullanamıyoruz. Bu kelimeler Ruby komutları ve özel durumlar için rezerve edilmiş.

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



# Değişkenler

Değişken denen şey, yani **Variable**, nesneye atanmış bir işaretçidir aslında. Ne demiştik? Ruby'de herşey bir nesne yani **Object**. Bu nesnelere erişmek için aracıdır değişkenler.



Farklı farklı türleri vardır. Birazdan bunlara değineceğiz. En basit anlamda değişken tanımlamak;

```
a = 5
user_email = "example@foo.com"
```

şeklinde olur. Yukarıdaki örnekte `a` ve `user_email` değişkenin adıdır. Değeri ise eşittir işaretinden sonra gelendir.

Yani yukarıda; `a` ya sayı olarak `5` ve `user_email` e metin olarak `example@foo.com` değerleri atanmıştır.

Ruby **Duck Typing** şeklinde çalışır. Yani atama yapmadan önce ne tür bir değer ataması yapacağımızı belirtmemize gerek yok. Ruby zaten `a = 5` dediğimizde, "Hmmm, bu değer Fixnum türünde" diye değerlendirir.

**Duck Typing** demek şudur; Eğer ördek gibi yürüyorsa, ördek gibi ses çıkartıyorsa e o zaman bu bir Ördektir! İngilizcesi;

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck

Yani, bir kuş, eğer ördek gibi yürüyorsa, ördek gibi yüzüyorsa ve ördek gibi ses çıkartıyorsa ben buna Ördek derim!

## Metinsel Atamalar ve Tırnak Kullanımı

Yeri gelmişken hızlıca bir konuya değinmek istiyorum. Metinsel değişkenler tanımlarken (**String**) eşitlik esnasında tek ya da çift tırnak işareti kullanabiliriz. Fakat aradaki farkı bilerek kullanmamız gerekir.

String içinde değişken kullanımı yaptığımız zaman yani;

```
a = 41
puts "Siz tam #{a} yaşındasınız"
```

gibi bir durumda, gördüğünüz gibi `#{a}` şeklinde yazı içinde değişken kullandık. Bu kodun çıktısı aşağıdaki gibi olacak.

```
Siz tam 41 yaşındasınız
```

Format olarak Ruby'de, `#{BU KISIMDA KOD ÇALIŞIR}` şeklinde istediğimiz kodu çalıştırma yetkimiz var. Bu işlem sadece **çift tırnak** kullanımında geçerlidir. Başka bir örnek vermek gerekirse;

```
a = 41
puts "Siz tam #{a+2} yaşındasınız"
```

Yukarıdaki örnekte Ruby `a+2` komutunu çalıştıracaktır ve sonuç olarak `43` değerini bulacaktır ve bunu ekrana yazdıracak. Yani sonuç:

```
Siz tam 43 yaşındasınız
```

şeklinde olacaktır. Ancak aynı kodu tek tırnak kullanarak yapmış olsaydık;

```
a = 41
puts 'Siz tam #{a} yaşındasınız'
```

çıktısı:

```
Siz tam #{a} yaşındasınız
```

olacaktı. **Tek tırnak** içinde bu işlem çalışmaz!

## Local (Bölgesel)

Bölgesel ya da **Yerel** değişkenler, bir **scope** içindeki değişkenlerdir. Scope nedir? Kodun çalıştığı bölge olarak tanımlayabiliriz. Bu tür değişkenler mutlaka küçük harfle ya da `_` (*underscore*) işareti ile başlamalıdır. Kesinlikle `@`, `@@` ya da `$` işareti gibi ön ekler alamazlar.

```
out_text = "vigo"
def greet_user(user_name)
  out_text = "Merhaba #{user_name}"
end

puts greet_user("vigo") # Merhaba vigo
puts out_text          # vigo
```

Program çalıştığında `out_text` değişkeninin değeri `vigo` olarak atanmaktadır. Daha sonra 6. satırda `greet_user` method'u (*fonksiyonu*) çalıştığında, o methodun içerisinde `out_text` değişkeninin değeri değiştiriliyor gibi görünüyor. Daha sonra 7. satırda `out_text` değişkeninin değeri `puts` methodu ile ekrana yazdırılmaktadır. Ancak burada çıktıları baktığınızda methodun içerisindeki `out_text` değişkenindeki değişim, programın en başında tanımladığımız `out_text` değişkenin değerini etkilememiştir. Method içerisinde kalmıştır. Burada method içerisindeki `out_text` değişkeni aslında `local variable` yani yerel değişken şeklinde çalışmaktadır.

## Global (Genel)

`$` işaretiyle başlayan tüm değişkenler **Global** değişkenlerdir. Kodun herhangi bir yerinde kullanılabilir ve erişilebilir.

```
$today = "Pazartesi"
def greet_user(user_name)
  out_text = "Merhaba #{user_name}, bugün #{today}"
  puts out_text
end

puts "Bugün günlerden ne? #{today}"
greet_user("vigo") # Merhaba vigo, bugün Pazartesi
```

Bu örnekteki **Global** değişken `$today` değişkenidir.

## Constants (Sabitler)

Sabit nedir? Değiştirelemeyendir. Yani bu tür değişkenler, ki bu değişken değildir :), **sabit** olarak adlandırılır. Kural olarak mutlaka **BÜYÜK HARF**'le başlar! Bazen de tamamen büyük harflerden oluşur.

```
My_Age = 18
your_age = 22

puts defined?(My_Age) # constant
puts defined?(your_age) # local-variable
```

`My_Age` sabit, `your_age` de yerel değişken...

Ruby'de ilginç bir durum daha var. Constant'lar **mutable** yani değiştirilebilir. Nasıl yani?

```
My_Age = 18

puts defined?(My_Age) # constant
puts "My_Age: #{My_Age}" # My_Age: 18

My_Age = 22

puts defined?(My_Age) # constant
puts "My_Age: #{My_Age}" # My_Age: 22
```

ama `warning` yani **uyarı mesajı** aldık!

```
untitled:6: warning: already initialized constant My_Age
untitled:1: warning: previous definition of My_Age was here
```

`My_Age` sabiti 6.satırda zaten tanımlıydı. Önceki değeri de 1.satırda diye bize ikaz etti Ruby yorumlayıcısı.

# Paralel Atama

Hemen ne demek istediğimi bir örnekle açayım:

```
x, y, z = 5, 11, 88
puts x # 5
puts y # 11
puts z # 88

a, b, c = "Uğur", 5.81, 1972
puts a # Uğur
puts b # 5.81
puts c # 1972
```

`x, y, z = 5, 11, 88` derken tek harekette `x = 5` , `y = 11` ve `z = 88` yaptık. İşte bu paralel atama.

## Instance Variable

**Instance** dediğimiz şey **Class**'dan türemiş olan nesnedir. Bu konuyu detaylı olarak ileride inceleyeceğiz. Sadece ön bilgi olması adına değiniyorum. `@` işareti ile başlarlar.

```
class User
  attr_accessor :name
  def initialize(name)
    @name = name
  end

  def greet
    "Merhaba #{@name}"
  end
end

u = User.new("Uğur")
puts u.greet      # Merhaba Uğur
puts u.name       # Uğur
```

`u.name` diye çağırdığımız şey `User` class'ından türemiş olan `u` nesnesinin **Instance Variable**'ı yani türemiş nesneye ait değişkenidir. Fazla takılmayın, `Class` konusunda bol bol değineceğiz...

## Class Variable

**Class**'a ait değişkendir. Dikkat edin burada türeyen birşey yok. `@@` ile başlar. Kullanmadan önce bu değişkeni mutlaka `init` etmelisiniz (*Yani ön tanımlama yapmalısınız*)

```
class User
  attr_accessor :name
  @@instance_count = 0 # Kullanmadan önce init ettim
  def initialize(name)
    @name = name
    @@instance_count += 1 # Class'dan her instance oluşmasında sayacı 1 arttırıyorum
  end

  def greet
    "Merhaba #{@name}"
  end

  def self.instance_count # burası öneli
    @@instance_count
  end
end

user1 = User.new("Uğur")
user2 = User.new("Ezel")
user3 = User.new("Yeşim")

puts "Kaç defa User instance'ı oldu? #{User.instance_count}" # Kaç defa User instance'ı oldu? 3
```

Eğer kafanız karıştıysa sorun değil, **Class** konusunda hepsinin üzerinden tekrar geçeceğiz.

# Ön Tanımlı Değişkenler

Ruby, bir kısım ön tanımlı yani **Predefined** değişkenlerle geliyor. Değişkenlerin ne olduğunu;

```
puts global_variables
```

şeklinde görebiliriz. Önden bu bilgileri vermek zorundayım, daha geniş kullanımı ve tam olarak ne işe yaradıklarını ileriki bölümlerde daha iyi anlayacaksınız.

Değişken	Açıklama
\$!	<code>raise</code> ile atanan <b>exception</b> bilgisidir. <code>rescue</code> ile erişilir.
\$@	Son <b>exception</b> 'a ait <b>backtrace</b> ( <i>bir tür log</i> ) bilgilerinin tutulduğu dizi ( <i>Array</i> )
\$&	Son yakalanan <b>match</b> 'in tutulduğu <b>String</b> ( <i>Regex konusunda göreceğiz</i> )
\$`	Son yakalanan <b>match</b> 'in solunda kalan kısım
\$'	Son yakalanan <b>match</b> 'in sağında kalan kısım
\$+	En yüksek <b>group match</b> 'in tutulduğu yer. ( <i>Regex yaparken group yakalama konusunda göreceğiz</i> )
\$1, \$2, ..., \$9	Yine, Regex ile patern yakalama ( <i>pattern matching</i> ) yaptığımızda, yakaladığımız şeylerin sıra numarası.
\$~	O anki kapsama alanında ( <i>scope</i> ) son yakalananla ilgili bilgilerin tutulduğu değişken
\$=	Regex ile uğraşırken, karakterlerin büyük/küçük harfe duyarlılığı ile ilgi ayarlar vardır. Örneğin büyük/küçük harf farkı olmadan aramak yaparken ( <i>case insensitive</i> ) bu değişkene atama yaparız. Varsayılan değer ( <i>default</i> ) <code>nil</code> 'dir
\$/	Dosyadan okuma yapılırken satırların nasıl ayrıldığıнын tespit edildiği değişkendir. Eğer <code>nil</code> olarak atarsa, dosya okuması esnasında satır-satır okuma yerine tüm dosya bir anda okunur.
\$\	Bu da çıktı için ayrıdır. <code>print</code> ve <code>puts</code> gibi komutlarda <code>IO#write</code> gibi işlemlerde kullanılır. Varsayılan değer <code>nil</code> 'dir
\$_	<code>print</code> ve <code>Array#join</code> de kullanılan ayrıdır.
\$;	<code>String#split</code> de kullanılan ayrıdır.
\$.	Dosya işlemlerinde son okunan dosyanın aktif satır numarasını verir.
\$<	Aynı <code>shell</code> deki ekleme ( <i>concatenation</i> ) işlemi gibi sanal ekleme yapar.

\$>	<code>print</code> ve <code>printf</code> işlemi için varsayılan çıktıdır. Varsayılan değeri de <code>\$stdout</code>
\$_	<code>gets</code> veya <code>readline</code> ile alınan son satırdır, cinsi <b>String</b> 'dir.
\$0	Çalıştırılan script'in dosya adıdır.
\$*	Komut satırı işlemlerinde, dosyaya geçilen argümanların saklandığı değişkendir.
\$\$	Çalıştırılan script'in işlem numarası ( <i>Process Number</i> )
\$?	Çalıştırılan son alt işlemin ( <i>Child Process</i> ) durumu.
\$:	Modüller ve ek dosyalar için <b>Path</b> ( <i>Load Path_</i> ) bilgisi. <code>require</code> komutunda göreceğiz.
\$"	<code>require</code> ile yüklenen dosyaların adlarının tutulduğu dizi ( <i>Array</i> )
\$DEBUG	Adından da anlaşıldığı gibi, eğer <b>DEBUG</b> modda çalıştırma yapıyorsak ( <i>ki bunu -d ile yaparız</i> ) oluşan her <b>exception</b> 'in <code>\$stderr</code> değişkenine atanmasını sağlar.
\$KCODE	Kod yazdığımız script dosyasının <b>encoding</b> tipini seçmemize yarar. Son sürümlerde ihtiyaç kalmadı, herşey <b>default</b> olarak <code>UTF-8</code> çalışıyor.
\$FILENAME	Komut satırından argüman olarak dosya geçtiğimizde geçilen dosyanın adını almak için kullanılır. Aslında <code>ARGF.filename</code> ile aynı işi yapar.
\$LOAD_PATH	<code>\$:</code> ile aynı işi yapan <b>alias</b> 'dır ( <i>alias = takma ad</i> )
\$SAFE	Güvenlik seviyesidir. Varsayılan değer <code>0</code> dır. Bu dereceler <code>0</code> 'dan <code>4</code> 'e kadardır. Kod güvenliği ve kilitleme yapmak için kullanılır. Biraz karmaşık bir konudur :) Örneğin, emin olmadığınız bir kütüphane kullanırken kodunuzu güvenli hale getirmek için, kod bloğunun önüne <code>\$SAFE=4</code> ekerseniz, takip eden kod <code>array</code> <code>hash</code> ve <code>string</code> lerde hiç bir modifikasyon yapamaz! Hatta pek <b>çok şeyi</b> yapamaz!
\$stdin	Standart giriş.
\$stdout	Standart çıkış.
\$stderr	Giriş/Çıkış hata bildirimi.
\$VERBOSE	Kernel tarafından üretilen tüm uyarı mesajlarının ( <i>warning gibi...</i> ) görüntülenmesi için kullanılır.
\$-0	<code>\$/</code> ile aynı işi yapar.
\$-a	Komut satırından çalıştırma yaparken <code>-a</code> ataması yapılmışsa <code>\$-a</code> <b>true</b> döner.
\$-d	<code>\$DEBUG</code> ile aynı işi yapar.
\$-F	<code>\$;</code> ile aynı işi yapar.
\$-i	Bu değişken <b>in-place-edit</b> modda <b>extension</b> 'i saklar.
\$-l	<code>\$:</code> ile aynı işi yapar. ( <i>Büyük i</i> )



\$-l	Eğer <code>-lis</code> set edilmişse <code>true</code> döner. <b>Read Only</b> yani sadece okunur, değeri değiştirilemez! ( <i>Küçük l</i> )
\$-p	Eğer <code>-pis</code> set edilmişse <code>true</code> döner. <b>Read Only</b> yani sadece okunur, değeri değiştirilemez!
\$-K	<code>\$KCODE</code> ile aynı işi yapar.
\$-v	<code>\$VERBOSE</code> ile aynı işi yapar.
\$-w	Eğer <code>-w</code> set edilmişse <code>true</code> döner.
\$-W	<b>Warning Level</b> yani oluşabilecek hatalar vs ile ilgili 0, 1 ya da 2.seviyede uyarı mesajları göstermek için.
\$LOADED_FEATURES	<code>\$"</code> ile aynı işi yapar.
\$PROGRAM_NAME	<code>\$0</code> ile aynı işi yapar.

## Pseudo (Gerçek Olmayan) Değişkenler

**Değişken** (*Variable*) gibi görünen ama **Sabit** (*Constant*) gibi davranan ve kesinlikle **değer ataması yapılamayan** şeylerdir.

Değişken	Açıklama
<code>self</code>	Alıcı nesnenin o anki aktif method'u. Yani bu bir <b>Class</b> ise kendisi...
<code>nil</code>	Tanımı olmayan ( <i>Undefined</i> ) şeylerin değeri.
<code>true</code>	Mantıksal ( <i>Boolean</i> ) işlem, anlayacağınız gibi <b>true</b> yani <b>1</b>
<code>false</code>	<b>true</b> 'nun tersi ( <i>Boolean</i> ) yani <b>0</b>
<code>__FILE__</code>	Çalışan kaynak kod dosyasının adı
<code>__LINE__</code>	Çalışan kaynak kod dosyasındaki, o anki aktif satırın numarası

# Operatörler

Operatörler çeşitli kontrolleri yapmak için kullanılır. Hatta bazıları aynı zamanda **method** olarak çalışır. Bazı operatörlerin birden farklı işlemi vardır. Örneğin `+` hem matematik işlemi olan **toplama** için hem de **pozitif** değer kontrolü için kullanılabilir.

Operatör	Açıklama	Method mu?
::	İki tane iki nokta. <b>Scope resolution</b> anlamındadır. Class ve Modül konusunda detayları göreceğiz.	
[]	Referans, set	Evet
[]=	Referans, set	Evet
**	Üssü, kuvveti	Evet
+	Pozitif	Evet
-	Negatif	Evet
!	Mantıksal uzlaşma	
~	Tamamlayıcı	Evet
*	Çarpma	Evet
/	Bölme	Evet
%	Modülo ( <i>Kalan</i> )	Evet
+	Ekleme	Evet
-	Çıkartma	Evet
<<	Sola kaydır	Evet
>>	Sağa kaydır	Evet
&	<b>Bit</b> seviyesinde <b>and</b> (Ve) işlemi	Evet
	<b>Bit</b> seviyesinde <b>or</b> (Veya) işlemi	Evet
^	<b>Bit</b> seviyesinde <b>exclusive or</b> (Veya'nın tersi gibi) işlemi	Evet
>	Büyüktür	Evet
>=	Büyük ve eşit	Evet
<	Küçüktür	Evet
<=	Küçük ve eşit	Evet
<=>	Eşitlik karşılaştırma operatörü ( <i>Spaceship</i> yani uzay gemisi )	Evet
==	Eşitlik	Evet
===	Denklik	Evet

!=	Eşit değil	
!~	Yakalanmayan ( <i>not match</i> )	
=~	Yakalanan ( <i>match</i> )	Evet
&&	Mantıksal ve ( <i>and</i> )	
	Mantıksal veya ( <i>or</i> )	
..	Range'i kapsayan	Evet
...	Range'i kapsamayan	
? :	<b>Ternary</b>	
=	Atama	
+=	Arttırma ve atama	
-=	Eksiltme ve atama	
*=	Çarpma ve atama	
/=	Bölme ve atama	
%=	Modülo ve atama	
**=	Üssü ve atama	
<<=	Bit seviyesinde sola kaydırma ve atama	
>>=	Bit seviyesinde sağa kaydırma ve atama	
&=	Bit seviyesinde <b>and</b> ve atama	
=	Bit seviyesinde <b>or</b> ve atama	
^=	Bit seviyesinde <b>eor</b> ve atama	
&&=	Mantıksal <b>and</b> ve atama	
=	Varlık ataması ( <i>Existential Operator</i> )	

İlk bakışta insanın aklını durduran bir sürü garip işaretler bunlar değil mi? Hemen örneklerle pekiştirelim.

```
a = []
a.class # => Array
a.length # => 0
```

## []= Kullanım Örneği

```
a = [] # Bu bir dizi / Array
a.[]=5, "Merhaba" # 0 indekli, 5. eleman Merhaba olsun
p a # [nil, nil, nil, nil, nil, "Merhaba"]
```

# Unary Operatörleri

**Unary** demek, `+=` , `-=` , `*=` gibi işlemleri yaptığımız operatörler. Yani `x+= 5` dediğimizde (*x'in değerine 5 ekle ve sonucu tekrar x'e ata*) aslında **Unary** operatörü kullanmış oluruz.

Keza, aşağıdaki örnekteki gibi kullanımlarda ek fayda sağlamış oluruz:

```
str = "Merhaba Dünya"

class String
  def -@
    reverse
  end
end

p str      # "Merhaba Dünya"
p -str     # "aynÜD abahreM"
```

# Global Constants (Genel Sabitler)

Ruby, ön tanımlı olarak çeşitli sabitlerle birlikte geliyor. Ben an itibariyle Mac OSX üzerinde, `rbenv` ile `ruby 2.1.0` kullanıyorum. Bu bağlamda sizin kullandığınız Ruby versiyonuna göre değişkenlikler olabilir.

Sabit	Değeri
TRUE	Anlaşılacağı gibi bu <b>true</b> değeri
FALSE	Bu da <b>false</b> değeri
NIL	<b>nil</b>
STDIN	Standart giriş. <code>\$stdin</code> için varsayılan değer.
STDOUT	Standart çıkış. <code>\$stdout</code> için varsayılan değer.
STDERR	Standart hata. <code>\$stderr</code> için varsayılan değer.
ENV	Aktif çevre değişkenlerinin ( <i>Environment Variables</i> ) bulunduğu <b>Hash</b>
ARGF	<code>\$&lt;</code> ile aynı işi yapıyor.
ARGV	<code>\$*</code> ile aynı işi yapıyor.
DATA	Herhangi bir Ruby script dosyasında, <code>__END__</code> sonrasına yazılan şeylerin saklandığı değişken.
RUBY_VERSION	"2.1.0"
RUBY_RELEASE_DATE	"2013-12-25"
RUBY_PLATFORM	"x86_64-darwin13.0"
RUBY_COPYRIGHT	"ruby - Copyright (C) 1993-2013 Yukihiro Matsumoto"
RUBY_DESCRIPTION	"ruby 2.1.0p0 (2013-12-25 revision 44422) [x86_64-darwin13.0]"
RUBY_ENGINE	"ruby"
RUBY_PATCHLEVEL	"0"
RUBY_REVISION	44422

## Bölüm 3

Bu bölümde;

- [Methods](#) (*Fonksiyonlar*)
- [Blocks](#) (*Bloklar*)
- [Proc ve Lambda](#)
- [Conditional Statements](#) (*Koşullar*)

Konularını işleyeceğiz.

# Methods (Fonksiyonlar)

Programlama parçacıklarının ve/veya ifadelerin bir araya toplandığı şeydir method. Aslında lise matematiğinden hepimiz aşinayız.

Bildiğiniz matematik fonksiyonu. Bundan böyle fonksiyon yerine **method** kullanacağım. Çünkü Ruby demek neredeyse **Obje** (*Nesne*) ve **Method** (*Method*) demek.

Önceki konularda gördüğümüz operatörlerin neredeyse hepsi bir method! Hemen **Method Definition**'a yani nasıl method tanımlandığına bir göz atalım.

```
def merhaba
  "Merhaba"
end

merhaba      # => "Merhaba"
```

`def` ve `end` anahtar kelimeleri arasına method'un adı geldi. Önce method'u tanımladık, sonra çağırdık.

Ruby'de herşey mutlaka **geriye birşey döner**. Ne demek bu? Prensip olarak method'lar zincir olarak çalıştığı için, method denen şey de aslında bir fonksiyon ve fonksiyon denen şey de bir dizi işlemin yapılp geriye sonucun dönlüldüğü bir taşıyıcı aslında.

Bir örnek vermek için hemen `irb` ye geçelim:

```
irb(main):001:0> puts "Merhaba"
Merhaba
=> nil
irb(main):002:0>
```

`puts "Merhaba"` önce işini yaptı ve çıktı olarak **Merhaba** verdi. sonra `=> nil` dikkatinizi çekti mi?

Çünkü `puts` method'u işini yaptı bitirdi ve geriye `nil` döndü! Peki daha önceki programlama tecrübelerimize dayanak, **geriye döndü** işini hangi komut yapmış olabilir?

Pek çok dilde fonksiyondan birşey geri dönmek için **return** kelimesi kullanılır. Ruby'de de kullanılır ama zorunlu değildir. Yukarıdaki `def merhaba` örneğinde `return` kullanmamamıza rağmen geriye **Merhaba** dönebildi.

Ruby'de methodlar içerisindeki çalıştırılan en son satırın değerini döndürür. Ancak siz daha öncesinde özellikle `return` anahtar kelimesi ile bir sonuç dönmezseniz. Bir örnekle konuyu pekiştirelim.

## Methods (Fonksiyonlar)

```
def merhabaBir
  m = "Merhaba"
  n = "Ornek"
end

def merhabaIki
  m = "Merhaba"
  n = "Ornek"
  return "Return Ornek"
end

puts merhabaBir    # Sonuç : Ornek
puts merhabaIki    # Sonuç : Return Ornek
```

İşte bu Ruby'nin özelliği. Kodu okurken bunu bilmezsek kafamız süper karışabilir.

`def` ile tanımlanan method'u, `undef` ile yok edebilirsiniz.

```
def merhaba
  "Merhaba"
end

merhaba # => "Merhaba"

undef merhaba

merhaba # =>
# ~> -:9:in `<main>': undefined local variable or method `merhaba' for main:Object (NameError)
```

Gördüğümüz gibi `undefined local variable or method .. Object (NameError)` hatasını aldık.

Method'lar argüman alabilir. Yani fonksiyona, doğal olarak, parametre geçebilirsiniz.

```
def merhaba(isim)
  "Merhaba #{isim}"
end

merhaba("vigo") # => "Merhaba vigo"
```

aynı örneği aşağıdaki gibi de yazabiliriz:

```
def merhaba isim
  "Merhaba #{isim}"
end

merhaba "vigo" # => "Merhaba vigo"
```



Method'u tanımlarken ve çağırırken **parantez** kullanmadık! Bu alışmanız gereken önemli konulardan birisi. Şahsen ben, daha önce hiçbir programlama dilinde böyle birşey görmedim!

Bazı durumlara, argüman alan method çağırırken, argümanın tipine göre, eğer parantez kullanmadan çağırma yaparsanız **warning** alabilirsiniz!

## Method Yazma Kuralları (*Method Conventions*)

Ruby, pek çok konuda rahat gibi görünse bile bazı kuralları var tabi. Özellikle method'ların son karakteri ile ilgili. Eğer bir method'un son karakteri `?` ise bu o method'un `true` ya da `false` yani **Boolean** bir değer döneceğini ifade eder.

```
a = "ali"
b = "ali"

a.eql? b # => true
a.eql?(b) # => true
```

`.eql?` method'u eşitliği kontrol eder ve mutlaka sonuç **Boolean** döner.

Eğer method'un son karakteri `!` (*Ünlem*) ise; bu, o method'un tehlikeli bir iş yaptığını anlatır. Yani ilgili nesnenin kopyalanmadan direk üzerinde değişiklik yapacağı anlamına gelir.

```
a = "deneme"

a.upcase # => "DENEME"
a        # => "deneme"

a.upcase! # => "DENEME"
a         # => "DENEME"
```

`a` değeri **deneme**. `.upcase` ile orijinal değeri değiştirmeden **uppercase** (*Büyük harf*) yaptık. Değeri kontrol ettiğimizde halen küçük harf olduğunu gördük. `.upcase!` kullandığımız anda değişkenin orijinal değerini de bozduk.

Eğer bir method `=` ile bitiyorsa, bu, o method'un bir **setter** method'u olduğu anlamına gelir ve **Class** ile ilgili bir konudur.

```
class User
  def email=(email)
    @email = email
  end
end

u = User.new
u # => #<User:0x007ff7229ed880>

u.email = "vigo@xyz.com"
u # => #<User:0x007ff7229ed880 @email="vigo@xyz.com">
```

## Varsayılan Argümanlar (*Default Arguments*)

Method argümanlarına varsayılan değerler atayabilirsiniz. Bu, eğer methoda gönderilmesi beklenen argüman gelmemişse otomatik olarak değer atama yapmayı sağlar.

```
def merhaba(isim="insalık!")
  "Merhaba #{isim}"
end

merhaba          # => "Merhaba insalık!"
merhaba("vigo")  # => "Merhaba vigo"
```

Parametre geçmeden çağırdığımızda, tanımladığımız varsayılan (*default*) değer atandı.

## Değişken Sayıda Argümanlar (*Variable-Length Arguments*)

Bazı durumlarda method'a değişken sayıda olarak parametre geçmek gerekebilir. Bu durumda argümanın başına `*` işareti gelir. Bu sayede o argüman artık bir dizi (*Array*) haline gelir.

```
def merhaba(*isimler)
  "Merhaba #{isimler.join(" ve ")}"
end

merhaba("vigo")          # => "Merhaba vigo"
merhaba("vigo", "yeşim", "ezel") # => "Merhaba vigo ve yeşim ve ezel"

merhaba "dünya", "uzay", "evren", "ay" # => "Merhaba dünya ve uzay ve evren ve ay"
```

Keza, şu şekilde de kullanılabilir:

```
def custom_numbers(first, second, *others)
  puts "ilk sayı: #{first}"
  puts "ikinci sayı : #{second}"
  puts "diğer sayılar : #{others.join(",")}"
end

custom_numbers 1,2,50,100 # => nil
# >> ilk sayı: 1
# >> ikinci sayı : 2
# >> diğer sayılar : 50,100
```

## Method'a Takma İsim Vermek (*Aliasing*)

Varolan bir method'u, başka bir isimle çağırmak. Bu aslında **Class** konusuyla çok ilintili ama kısaca değinmek istiyorum.

```
def merhaba(isim)
  "Merhaba! #{isim}"
end

alias naber merhaba

merhaba "Uğur" # => "Merhaba! Uğur"
naber "Uğur"   # => "Merhaba! Uğur"
```

Formül şu: `alias TAKMA AD ORJİNAL` yani `alias naber merhaba` derken, `merhaba` method'una takma ad olarak `naber` i tanımladık!

## Unutma!

- `return` kullanmadan method'dan geri dönüş yapılabilir
- Parantez kullanmadan method tanımlanabilir
- Parantez kullanmadan method çağırılıp parametre geçilebilir.
- `?` ile biten method mutlaka `true` ya da `false` döner.
- `!` ile biten orijinal değeri mutlaka değiştirir.
- `=` ile biten **setter**'dir.

# Blocks (Bloklar)

Blok olayı, bence Ruby'nin en çılgın özelliklerinden biri. Aslında bu konu, neredeyse başlı başına bir kitap konusu olabilir. Genelde **Block**, **Proc**, **Lambda** üçlemesi olarak anlatılır.

Kitabımız 101 yani giriş seviyesinde olduğu için, kafaları minimum karıştırma adına basit şekilde değineceğim.

Blok'lar, genelde [Closure](#) ya da [Anonim](#) fonksiyonlar olarak tanımlanır. Sanki method içinde başka bir method'u işaret eden ya da değişkenleri method'lar arasında paylaşan kapalı bir ortam gibidirler.

Genelde ya `{ }` ile ya da `do/end` ile sarmalanmışlardır.

```
family_members = ["Yeşim", "Ezel", "Uğur", "Ömer"]

family_members.each do |member_name|
  puts member_name
end

# Yeşim
# Ezel
# Uğur
# Ömer
```

Aynı kod:

```
family_members = ["Yeşim", "Ezel", "Uğur", "Ömer"]

family_members.each { |member_name| puts member_name }

# Yeşim
# Ezel
# Uğur
# Ömer
```

şeklinde de yazılabilirdi. `do/end` ya da `{ }` arasında kalan kısım **Block** kısmıdır.

`family_members` bir **Array** yani dizidir. Eğer `puts family_members.class` dersek bize `Array` olduğunu söyler. Array'in `each` method'u bize block işleme şansını sağlar.

Komut satırında `ir Array#each` dersek bize Array'in **each** method'uyla ilgili tüm bilgiler gelir.

`do` komutundan hemen sonra gelen `|member_name|` bizim kafamıza göre tanımladığımız bir değişkendir ve Array'in her elemanı bu değişkene atanır.

**Enumeration** bölümünde bunlardan sıkça bahsedeceğiz. Blockların esas gücü `yield` olayından gelir. Hemen bir örnek verelim:

```
def test_function
  yield
end

test_function {
  puts "Merhaba"
}

# Merhaba

test_function do
  puts "Ben block içinden geliyorum"
end

# Ben block içinden geliyorum

test_function do
  [1, 2, 3, 4].each do |n|
    puts "Sayı #{n}"
  end
end

# Sayı 1
# Sayı 2
# Sayı 3
# Sayı 4
```

`test_function` adında bir fonksiyonum var (*yani method'um var*) Hiç parametre almıyor! ama **Block** alıyor. İlkinde **curly brace** ile (*yani { ve }*), ikincisinde `do/end` ile, son örnekte `do/end` ile ve iç kısımda başka bir iterasyonla kullandım.

Kabaca, fonksiyona kod bloğu geçtim.

Peki, ya şu şekilde kullansaydım `test_function` ? yani hiçbirşey geçmeden? Alacağım hata mesajı:

```
no block given (yield)
```

olacaktı. Demek ki block geçilip geçirilmediğini anlamamanın bir yolu var :)

```
def test_function
  if block_given?
    yield
  else
    puts "Lütfen bana block veriniz!"
  end
end
```

`block_given?` ile bu kontrolü yapabiliyoruz. Şimdi biraz daha kafa karıştıralım :)

```
def numerator
  yield 10
  yield 4
  yield 8
end

numerator do |number|
  puts "Geçilen sayı #{number}"
end
```

Örnekte, `yield` block içinden gelen kod bloğunu bir **fonksiyon** gibi çağırıyor, çağırırken de bloğa **parametre** geçiriyor. Dikkat ettiyseniz kaç tane `yield` varsa o kadar kez block çağırıldı (*call edildi*)

```
def print_users
  ["Uğur", "Yeşim", "Ezel"].each do |name|
    yield name
  end
end

print_users do |name|
  puts "Kullanıcı Adı: #{name}"
end

# Kullanıcı Adı: Uğur
# Kullanıcı Adı: Yeşim
# Kullanıcı Adı: Ezel
```

Fonksiyon içine fonksiyon geçtik gibi.

**Enumeration / Number** bölümünde de göreceğiz ama hemen hızlı bir-iki örnek vermek istiyorum blok kullanımıyla ilişkili.

```
5.times { puts "Merhaba" }
5.times { |i| puts "Sayı #{i}" }
5.times do |i|
  puts "Sayı #{i}"
end
```

Not: Aslında `times` sayılara ait bir method ve yukarıdaki örneklerde gördüğünüz gibi blok geçebiliyoruz kendisine.

# Proc ve Lambda

**Block** kullanımını daha dinamik ve programatik hale getirmek için **Procedures** ya da genel adıyla **Procs**, object-oriented Ruby'nin vazgeçilmezlerindendir.

Bazen, her seferinde aynı bloğu sürekli geçmek gerektiğinde imdadımıza **Proc** yetişiyor. Nasıl mı?

Düşünün ki bir method'unuz (*fonksiyonunuz*) olsun, ve bu method'u dinamik olarak programlayabilseniz?

```
def multiplier(with)
  return Proc.new {|number| number * with }
end
```

Çarpma yaptıran dinamik bir fonksiyon. Sayıyı kaç ile çarpacaksak `with` e o sayıyı geçiyoruz.

```
multiply_with_5 = multiplier(5)
```

Elimizde geçtiğimiz sayıyı **5** ile çarpacak, fonksiyondan türemiş bir fonksiyon oluştu. Eğer `multiply_with_5` acaba neymiş dersek?

```
puts multiply_with_5
# =<Proc:0x007fcc9c94a938@/Users/vigo/Desktop/ruby101_book_tests.rb:2>

puts multiply_with_5.class
# Proc
```

Gördüğünüz gibi elimizde bir adet `Proc` objesi var!. Haydi kullanalım!

```
puts multiply_with_5.call(5) # 25
puts multiply_with_5.call(10) # 50
```

Bu kadar kasmadan, basit bir method ile yapsaydık:

```
def multiplier(number, with)
  return number * with
end

puts multiplier 5, 5 # 25
```

şeklinde olurdu. Benzer bir örnek daha yapalım:

```
multiplier = Proc.new { |number|
  number.collect { |n| n ** 2 }
}

multiplier.call(1)      # => [1]
multiplier.call(2,4,6)  # => [4, 16, 36]
multiplier[2,4,6].class # => Array
```

Az ileride `Array` konusunda göreceğimiz `collect` method'unu kullandık. Bu method ile `Array`'in her elemanını okuyor ve karesini `n ** 2` alıyoruz. `*` işareti yine `Array`'de göreceğimiz **splat** özelliği. Yani geçilen parametre grubunu **Array** olarak değerlendiriyoruz.

## Lambda

Python'la uğraşan okurlarımız **Lambda**'ya aşina olabilirler. **Proc** ile ilk göze çarpan fark, argüman olarak geçilen parametrelerin sayısı önemlidir `Lambda`'da. Aynı **Proc** gibi çalışır.

```
custom_print = lambda { |txt| puts txt }
custom_print.call("Hello") # Hello
```

Eğer 2 parametre geçseydik:

```
custom_print = lambda { |txt| puts txt }
custom_print.call("Hello", "World")

# ArgumentError: wrong number of arguments (2 for 1)
```

Ya da;

```
l = lambda { "Merhaba" }
puts l.call # Merhaba
```

Başka bir kullanım;

```
l = lambda do |user_name|
  if user_name == "vigo"
    "Selam vigo! nasılsın?"
  else
    "Selam sana #{user_name}"
  end
end

puts l.call("vigo") # Selam vigo! nasılsın?
puts l.call("uğur") # Selam sana uğur
```



# Proc ve Lambda Farkı

```
def arguments(input)
  one, two = 1, 2
  input.call(one, two)
end

puts arguments(Proc.new{ |a, b, c| puts "#{a} ve #{b} ve #{c.class}" })

# 1 ve 2 ve NilClass

puts arguments(lambda{ |a, b, c| puts "#{a} ve #{b} ve #{c.class}" })

# ArgumentError: wrong number of arguments (2 for 3)
```

Aynı kodu kullandık, **Lambda** yeterli parametre almadığı için hata verdi.

## Proc'u Block'a çevirmek

Elimizdeki **Array** elemanlarının her birini bir fonksiyona tabii tutsak? Dizinin her elemanını ekrana yazdıran birşey?

```
items = ["Bu bir", "bu iki", "bu üç"]
print_each_element = lambda { |item| puts item }
items.each(&print_each_element)
```

Bu örnekte `items.each(&print_each_element)` satırı Proc'u Block'a çevirdiğimiz yer. `&` işin sırrı. `each` 'den gelen eleman, sanki method çağırır gibi `print_each_element` e pas ediliyor, karşılayan da `{ }` içinde tanımlı kod bloğunu çalıştırıyor.

Keza aynı işi;

```
items = ["Bu bir", "bu iki", "bu üç"]
def print_each_element(item)
  puts item
end
items.each(&method(:print_each_element))
```

şeklinde de yapabildik!

# Conditional Statements (Koşullar)

Verilen ifadenin doğru ya da yanlış olduğunun testi yapılır, akış **true** ya da **false** durumuna göre seyreder.

## if durumu

`if a == b then` dediğimizde, **Eğer "a, b'ye eşittir" önermesi doğruysa** demiş oluruz.

`if a != b then` dediğimizde, **Eğer "a, b'ye eşit değildir" önermesi doğruysa** demiş oluruz.

Aynı mantıkta, `a > b` ("a, b'den büyüktür"), `a < b` ("a, b'den küçüktür"), `a >= b` ("a, b'en büyük ya da eşittir"), `a <= b` ("a, b'den küçük ya da eşitse") şeklinde önermeler de kurulabilir.

## Negatiflik Operatörü

`!` işareti önermenin sol tarafında kullanılırsa, negatiflik ya da olumsuzluk kontrolü olduğu anlamındadır.

`if !a == b then puts "a, b'ye eşit değil" end` ya da `if !a > b then puts "a, b'den büyük değil" end` gibi kullanılır. İçerideki önermenin tersi doğruysa demektir.

`if !a == b then` örneğini daha iyi inceleyecek olursak. Örnekte önerme "a, b'ye eşittir" önermesinin olumsuzluğu, yani "a, b'ye eşit değildir"dir. Yani bu `if` durumu şuna karşılık gelmektedir: **"a, b'ye eşit değildir" doğru ise**

## Çoklu Kontrol

Eğer önermeler arasında `and` (ve), `or` (veya) ya da bunların kısa yollarını (`and` için `&&`, `or` için `||`) kullanırsak birden fazla şeyi kontrol etmiş oluruz.

```
a = 5
b = 10
if a == 5 && b == 10
  puts "İşleme devam edebiliriz!"
end
```

`if a == 5 && b == 10` yerine `if a == 5 and b == 10` de yazabilirdik.

Ruby'nin konuşma diline yakın olması sebebiyle, çok daha anlaşılır kontrol satırları yazmak mümkün. Örneğin, **Eğer, a, 5'e eşitse Merhaba Yaz** demek için ilk akla gelen yöntem:

```
if a == 5
  puts "Merhaba"
end
```

ama bunu çok daha basit hale getirebiliriz:

`puts "Merhaba" if a == 5` Tek satırda, `if` i koşul sonucunda olacak şeyin sağına yazmak yeterlidir.

## elsif

Programlama mantığında pek de sevmediğim (*daha düzgün yöntemleri var*) ama bazen mecbur kaldığımız bir durumdur.

```
if a == b
  puts "a, b'ye eşit"
elsif a < b
  puts "a, b'den küçük"
else
  puts "a, b'ten büyük"
end
```

İlk olarak `a == b` kontrolü yapılır, eğer sonuç `false` ise, `a < b` kontrol edilir, o da `false` ise en sondaki `else` devreye giriyor ve çıktı olarak "a, b'ten büyük" yazdırılıyor.

## unless

Bu, aslında `if` in tersi gibi. Daha doğrusu `if not` anlamında. **Eğer a, b'ye eşit değilse** demek için;

```
unless a == b
  puts "Eşit değil"
end
```

Aynı mantıkta `puts "Eşit değil" unless a == b` şeklinde de yazabiliriz. Semantik olarak olumsuzluk kontrolü yaparken `unless` kullanılması önerilir. Kodu okuma ve anlama açısından kolay olması için.

## while , break , until Döngüleri

Tanımlanan önerme `true` olduğu sürece **loop** yani döngü çalıştırma kontrolüdür.

```
i = 0
while i < 5 do
  puts "i = #{i}"
  i += 1
end
```

Eğer `i += 1` yani `i` yi bir arttır, demezsek sonsuz döngüye gireriz. Eğer belli bir anda döngüyü kırmak istersek,

```
i = 0
while i < 5 do
  puts "i = #{i}"
  break if i == 3
  i += 1
end
```

`i` **3** olduğunda loop devre dışı kalır...

Aynı `unless` mantığında, `until` kullanılır loop'larda.

```
i = 0
until i == 10 do
  puts "i = #{i}"
  i += 1
end
```

Yani `i` **10**'a eşit olmadığı sürece bu loop çalışır.

## case , when Yapısı

`elsif` yerine kullanılması muhtemel, daha anlaşılır kontrol mekanizmasıdır. Hemen örneğe bakalım:

```
computer = "c64"
year = case computer
  when "c64" then "1982"
  when "c16" then "1984"
  when "amiga" then "1985"
  else
    "Tarih bilinmiyor"
end

puts "#{computer} çıkış yılı #{year}"

# c64 çıkış yılı 1982
```

Yukarıdaki kodu bir ton `if` , `elsif` ile yapmak yerine, `when` ve `then` ile daha anlaşılır hale getirdiğimizi düşünüyorum.

`when` kullanırken **range** (*aralık*) belirmesi de yapma şansı var.

```
student_grade = 8
case student_grade
when 0
  puts "Çok kötü"
when 1..4
  puts "Başarısız"
when 5..7
  puts "İyi"
when 8..9
  puts "Çok İyi"
when 10
  puts "Süper"
end
```

Eğer not **1** ile **4** aralığındaysa (ve dahil ise) ya da **5** ile **7** aralığındaysa gibi bir kontrol ekledik.

## for Döngüsü

**1**'den **10**'a kadar (1 ve 10 dahil) bir döngü yapalım:

```
for i in 1..10
  puts "i = #{i}"
end

# i = 1
# i = 2
# i = 3
# i = 4
# i = 5
# i = 6
# i = 7
# i = 8
# i = 9
# i = 10
```

Aynı işi çok daha kolay yapmanın yolunu **5.Bölüm**'de **Iteration** kısmında göreceğiz!

## Ternary Operatörü

Kısaltılmış `if` yapısıdır. Hemen hemen pek çok dilde kullanılan, **Eğer şu doğru ise bu değilse bu** ifadesi için kullanılır.

```
amount = 2
pluralize = amount == 1 ? "apple" : "apples"
puts "#{amount} #{pluralize}."
```

Bu örnekte **Ternary** olarak `amount == 1 ? "apple" : "apples"` kullanılmış, eğer `amount` **1** ise "apple" dönecek, değil ise "apples" dönecek. Yani `pluralize` değişkenine kontrolden dönen atanıyor.

## BEGIN ve END

Ruby'de ilginç bir özellik de, kodun çalışmasından önceye ve sonraya bir ek takabiliyoruz.

Aşağıdaki örnekte BEGIN block'undaki kodlar program başladığında, END block'undaki kodlar program bitmeden hemen önce çalışacaktır.

```
BEGIN { puts "Kodun başlama saati #{Time.now.to_s}" }
END { puts "Kodun bitme saati #{Time.now.to_s}" }

def say_hello(username)
  "Merhaba #{username}"
end

puts say_hello "Uğur"
sleep 5 # zaman farkı için 5 saniye bekle

# Kodun başlama saati 2014-08-04 09:30:24 +0300
# Merhaba Uğur
# Kodun bitme saati 2014-08-04 09:30:29 +0300
```

# Bölüm 4

Bu bölümde Ruby'deki veri tipleri yani **Data Types**'ı inceleyeceğiz.

- Object
- Number
- String
- Array
- Hash
- Symbol
- Class
- Module

# Object (Nesne)

Ruby, **Object Oriented Programming**'in dibidir :) Herşey nesnelerden oluşur. Nasıl mı? hemen basit bir değişken oluşturup içine bir tekst yazalım.

```
mesaj = "Merhaba"
```

mesaj değişkeninin türü ne?

```
mesaj.class # => String
```

Bu değişken **String** nesnesinden türemiş. Peki, **String** nereden geliyor?

```
mesaj.class.superclass # => Object
```

Dikkat ettiyseniz burada `superclass` kullandık. Yani bu hiyerarşideki bir üst sınıfı arıyoruz. Karşımıza ne çıktı? **Object**. Peki acaba **Object** nereden türemiş?

```
mesaj.class.superclass.superclass # => BasicObject
```

Hmmm.. Peki **BasicObject** nereden geliyor?

```
mesaj.class.superclass.superclass.superclass # => nil
```

İşte şu anda dibi bulduk :) Demek ki hiyerarşi;

```
BasicObject > Object > String
```

şeklinde bir hiyerarşi söz konusu.

Peki, sayılarda durum ne?

```
numara = 1
numara.class # => Fixnum
numara.class.superclass # => Integer
numara.class.superclass.superclass # => Numeric
numara.class.superclass.superclass.superclass # => Object
numara.class.superclass.superclass.superclass.superclass # => BasicObject
numara.class.superclass.superclass.superclass.superclass.superclass # => nil
```



Ufff... bir an için bitmeyecek sandım :) Çok basit bir sayı tanımlası yaptığımızda bile, arka plandaki işleyiş yukarıdaki gibi oluyor. Yani

```
BasicObject > Object > Numeric > Integer > Fixnum
```

şeklinde yine ana nesne **BasicObject** olmak koşuluyla uzun bir hiyerarşi söz konusu.

Herşey **BasicObject** den türüyor, bu yüzden de aslında herşey bir **Class** dolayısıyla bu durum dile çok ciddi esneklik kazandırıyor.

## Nesne Metodları (Object Instance Methods)

Şimdi boş bir nesne oluşturalım. **Class** bölümünde daha detaylı göreceğimiz **instantiate** işlemiyle `new` methodunu kullanarak;

```
o = Object.new # => #<Object:0x007fe552099a68>
o.__id__       # => 70311450299700
```

yaptığımızda, oluşan nesnenin hafızada **unique** (yani *bundan sadece bir tane*) bir **identifier**'i (*kabaca buna kimlik diyelim*) yani **ID**'si olduğunu görürüz. `__id__` yerine `object_id` yani `o.object_id` şeklinde de kullanabiliriz.

Eğer `hash` method'unu çağırırsak, Ruby bize ilgili objenin `Fixnum` türünde sayısal değerini üretir ve verir.

```
o = Object.new # => #<Object:0x007f8c3b0a3420>
o.__id__       # => 70120131336720
o.object_id    # => 70120131336720
o.hash         # => -229260864779029724
```

Neticede **String** de bir nesne ve;

```
t = String.new("Hello") # => "Hello"
t.__id__                # => 70170408456140
t.methods               # => [:<=>, :==, :===, :eql?, :hash, :casecmp, :+, :*, :%, :[], :[]=, :insert, :length, :size, :bytesize, :empty?, :=~, :match, :succ, :succ!, :next, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :getbyte, :setbyte, :byteslice, :scrub, :scrub!, :freeze, :to_i, :to_f, :to_s, :to_str, :inspect, :dump, :upcase, :downcase, :capitalize, :swapcase, :upcase!, :downcase!, :capitalize!, :swapcase!, :hex, :oct, :split, :lines, :bytes, :chars, :codepoints, :reverse, :reverse!, :concat, :<<, :prepend, :crypt, :intern, :to_sym, :ord, :include?, :start_with?, :end_with?, :scan, :ljust, :rjust, :center, :sub, :gsub, :chop, :chomp, :strip, :lstrip, :rstrip, :sub!, :gsub!, :chop!, :chomp!, :strip!, :lstrip!, :rstrip!, :tr, :tr_s, :delete, :squeeze, :count, :tr!, :tr_s!, :delete!, :squeeze!, :each_line, :each_byte, :each_char, :each_codepoint, :sum, :slice, :slice!, :partition, :rpartition, :encoding, :force_encoding, :b, :valid_encoding?, :ascii_only?, :unpack, :encode, :encode!, :to_r, :to_c, :>, :>=, :<, :<=, :between?, :nil?, :!~, :class, :singleton_class, :clone, :dup, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :frozen?, :methods, :singleton_methods, :protected_methods, :private_methods, :public_methods, :instance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_defined?, :remove_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?, :extend, :display, :method, :public_method, :singleton_method, :define_singleton_method, :object_id, :to_enum, :enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]

t.method(:upcase).call # => "HELLO"
```

`t.methods` ise **String**'den türeyen `t` ye ait tüm method'ları listeledik. Sonuç **Array (Dizi)** olarak geldi ve bu dizinin tüm elemanları `:` işaretiyle başlıyor. Çünkü bu elemanlar birer **Symbol**.

`t.method(:upcase).call` da ise, `t` 'nin `:upcase` method'unu `call` ile çağırır. Aslında yaptığımız iş: `"hello".upcase` ile birebir aynı.

Acaba bu nesne ne?

`t.is_a?(String)` # => true `is_a?` method'u ile nesnenin türünü kontrol edebiliriz.

Diğer dillerin pek çoğunda (özellikle *Python*) bir işi yapmanın bir ya da en fazla iki yolu varken, **Ruby** bu konuda çok rahattır. Bir işi yapmanın her zaman birden fazla yolu olur ve bunların neredeyse hepsi doğrudur. (*Kullanıldığı yere ve amaca bağlı olarak*)

`is_a?` yerine `kind_of?` da kullanabiliriz!

Bir nesneye ait hangi method'ların olduğunu;

```

o = Object.new
o.methods # => [:nil?, :==, :=~, :!~, :eql?, :hash, :<=>, :class, :singleton_class, :clone, :dup, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :freeze, :frozen?, :to_s, :inspect, :methods, :singleton_methods, :protected_methods, :private_methods, :public_methods, :instance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_defined?, :remove_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?, :extend, :display, :method, :public_method, :singleton_method, :define_singleton_method, :object_id, :to_enum, :enum_for, :==, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]

o.public_methods # => [:nil?, :==, :=~, :!~, :eql?, :hash, :<=>, :class, :singleton_class, :clone, :dup, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :freeze, :frozen?, :to_s, :inspect, :methods, :singleton_methods, :protected_methods, :private_methods, :public_methods, :instance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_defined?, :remove_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?, :extend, :display, :method, :public_method, :singleton_method, :define_singleton_method, :object_id, :to_enum, :enum_for, :==, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]

o.private_methods # => [:initialize_copy, :initialize_dup, :initialize_clone, :sprintf, :format, :Integer, :Float, :String, :Array, :Hash, :warn, :raise, :fail, :global_variables, :__method__, :__callee__, :__dir__, :eval, :local_variables, :iterator?, :block_given?, :catch, :throw, :loop, :respond_to_missing?, :trace_var, :untrace_var, :at_exit, :syscall, :open, :printf, :print, :putc, :puts, :gets, :readline, :select, :readlines, :`, :p, :test, :srand, :rand, :trap, :exec, :fork, :exit!, :system, :spawn, :sleep, :exit, :abort, :load, :require, :require_relative, :autoload, :autoload?, :proc, :lambda, :binding, :caller, :caller_locations, :Rational, :Complex, :set_trace_func, :gem, :gem_original_require, :initialize, :singleton_method_added, :singleton_method_removed, :singleton_method_undefined, :method_missing]

o.protected_methods # => []

o.public_methods(false) # => []

```

`public_methods` default olarak `public_methods(all=true)` şeklinde çalışır. Eğer parametre olarak `false` geçerse ve bu bizim oluşturduğumuz bir nesne ise, sadece ilgili nesnenin **public\_method**'ları geri döner.

Başta belirttiğimiz gibi, basit bir nesne bile **Object**'den türediği için ve default olarak bu türeme esnasında tüm özellikler diğerine geçtiği için, sadece sizin method'larınızı görüntülemek açısından `false` olayı çok işe yarar.

## Method Missing

Bence Ruby'nin en süper özelliklerinden biridir. Olmayan bir method'u çağırdığınız zaman tetiklenen method `method_missing` method'udur. Ruby on Rails framework'ü neredeyse bu mekanizma üzerine kurulmuştur. 3 parametre alır; çağırılan method, eğer parametre geçilmişse parametreler, eğer block geçilmişse block.

```
class User
  def method_missing(method_name, *args, &block)
    if method_name == :show_user_info
      "This user has no information"
    else
      "You've called #{method_name}, You've passed: #{args}"
    end
  end
end

u = User.new
u.show_user_info # => "This user has no information"
u.show_user_age  # => "You've called show_user_age, You've passed: []"
```

`user` adında bir Class'ımız var. İçinde hiçbir method tanımlı değil. `u.show_user_info` satırında, olmayan bir method'u çağırıyoruz. Tanımladığımız `method_missing` method'u ile olmayan method çağırılmasını yakalıyoruz. Eğer `show_user_info` diye bir method çağırılsa yakalıyoruz, bunun dışında bir şey olursa da method adını ve geçilen parametreleri gösteriyoruz.

Bu sayede `NoMethodError` hatası almadan işimize devam edebiliyoruz.

Anlamak açısından, Roman rakamları için bir sınıf yaptığınızı düşünün. Sadece örnek olması için gösteriyorum, C,X ve M için;

```
class Roman
  def roman_to_str(str)
    case str
    when "x", "X"
      10
    when "c", "C"
      100
    when "m", "M"
      1000
    end
  end
  def method_missing(method)
    roman_to_str method.id2name
  end
end

r = Roman.new
r.x # => 10
r.X # => 10
r.C # => 100
r.M # => 1000
```

Bunu geliştirip "MMCX" ya da "III" gibi gerçek dönüştürme işini yapabilirsiniz.

**respond\_to\_missing?**

Yukarıdaki örnekte, olmayan method'ları ürettik. Peki, acaba bu olmayan method'ları nasıl çağırabilir ya da kontrol edebiliriz? Normalde, bir Class'ın hangi method'u olduğunu `respond_to?` ile öğreniyorduk. Örneğe uygulayalım;

`r.c` derken aslında `:c` method'unu çağırıyoruz. Peki böyle bir method var mı?

```
r.method(:C)      # => `method': undefined method `C' for class `Roman' (NameError)
```

Nasıl yani? peki kontrol edelim?

```
r.respond_to?(:C) # => false
```

Çünkü biz `:c` yi dinamik olarak ürettik ama öylece ortada bıraktık. Yapmamız gereken `respond_to_missing?` ile gereken cevabı vermektir:

```
class Roman
  def roman_to_str(str)
    case str
    when "x", "X"
      10
    when "c", "C"
      100
    when "m", "M"
      1000
    end
  end
  def method_missing(method)
    roman_to_str method.id2name
  end
  def respond_to_missing?(method_name, include_private = false)
    [:x, :X, :c, :C, :m, :M].include?(method_name) || super
  end
end

r.method(:C)      # => #<Method: Roman#C>
r.respond_to?(:C) # => true
r.respond_to?(:Q) # => false # olmayan method
```

# Number (Sayılar)

Sayılar, temel öge olmayıp, direk nesneden türemişlerdir. Türedikleri nesne de Ruby'nin sayılar için olan **base class**'ıdır.

Örneğin **3** sayısına bakalım:

```
3.class # => Fixnum
3.class.superclass # => Integer
3.class.superclass.superclass # => Numeric
3.class.superclass.superclass.superclass # => Object
```

**Numeric** sınıfıdır asıl olan. İlk türediği sınıfı **Fixnum** dir. Örnekte gördüğünüz gibi;

**Fixnum > Integer > Numeric** şeklinde bir hiyeraşi söz konusudur.

```
2014.class # => Fixnum
2_014.class # => Fixnum
201.4.class # => Float
1.2e3.class # => Float
7e4.class # => Float
7E-4.class # => Float
0664.class # => Fixnum
0xffff.class # => Fixnum
0b1111.class # => Fixnum
45678327041234321312.class # => Bignum
```

Ruby'de sayı işlerinde **\_** hiçbir etki yapmaz. Bir şekilde okumayı kolaylaştırmak için kullanılır. Örnekteki **2014** ile **2\_014** aynı şeydir. Büyük sayıları yazarken; **1\_345\_201** gibi bir ifade **1345201** 'dir aslında.

Ondalık sayılarda **.** kullanılır. **Octal** yani 8'lik sayı sistemi için direk **0** ile **0664** gibi kullanılır. 16'lık yani **Hexadecimal** sayı sistemi için, css dünyasından tanıdığınız **beyaz** rengini ifade etmek için **\$fff** yerine **0xffff** şeklinde bir kullanım mevcut. 2'lik yani **Binary** sayı sistemi için **0b** ile başlamak yeterlidir. **Scientific Notation** ifadeleri için de **1.2e3** ya da **7e4** gibi kullanımlar mümkündür.

## Number Method'ları

**5** sayısı **Fixnum** sınıfındandır ve neticede üst sınıfları;

```
Numeric -> Integer -> Fixnum
```

şeklinde olduğu için (*en üstte Numeric*) ilgili üst sınıfların method'ları da `Fixnum` tarafından kullanılabilir durumdadır.

Her zamanki gibi, **acaba bu sınıfa ait methodlar neymiş?** dediğimiz an bir ton method gelir karşımıza;

```
5.methods # => [:to_s, :inspect, :-@, :+, :-, :*, :/, :div, :%, :modulo, :divmod, :fdi
v, :**, :abs, :magnitude, :==, :===, :<=>, :>, :>=, :<, :<=, :~, :&, :|, :^, :[], :<<,
: >>, :to_f, :size, :bit_length, :zero?, :odd?, :even?, :succ, :integer?, :upto, :down
to, :times, :next, :pred, :chr, :ord, :to_i, :to_int, :floor, :ceil, :truncate, :round
, :gcd, :lcm, :gcdlcm, :numerator, :denominator, :to_r, :rationalize, :singleton_metho
d_added, :coerce, :i, :+@, :eql?, :remainder, :real?, :nonzero?, :step, :quo, :to_c, :
real, :imaginary, :imag, :abs2, :arg, :angle, :phase, :rectangular, :rect, :polar, :co
njugate, :conj, :between?, :nil?, :=~, :!~, :hash, :class, :singleton_class, :clone, :
dup, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :freeze, :frozen?, :m
ethods, :singleton_methods, :protected_methods, :private_methods, :public_methods, :in
stance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_d
efined?, :remove_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :pu
blic_send, :respond_to?, :extend, :display, :method, :public_method, :singleton_method
, :define_singleton_method, :object_id, :to_enum, :enum_for, :equal?, :!, :!=, :instan
ce_eval, :instance_exec, :__send__, :__id__]
```

Bunların içinden en sık kullanılanlara ve kullanım şekillerine değineceğim.

```

5.to_s # => "5"
# Sayısal değeri String'e çevirdik

-5.abs # => 5
# Mutlak değer

5.zero? # => false
# Sıfır mı?

5.even? # => false
# Çift sayı mı?

5.odd? # => true
# Tek sayı mı?

5.next # => 6
# Sonraki sayı?

5.pred # => 4
# Önceki sayı?

3.14.floor # => 3
# Taban değeri

3.14.ceil # => 4
# Tavan değeri

1.49.round # => 1
1.51.round # => 2
# Yuvarlama

1.bit_length # => 1
15.bit_length # => 4
255.bit_length # => 8
# Bit cinsinden uzunluğu/boyu

1.size # => 8
10.size # => 8
10242048.size # => 8
1024204810242048102420481024.size # => 12
# Byte cinsinden kapladığı yer

```

`upto` , `downto` gibi iterasyonla ilgili olanları **Enumeration ve Iteration** bölümünde göreceğiz!



# String

Kabaca, insanların anlayacağı şekilde tekst / metin bilgisi içeren nesnelerdir. Örneğin;

```
m = "Merhaba"
m.class # => String
```

şeklinde. Tanımlama yaparken **tek** ya da **çift** tırnak kullanılabilir ama aralarında fark vardır. **Expression Substitution** ya da **String Interpolation** olarak geçen, **String** içinde değişken kullanımı esnasında **çift tırnak** kullanmanız gerekir.

```
m = "Merhaba"
puts "#{m} Uğur" # Merhaba Uğur
puts '#{m} Uğur' # #{m} Uğur
```

Tek tırnak kullandığımız örnekte `#{m}` değişkeni işlenmeden olduğu gibi çıktı vermiştir. Aynı şekilde **escape codes** yani görünmeyen özel karakterleri de sadece çift tırnak içinde kullanmak mümkündür.

Çift tırnak içinde kullanılan `#{BURAYA RUBY KODU GELİR}` çok işe yarar. `{` ve `}` arasında kod çalıştırmamızı sağlar.

```
puts "Saat: #{Time.now}" # Saat: 2014-08-12 10:37:22 +0300
```

dediğimizde; Ruby Kernel'dan `Time` nesnesinin `now` method'unu çalıştırmış oluruz.

```
puts "Merhaba\nDünya"

# Merhaba
# Dünya
```

`\n` **New Line** ya da **Line Feed** ya da yeni satıra geçme karakteri çift tırnakta çalışır.

Escape Kod	Anlamı
\n	Yeni satır (0x0a)
\s	Boşluk (0x20)
\r	Satır Başı (0x0d)
\t	Tab Karakteri (0x09)
\v	Dikey Tab (0x0b)
\f	Formfeed (0x0c)
\b	Backspace (Bir geri) (0x08)
\a	Bell/Alert (Uyarı) (0x07)
\e	Escape (0x1b)
\nnn	Octal, 8'lik değer
\xnn	Hexadecimal, 16'lık değer
\unnnn	Unicode: U+nnnn (Ruby 1.9+)
\cx	Control-x
\C-x	Control-x
\M-x	Meta-x
\M-\C-x	Meta-Control-x
\x	x'in kendisi (\ " çift tırnak demektir.)

String'ler **byte array**'lerden oluşur yani elemanları **byte** cinsinden dizidir aslında.

```
m = ""
m << 65
puts m # A
```

`m` boş bir String, diziye eleman ekler gibi ( `<<` *diziye eleman ekler, az sonra göreceğiz*) içine **65** ekledik. Bu `A` harfinin 10'luk sayı sistemindeki *ASCII* değeridir. Aslında `m = "A"` yaptık :)

Eğer **65** in karakter setindeki değeri neydi? dersek `put 65.chr` yaptığımızda bize `A` döner. **0** ile **255** arası değerlerdir bunlar.

Daha ilginç bir olay;

```
puts "öz" "yıl" "maz" "el" # özyılmazel
```

yani `"tekst" "tekst" "tekst"` şeklinde bir kullanım mevcuttur.

## String Literals (String Kalıpları)

Ruby'de, yine diğer dillerde olmayan ilginç bir özellik. `%` işareti ve sonrasında gelen bazı karakterler yardımıyla enteresan şeyler yapmak mümkün:

**%**

Süslü parantezler arasında kalan herşey **concat** (*yani toplanarak*) edilir ve **String** olarak çıktı verir ve tırnakları **escape** eder.

```
%{Merhaba Dünya Ben vigo nasılsınız?} # => "Merhaba Dünya Ben vigo nasılsınız?"  
%{Bu işlemlerin %80'i "uydurma"} # => "Bu işlemlerin %80'i \"uydurma\""
```

aynı işi; `%|Merhaba Dünya Ben vigo nasılsınız?|` süslü parantez yerine **pipe** `|` kullanarak da yapabilirsiniz!

**%w**

Hızlıca **Array** üretmeyi sağlar:

```
%w{foo bar baz} # => ["foo", "bar", "baz"]  
%w{foo bar baz}.class # => Array
```

**%i**

İçinde **Symbol** olan **Array** üretir:

```
%i{foo bar baz} # => [:foo, :bar, :baz]
```

**%q** ve **%Q**

`q` tek tırnak, `Q` çift tırnakla sarmalamış gibi yapar:

```
person = "Uğur"  
%q{Merhaba #{person}} # => "Merhaba \#{person}"  
%Q{Merhaba #{person}} # => "Merhaba Uğur"
```

**%s**

**Symbol**'e çevirir:

## String

```
%s{my_variable} # => :my_variable  
%s{email} # => :email
```

## %r

**Regular Expression'a** çevirir:

```
%r{(.*)hello}i # => /(.* )hello/i
```

## %X

Ruby'de **back tick** kullanarak **shell** komutu çalıştırabilirsiniz. Yani Linux ve Mac kullanan okuyucular **Terminal** ile haşır-neşir olmuşlardır. Örneğin, bulunduğunuz dizindeki dosya listesini almak için `ls` komutunu kullanırsınız. Örneğin kendi **home** dizinimde `ls` yaptığımda (*OSX kullanıyorum*)

```
ls -l # alt alta listemelek için
```

```
Applications  
Desktop  
Development  
Documents  
Dotfiles  
Downloads  
Library  
VirtualBox VMs  
Works  
bin
```

gibi bir liste alıyorum. Yapacağım bir Ruby uygulamasında;

```
%x{ls -l $HOME} # => "Applications\nDesktop\nDevelopment\nDocuments\nDotfiles\nDownloa  
ds\nLibrary\nVirtualBox VMs\nWorks\nbin\n"
```

Tüm liste tek bir String olarak geldi ve `\n` karakteri ile birleşti çünkü sonuç alt alta satır satır geldi. Eğer;

```
%x{ls -l $HOME}.split("\n")
```

deseydim sonuç bana dizi olarak gelecekti!

```
# ["Applications", "Desktop", "Development", "Documents", "Dotfiles", "Downloads", "Library", "VirtualBox VMS", "Works", "bin"]

%x{ruby --copyright} # => "ruby - Copyright (C) 1993-2014 Yukihiro Matsumoto\n"
```

## Here Document Kullanımı

Uzun metin kullanımlarında çok işe yarar:

```
mesaj = <<END
Merhaba nasılsınız?
Biz de çok iyiyiz
Görüşürüz!
END
puts mesaj

# Merhaba nasılsınız?
# Biz de çok iyiyiz
# Görüşürüz!
```

Bu örnekte `mesaj = <<END` ile `END` kelimesini görene kadar içinde ne varsa kullan diyoruz! Daha da çılgın bir kullanım şekli;

```
mesaj = [<<BİR, <<İKİ, <<ÜÇ]
  Bu Bir
BİR
  Bu iki....
İKİ
  Bu da ÜÜÜÜÜÜÜÜÇ
ÜÇ

puts mesaj

# - Bu Bir
# - Bu iki....
# - Bu da ÜÜÜÜÜÜÜÜÇ
```

## String Method'ları

### String % argüman -> yeni String

```
"Merhaba %s" % "Uğur" # => "Merhaba Uğur"
"Sayı: %010d" % 2014 # => "Sayı: 0000002014"
"Kullanıcı Adı: %s, E-Posta: %s" % [ "vigo", "vigo@foo.com" ] # => "Kullanıcı Adı: vigo, E-Posta: vigo@foo.com"
"Merhaba %{username}!" % { :username => 'vigo' } # => "Merhaba vigo!"
```

## String

Keza bu method, `printf`, `sprintf` gibi, **String Format** mantığında çalışır. `"Sayı: %010d" % 2014` örneğinde `%010d` aslında **10** basamaklı, **0** ile pad edilmiş şekilde göster anlamındadır. **2014** 4 basamaklıdır, solunda **6** tane sıfır gelmiştir.

## String \* sayı -> yeni String

String ile sayıyı çarpmak mümkün!

```
"Merhaba!" * 3 # => "Merhaba!Merhaba!Merhaba!"
"Merhaba!" * 0 # => ""
```

## String + String -> yeni String

```
"Merhaba" + " " + "Dünya" # => "Merhaba Dünya"
```

## String << sayı -> String

## String << nesne -> String

```
a = "Merhaba"
a << " dünya" # => "Merhaba dünya"
a # => "Merhaba dünya"
a.concat(33) # => "Merhaba dünya!"
a << 33 # => "Merhaba dünya!!"
```

## String <=> başka string → -1, 0, +1 ya da nil

`<=>` Ruby dünyasında **Spaceship** operatörü olarak geçer. Aynı cins nesneleri karşılaştırmak için kullanılır.

```
"vigo" <=> "vigo" # => 0 # eşit
"vigo" <=> "vig" # => 1 # vigo büyük
"vigo" <=> "vigoo" # => -1 # vigo küçük
"vigo" <=> 3 # => nil # alakasız iki şey
```

`casecmp` ile aynı işi yapar

## String =~ Nesne -> Fixnum ya da nil

## String

```
"Saat tam 4'de buluşalım" =~ /\d/ # => 9

# \d sayı yakaladı ve indeksi döndü

"Saat tam 4'de buluşalım"[9] # => "4"
```

## String içinde hareket

String aslında karakterlerden oluşan bir dizi olduğu için aşağıdaki gibi atraksiyonlar yapmak mümkün.

```
String[indeks] -> yeni string ya da nil
String[başlangıç, uzunluk] -> yeni string ya da nil
String[range] -> yeni string ya da nil
String[regexp] -> yeni string ya da nil
String[regexp, yakalan] -> yeni string ya da nil
String[metinni_bul] -> yeni string ya da nil
```

örnek olarak;

```
m = "Merhaba Dünya"
m[0]          # => "M"   # 0.karakter
m[0, 2]       # => "Me"  # 0'dan itibaren 2 karakter
m[0..4]       # => "Merha" # range, 0'dan 4 dahil
m[-1,]        # => "a"   # son karakter
m[-13..-1]    # => "Merhaba Dünya" # sondan başa
m[15, 1]      # => nil   # olmayan indeks
m[/(?<sesli>[aeiou])/, "sesli"] # => "e" # regexp
m["Merhaba"]  # => "Merhaba" # metni bul
m["vigo"]     # => nil   # olmayan metin
```

aynı işi `slice` ile de yapabilirsiniz.

```
m = "merhaba"
m.slice(2,5) # => "rhaba"
```

gibi...

## Yardımcı Methodlar

Sıkça kullanılanlar arasında; `capitalize` , `center` , `chars` , `chomp` , `chop` , `clear` , `count` , `size` , `length` , `delete` , `ljust` , `rjust` , `reverse` , `upcase` , `downcase` , `swapcase` , `reverse` , `index` , `hex` , `rindex` , `insert` gibi methodlar'dan örnekler ekledim. Her zaman olduğu gibi, hangi method'ların olduğunu görmek için;

```
String.new.methods # => [:<=>, :==, :===, : eql?, :hash, :casecmp, :+, :*, :%, :[], :[]
=, :insert, :length, :size, :bytesize, :empty?, :=~, :match, :succ, :succ!, :next, :ne
xt!, :upto, :index, :rindex, :replace, :clear, :chr, :getbyte, :setbyte, :byteslice, :
scrub, :scrub!, :freeze, :to_i, :to_f, :to_s, :to_str, :inspect, :dump, :upcase, :down
case, :capitalize, :swapcase, :upcase!, :downcase!, :capitalize!, :swapcase!, :hex, :o
ct, :split, :lines, :bytes, :chars, :codepoints, :reverse, :reverse!, :concat, :<<, :p
repend, :crypt, :intern, :to_sym, :ord, :include?, :start_with?, :end_with?, :scan, :l
just, :rjust, :center, :sub, :gsub, :chop, :chomp, :strip, :lstrip, :rstrip, :sub!, :g
sub!, :chop!, :chomp!, :strip!, :lstrip!, :rstrip!, :tr, :tr_s, :delete, :squeeze, :co
unt, :tr!, :tr_s!, :delete!, :squeeze!, :each_line, :each_byte, :each_char, :each_code
point, :sum, :slice, :slice!, :partition, :rpartition, :encoding, :force_encoding, :b,
:valid_encoding?, :ascii_only?, :unpack, :encode, :encode!, :to_r, :to_c, :>, :>=, :<
, :<=, :between?, :nil?, :!~, :class, :singleton_class, :clone, :dup, :taint, :tainted
?, :untaint, :untrust, :untrusted?, :trust, :frozen?, :methods, :singleton_methods, :p
rotected_methods, :private_methods, :public_methods, :instance_variables, :instance_va
riable_get, :instance_variable_set, :instance_variable_defined?, :remove_instance_vari
able, :instance_of?, :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?, :exte
nd, :display, :method, :public_method, :singleton_method, :define_singleton_method, :o
bject_id, :to_enum, :enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec, :__se
nd__, :__id__]
```

## kullanabiliriz!

```
m = "merhaba"

m.capitalize # => "Merhaba"
m # => "merhaba"

m.capitalize! # => "Merhaba" # m'in değeri artık değişti!
m # => "Merhaba"

"vigo".center(12) # => "    vigo    "
"vigo".center(12, "*") # => "****vigo****"

"merhaba".chars # => ["m", "e", "r", "h", "a", "b", "a"]

"merhaba\n".chomp # => "merhaba"
"merhaba dünya".chomp(" dünya") # => "merhaba"

"merhaba vigo".chop # => "merhaba vig"
"merhaba vigo\n".chomp # => "merhaba vigo"

"a".chr # => "a"

x = "Merhaba"
x.clear # => ""

"Merhaba Dünya".count("a") # => 3 # 3 adet a
"Merhaba Dünya".count("ab") # => 4 # a ve b toplam 4 tane
"Merhaba Dünya".count("e") # => 1 # e'den 1 tane

"Merhaba Dünya".delete("e") # => "Mrhaba Dünya"
"Merhaba Dünya".delete("a", "ba") # => "Merhb Düny"
```



```

"MERHABA".downcase # => "merhaba"
"merhaba".upcase # => "MERHABA"
"Merhaba".swapcase # => "mERHABA"

"merhaba".size # => 7
"merhaba".length # => 7

"merhaba".ljust(20) # => "merhaba          "
"merhaba".ljust(20, "*") # => "merhaba*****"
"merhaba".rjust(20) # => "          merhaba"
"merhaba".rjust(20, "*") # => "*****merhaba"

"  merhaba  ".strip # => "merhaba"
"  merhaba".lstrip # => "merhaba"
"merhaba  ".rstrip # => "merhaba"

"merhaba".index("m") # => 0
"merhaba".index("ba") # => 5

"merhaba".rindex("m") # => 0
"merhaba".rindex("h") # => 3

"a".next # => "b"
"abcd".next # => "abce" # d'den sonra e geldi...
"b".succ # => "c"

# baş, ayraç, son
"merhaba".partition("r") # => ["me", "r", "haba"]
"merhaba".partition("a") # => ["merh", "a", "ba"]
"merhaba".partition("x") # => ["", "", "merhaba"]

"merhaba dünya".reverse # => "aynüd abahrem"

"hey seeeeeeeeeeeeeen! alooooooooo".squeeze # => "hey sen! alo"

"Merhaba".dump # => "\"Merhaba\""
"merhaba".getbyte(0) # => 109 # m'in ascii değeri

"0x0f".hex # => 15
"0xffff".hex # => 4095

"merhaba".insert(0, "X") # => "Xmerhaba"
"merhaba".insert(3, "A") # => "merAhaba"
"merhaba".insert(-1, ".") # => "merhaba."

"123".oct # => 83 # Octal'e çevirdi (8'lik)

"A".ord # => 65 # Ascii değeri
"a".ord # => 97 # Ascii değeri

"dünya".prepend("Merhaba ") # => "Merhaba dünya" # öne ekledi

# transform
"merhaba hello".tr("el", "ip") # => "mirhaba hippo" # e=>i, l => p oldu
"ArkAdAşlar nasılsınız?".tr("A", "a") # => "arkadaşlar nasılsınız?"

```

```
# a'dan e'ye kadar X ile transform yap
"merhaba dünya".tr("a-e", "X") # => "mXrhXXX XünyX"
```

## Convert Method'ları

Tip değiştirmek için kullanılır. `to_i`, `to_f`, `to_s`, `to_str`, `to_sym`, `to_r`, `to_c`, `to_enum` method'larına bakalım:

```
"merhaba".to_i # => 0 # integer'a çevirdi
"merhaba".to_f # => 0.0 # float'a çevirdi
"5".to_i # => 5
"1.5".to_f # => 1.5
"merhaba".to_s # => "merhaba" # string
"merhaba".to_str # => "merhaba" # string
"merhaba".to_sym # => :merhaba # symbol
"merhaba".to_r # => (0/1) # Rasyonel sayı
"0.2".to_r # => (1/5) # Rasyonel sayı
"merhaba".to_c # => (0+0i) # Kompleks sayı
"1234".to_c # => (1234+0i)
"merhaba".to_enum # => #<Enumerator: "merhaba":each> # Enumeratör'e çevirdi
```

## Kontrol Method'ları

Method adı `?` ile bitiyor demek, bir kontrol olduğu ve sonucun **Boolean** yani `true` ya da `false` döndüğü anlamında olduğunu söylemiştik.

```
"merhaba".start_with?("m") # => true
"merhaba".start_with?("mer") # => true
"merhaba".start_with?("f") # => false

"merhaba".end_with?("a") # => true
"merhaba".end_with?("haba") # => true
"merhaba".end_with?("zoo") # => false

"merhaba".eql?("Merhaba") # => false
"merhaba".eql?("merhaba") # => true

"merhaba dünya".include?("dünya") # => true

"merhaba".empty? # => false
"".empty? # => true

"kedi".between?("at", "balık") # => false # başlangıç harfi a ve b arasındamı? gibi düşünün
"kedi".between?("fare", "sinek") # => true
```

## Array ve Block ile İlişkili Methodlar

`split` Metni parçalara böler, varsayılan **delimiter** (*ayırıcı*) boşuk karakteridir.

## String

```
"Selam millet nasıl sınız?".split # => ["Selam", "millet", "nasıl", "sınız?"]
"Salam millet-nasıl sınız?".split("-") # => ["Salam millet", "nasıl sınız?"]
"A takımı: 65 B takımı: 120".split(/ +\d+ ?/) # => ["A takımı:", "B takımı:"]
"1,2,3,4,5".split(",") # => ["1", "2", "3", "4", "5"]
```

## each\_byte

```
"merhaba".each_byte {|c| puts c }

# 109 (m)
# 101 (e)
# 114 (r)
# 104 (h)
# 97 (a)
# 98 (b)
# 97 (a)
```

## each\_char

```
"merhaba".each_char {|c| puts c }

# m
# e
# r
# h
# a
# b
# a
```

## each\_line

```
"Merhaba\nDünya\nNasıl sınız?".each_line {|l| puts l }

# Merhaba
# Dünya
# Nasıl sınız?

"Merhaba@@Dünya@@Nasıl sınız?".each_line("@@") {|l| puts l }

# Merhaba@@
# Dünya@@
# Nasıl sınız?
```

## upto

```
"a1".upto("b1"){ |t| puts t }
```

```
# a1
# a2
# a3
# a4
# a5
# a6
# a7
# a8
# a9
# b0
# b1
```

## Pattern Yakalama (Regexp)

Daha kapsamlı olarak **6.Bölüm**'de de değineceğimiz **Regular Expression** konusu, String'lerle çok ilişkili. Hemen ilgili method'lara bakalım

### gsub ve sub

`sub` ile `gsub` arasındaki fark, `sub` ilk bulduğunu işler, `gsub` **Global** anlamındadır.

```
"merhaba dünya, merhaba uzay".sub("merhaba", "olaa") # => "olaa dünya, merhaba uzay"
"merhaba dünya, merhaba uzay".gsub("merhaba", "olaa") # => "olaa dünya, olaa uzay"

"Merhaba Dünya".gsub(/[aeiou]/, "x") # => "Mxrhxbx Dünyx"
"Merhaba Dünya".gsub(/([aeiou])/, '(\1)') # => "M(e)rh(a)b(a) Düny(a)"
"Merhaba dünya, merhaba uzay, merhaba evren".gsub(/((m|M)erhaba)/){|c| c.upcase } # =>
"MERHABA dünya, MERHABA uzay, MERHABA evren"
"Merhaba Dünya".gsub(/(?<sesli_harf>[aeiou])/, '{\k<sesli_harf>}') # => "M{e}rh{a}b{a}
Düny{a}"
"Merhaba Dünya".gsub(/[ea]/, 'e' => 1, 'a' => 'X') # => "M1rhXbX DünyX"
```

### match

```
"merhaba".match("a") # => #<MatchData "a">

"merhaba".match("(a)") # => #<MatchData "a" 1:"a"> # 1 tane yakaladı, (a) ve Array geldi
"merhaba".match("(a)") [0] # => "a" # yakalanan

"merhaba 2014".match(/\d/) # => #<MatchData "2">

"merhaba 2014".match(/(\d)/) # => #<MatchData "2" 1:"2">
"merhaba 2014".match(/(\d+)/) # => #<MatchData "2014" 1:"2014">
"merhaba 2014".match(/(\d+)/) [0] # => "2014"
"merhaba 2014".match(/(\d+)/) [0].to_i # => 2014
```

**scan**

**Match** gibi, metin üzerinde bir nevi arama yapıyoruz:

```
"Merhaba millet!".scan(/\w+/) # => ["Merhaba", "millet"]
"Merhaba millet!".scan(/./) # => ["M", "e", "r", "h", "a", "b", "a", " ", "m", "i", "l", "l", "e", "t", "!"]
"Merhaba millet! Saat 10'da buluşalım".scan(/Saat \d+/) # => ["Saat 10"]
```

# Array

Kompakt, sıralı objeler içeren bir türü taşıyıcı nesnedir Array'ler. Neredeyse içinde her tür Ruby objesini taşıyabilir. (*String, Integer, Fixnum, Hash, Symbol hatta başka Array'ler vs...*)

Arka arkaya sıralanmış kutucuklar düşünün, her kutucuğun bir **index** numarası olduğunu hayal edin. Bu indeksleme işi otomatik olarak oluyor. Diziye her eleman eklediğinizde (*Eğer kendiniz indeks numarası atamadıysanız*) yeni indeks bir artarak devam ediyor. **Zero Indexed** yani ilk eleman **0**.eleman oluyor.

Aynı **String**'deki gibi negatif indeks değerleri ile tersten erişim sağlıyorsunuz. Yani ilk eleman `array[0]` ve son eleman `array[-1]` gibi...

Array oluşturmak için;

```
a = [] # Ya bu şekilde
a.class # => Array

b = Array.new # => Ya da bu şekilde
b.class # => Array
```

kullanabilirsiniz. Keza `a = [1, 2, 3]` dediğinizde de Array'i hem tanımlamış hem de elemanlarını belirlemiş olursunuz.

Array'i **initialize** ederken yani ilk kez oluştururken büyüklüğünü de verebilirsiniz.

```
a = Array.new(5) # içinde 5 eleman taşıyacak Array oluştur
a # => [nil, nil, nil, nil, nil]
```

Hatta, **default** değer bile atarsınız:

```
aylar = Array.new(12, "ay") # 12 eleman olsun ve hepsi "ay" olsun
aylar # => ["ay", "ay", "ay", "ay", "ay", "ay", "ay", "ay", "ay", "ay", "ay", "ay"]
```

Ruby'de her nesnenin bir **ID**'si ve **HASH** değeri vardır.

```
[1, 2, 3].hash # => 3384159031637530117
[1, 2, 3].__id__ # => 70147646473880
```

**Block** kabul ettiği için;

## Array

```
dizi = Array.new(10) { |eleman| eleman = eleman * 4 }
dizi # => [0, 4, 8, 12, 16, 20, 24, 28, 32, 36]

# ya da aynı kodu

dizi = Array.new(10) do |eleman|
  eleman = eleman * 4
end
dizi # => [0, 4, 8, 12, 16, 20, 24, 28, 32, 36]
```

şeklinde de kullanabilirsiniz. Neticede Array'in **initialize** method'una parametre geçmiş oluyoruz:

```
aylar = Array.[]( "oca", "şub", "mar", "nis", "may", "haz", "tem", "ağu", "eyl", "eki",
"kas", "ara")
aylar # => ["oca", "şub", "mar", "nis", "may", "haz", "tem", "ağu", "eyl", "eki", "kas",
"ara"]
```

Yani: `aylar = Array.[](param, param, param)` gibi.

Başka nasıl üretiriz? Sayılardan oluşan bir dizi lazım olsa; Örneğin **1984** ile **2000** arasındaki yıllar lazım olsa;

```
years = Array(1984..2000)
years # => [1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000]
```

Bir Array içinde farklı türden elemanlar olabilir;

```
a = ["Hello", :word, 2014, 3.14] # içinde String, Symbol, Fixnum ve Float var!
a # => ["Hello", :word, 2014, 3.14]
```

Array içindeki elemanlar sıralı bir şekilde dururlar. Bu sıraya **Index** denir. **0**'dan başlar. Yani ilk eleman demek Array'in 0. elemanı demektir. İsteğimiz elemanı almak için ya

`Array[index]` ya da `Array.fetch(index)` yöntemlerini kullanabiliriz.

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a[0] # => "Uğur"
a.fetch(0) # => "Uğur"

a[4] # => nil
a.fetch(4, "Hatalı Index") # => "Hatalı Index"

a = [1, 2, 3, 4] # İlk N elemanı al
a.take(2) # => [1, 2]

a.drop(2) # => [3, 4] # take'in tersi... İlk 2 haricini al
```

Örnekte `a[4]` dediğimiz zaman, olmayan index'li elemanı almaya çalışıyor ve eleman olmadığı için `nil` geri alıyoruz. `fetch` kullanarak hata kontrolü de yapmış oluyoruz. `nil` yerine belirlediğimiz hata mesajını dönmüş oluyoruz.

`values_at` method'u ile ilgili index ya da index'lerdeki elemanları alabiliriz, keza `at` de benzer işe yarar.

```
isimler = ["Uğur", "Ömer", "Yeşim", "Ezel", "Eren"]
isimler.values_at(0)      # => ["Uğur"]
isimler.values_at(1, 2)   # => ["Ömer", "Yeşim"]

["a", "b", "c", "d", "e"].at(1) # => "b"
["a", "b", "c", "d", "e"].at(-1) # => "e"
```

`rindex` ile sağdan hizalı index'e göre elemana ulaşıyoruz:

```
a = [ "a", "b", "b", "b", "c" ]

a.rindex("b") # => 3 # 3 tane b var, en sağdaki son b'yi verdi!
a.rindex("z") # => nil
```

Keza "acaba Array'in ne gibi method'ları var?" dersek; hemen `methods` özelliği ile bakabiliriz. Karıştırmamamız gereken önemli bir konu var. Detayını **Class** konusunda göreceğiz ama yeri gelmişken, Array'in **Class Method**'ları ve **Instance Method**'ları var.

`Array.methods` dediğimizde Kernel'dan gelen Array objesinin yani **Class**'ının method'larını görürüz. Eğer `Array.new.methods` dersek, Array'den türettiğimiz **instance**'a ait method'ları görürüz.

Yani `a = []` dediğimizde, aslında **Array**'den bir **instance** çıkartmış oluyoruz. Az önce `Array.[](...)` olarak yaptığımız şey de aslında Class method'u çağırmak.

## Class Method'ları



```
Array.methods # => [:[], :try_convert, :allocate, :new, :superclass, :freeze, :===, :==, :<=>, :<, :<=, :>, :>=, :to_s, :inspect, :included_modules, :include?, :name, :ancestors, :instance_methods, :public_instance_methods, :protected_instance_methods, :private_instance_methods, :constants, :const_get, :const_set, :const_defined?, :const_missing, :class_variables, :remove_class_variable, :class_variable_get, :class_variable_set, :class_variable_defined?, :public_constant, :private_constant, :singleton_class?, :include, :prepend, :module_exec, :class_exec, :module_eval, :class_eval, :method_defined?, :public_method_defined?, :private_method_defined?, :protected_method_defined?, :public_class_method, :private_class_method, :autoload, :autoload?, :instance_method, :public_instance_method, :nil?, :=~, :!~, :eql?, :hash, :class, :singleton_class, :clone, :dup, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :frozen?, :methods, :singleton_methods, :protected_methods, :private_methods, :public_methods, :instance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_defined?, :remove_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?, :extend, :display, :method, :public_method, :singleton_method, :define_singleton_method, :object_id, :to_enum, :enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]
```

Bu method'ların bir kısmı **Enumerable** sınıfından gelen method'lardır. Ruby, **Module** yapısı kullandığı için ortak kullanılan method'lar modül eklemelerinden gelmektedir. **Class** konusunda detayları göreceğiz.

Bu kısımdan en fazla kullanacağımız `[]` ve `new` method'ları olacaktır.

## Instance Method'ları

En çok kullanacağımız method'larsa;

```
Array.new.methods # => [:inspect, :to_s, :to_a, :to_h, :to_ary, :frozen?, :==, :eql?, :hash, :[], :[]=, :at, :fetch, :first, :last, :concat, :<<, :push, :pop, :shift, :unshift, :insert, :each, :each_index, :reverse_each, :length, :size, :empty?, :find_index, :index, :rindex, :join, :reverse, :reverse!, :rotate, :rotate!, :sort, :sort!, :sort_by!, :collect, :collect!, :map, :map!, :select, :select!, :keep_if, :values_at, :delete, :delete_at, :delete_if, :reject, :reject!, :zip, :transpose, :replace, :clear, :fill, :include?, :<=>, :slice, :slice!, :assoc, :rassoc, :+, :*, :-, :&, :|, :uniq, :uniq!, :compact, :compact!, :flatten, :flatten!, :count, :shuffle!, :shuffle, :sample, :cycle, :permutation, :combination, :repeated_permutation, :repeated_combination, :product, :take, :take_while, :drop, :drop_while, :bsearch, :pack, :entries, :sort_by, :grep, :find, :detect, :find_all, :flat_map, :collect_concat, :inject, :reduce, :partition, :group_by, :all?, :any?, :one?, :none?, :min, :max, :minmax, :min_by, :max_by, :minmax_by, :member?, :each_with_index, :each_entry, :each_slice, :each_cons, :each_with_object, :chunk, :slice_before, :lazy, :nil?, :===, :=~, :!~, :class, :singleton_class, :clone, :dup, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :freeze, :methods, :singleton_methods, :protected_methods, :private_methods, :public_methods, :instance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_defined?, :remove_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?, :extend, :display, :method, :public_method, :singleton_method, :define_singleton_method, :object_id, :to_enum, :enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]
```

Aynı **String**'deki gibi, şu Array'in bir röntgenini çekelim:

```
Array.class # => Class
Array.class.superclass # => Module
Array.class.superclass.superclass # => Object
Array.class.superclass.superclass.superclass # => BasicObject
Array.class.superclass.superclass.superclass.superclass # => nil
```

Array'in bir üst objesi ne? **Module** Yine **Class** konusunda göreceğiz diyeceğim ve siz bana kızacaksınız :) Ruby'de bir Class en fazla başka bir Class'dan türeyebilir. Örneğin Python'da bir Class N TANE Class'tan **inherit** olabilir (*Miras alabilir, türeyebilir*)

Ruby, bu sorunu **Module** yapısıyla çözüyor. Bu mantıkla aslında ortaklaşa kullanılan Kernel modülleri yardımıyla, ortak kullanılacak method'lar bu modüllerin **Include** edilmesiyle ilgili yerlere dağıtılıyor.

Acaba Array'de hangi modüller var?

```
Array.included_modules # => [Enumerable, Kernel]
```

Bu bakımdan Array, Hash gibi nesnelerde benzer ortak method'lar görmek mümkün.

## length veya count

Array'in boyu / içinde kaç eleman olduğu ile ilgili bilgiyi almak için kullanılır.

```
[1, 2, 3, 4].length # => 4
[1, 2, 3, 4].count  # => 4
```

## empty?

Array acaba boşmu? İçinde hiç eleman var mı?

```
[1, 2, 3, 4].empty? # => false
[].empty?           # => true
```

## eq?, ==, ===

Eşitlik kontrolü içindir. Eğer karşılığı aynı cinsse ve birebir aynı elemanlara sahipse `true` döner.

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]

a.eq?(["Yeşim", "Ezel", "Ömer", "Uğur"]) # => false
a.eq?([])                                # => false
a.eq?(["Uğur", "Yeşim", "Ezel", "Ömer"]) # => true
```

**== Generic Equality** yani genel eşitlik kontrolü yani hepimizin bildiği kontrol, **===** ise **Case Equality** yani `a === b` ifadesinde **a**, **b**'nin **subseti** mi? demek olur. Örnek verelim:

```
5.class.superclass # => Integer
Integer === 5      # => true
# 5, Integer subsetinde...

Integer.class # => Class
Integer.class.superclass # => Module
Integer.class.superclass.superclass # => Object
Integer.class.superclass.superclass.superclass # => BasicObject
Integer.class.superclass.superclass.superclass.superclass # => nil

# Integer, 5'in subsetinde değil.
5 === Integer      # => false
```

### include? ve member?

Acaba verdiğim eleman Array'in içinde mi? Verdiğim eleman bu dizinin üyesi mi?

```
[1, 2, 3, 4].include?(3)          # => true

[1, 2, 3, 4].member?(1) # => true
[1, 2, 3, 4].member?(5) # => false

["Uğur", "Ezel", "Yeşim"].include?("Uğur") # => true
["Uğur", "Ezel", "Yeşim"].include?("Ömer") # => false
```

### array & başka\_bir\_array

İki dizide de kullanın ortak elemanları alır yeni Array döner:

```
a = [1, 2, 3, 4]
b = [3, 1, 10, 22]
a & b # => [1, 3]
```

### array \* int [ya da] array \* str

```
a = ["a", "b", "c"]
a * 5 # => ["a", "b", "c", "a", "b", "c", "a", "b", "c", "a", "b", "c", "a", "b", "c"]

a * "-vigo-" # => "a-vigo-b-vigo-c"
```

\* çarparak **3** elemanlı `a` Array'inden sanki birleştirilmiş **15** elemanlı yeni bir Array oluşturduk. **String** ile çarpınca da aslında `join` methodu ile Array'den String yaptık ve birleştirici olarak **-vigo-** metni kullandık!

### array + başka\_array

İki Array'i toplar ve yeni Array döner:

```
a = ["Uğur", "Yeşim", "Ezel"]  
b = ["Ömer"]  
  
a + b # => ["Uğur", "Yeşim", "Ezel", "Ömer"]
```

### array - başka\_array

Array'ler arasındaki farkı Array olarak bulmak:

```
a = ["Uğur", "Yeşim", "Ezel"]  
b = ["Uğur", "Yeşim"]  
  
a - b # => ["Ezel"] # a'da olab b elemanları kayboldu
```

### array | başka\_array

İki Array'i **unique** (*tekil*) elemanlar olarak birleştirdi. Aynı eleman varsa bunlardan birini aldı:

```
a = ["Uğur", "Yeşim", "Ezel"]  
b = ["Uğur", "Ömer"]  
  
a | b # => ["Uğur", "Yeşim", "Ezel", "Ömer"]
```

### array << nesne ya da push

Array'in sonuna eleman eklemek için kullanılır.

```
a = ["Uğur", "Yeşim", "Ezel"]  
a << "Ömer" # => ["Uğur", "Yeşim", "Ezel", "Ömer"]  
a.push("Eren") # => ["Uğur", "Yeşim", "Ezel", "Ömer", "Eren"]
```

Keza zincirleme çağrı da yapabilirsiniz:

```
a.push("Tunç").push("Suat") # => ["Uğur", "Yeşim", "Ezel", "Ömer", "Eren", "Tunç", "Suat"]
```

### concat

Array sonuna Array eklemek için kullanılır.

```
a = [1, 2, 3]  
a.concat([4, 5, 6])  
a # => [1, 2, 3, 4, 5, 6]
```

### join

Array elemanlarını birleştirip **String**'e çevirmeye yarar. Eğer parametre verirse aradaki birleştiriciyi de belirlemiş oluruz.

```
["Commodore 64", "Amiga", "Sinclair", "Amstrad"].join      # => "Commodore 64AmigaSinclairAmstrad"
["Commodore 64", "Amiga", "Sinclair", "Amstrad"].join(" , ") # => "Commodore 64 , Amiga , Sinclair , Amstrad"
```

### unshift

Array'in başına eleman eklemek için kullanılır.

```
a = ["Uğur", "Yeşim", "Ezel"]
a.unshift("Ömer") # => ["Ömer", "Uğur", "Yeşim", "Ezel"]
```

### insert

Array'de istediğiniz bir noktaya eleman eklemek için kullanılır. İlk parametre **index** diğer parametre/ler de eklenecek eleman/lar.

```
a = ["Uğur", "Yeşim", "Ezel"]
a.insert(1, "Ömer") # => ["Uğur", "Ömer", "Yeşim", "Ezel"]
a.insert(1, "Ahmet", "Ece", "Eren") # => ["Uğur", "Ahmet", "Ece", "Eren", "Ömer", "Yeşim", "Ezel"]
```

### replace

Array'in içini, diğer Array'le değiştirir. Aslında Array'i başka bir Array'e eşitlemek gibidir. Eleman sayısının eşit olup olmaması hiç önemli değildir.

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.replace(["Foo", "Bar"]) # => ["Foo", "Bar"]
a                        # => ["Foo", "Bar"]
```

### array <=> başka\_array

**Spaceship** operatöründen bahsetmiştik. Array'ler arasında karşılaştırma yapmayı sağlar.

```
[1, 2, 3, 4] <=> [1, 2, 3, 4] # => 0 # Eşit
[1, 2, 3, 4] <=> [1, 2, 3]    # => 1 # İlk değer büyük
[1, 2, 3] <=> [1, 2, 3, 4]   # => -1 # İlk değer küçük
```

### pop, shift, delete, delete\_at, delete\_if

Son elemanı çıkartmak için **pop** ilk elemanı çıkartmak için **shift** kullanılır. Herhangibir elemanı çıkartmak için **delete**, belirli bir index'deki elemanı çıkartmak için **delete\_at** kullanılır.

## Array

```
a = ["Uğur", "Ömer", "Yeşim", "Ezel", "Eren"]
a.pop          # => "Eren"
a              # => ["Uğur", "Ömer", "Yeşim", "Ezel"]
a.shift       # => "Uğur"
a              # => ["Ömer", "Yeşim", "Ezel"]
a.delete("Ömer") # => "Ömer"
a              # => ["Yeşim", "Ezel"]
a.delete_at(1)  # => "Ezel"
a              # => ["Yeşim"]

# not 50'den küçükse sil :)
notlar = [40, 45, 53, 70, 99, 65]
notlar.delete_if { |notu| notu < 50 } # => [53, 70, 99, 65]
```

**pop**'a parametre geçerse **son n** taneyi uçurmuş oluruz:

```
a = ["Uğur", "Ömer", "Yeşim", "Ezel", "Eren"]
a.pop(2) # => ["Ezel", "Eren"]
a        # => ["Uğur", "Ömer", "Yeşim"]
```

## compact ve uniq

**nil** elemanları uçurmak için **compact**, duplike elemanları tekil hale getirmek için **uniq** kullanılır.

```
["a", 1, nil, 2, nil, "b", 1, "a"].compact # => ["a", 1, 2, "b", 1, "a"]
["a", 1, nil, 2, nil, "b", 1, "a"].uniq     # => ["a", 1, nil, 2, "b"]
["a", 1, nil, 2, nil, "b", 1, "a"].compact.uniq # => ["a", 1, 2, "b"]
```

## array == başka\_array

İki Array nitelik ve nicelik olarak birbirine eşit mi?

```
[1, 2, 3, 4] == [1, 2, 3, 4] # => true
[1, 2, 3, 4] == ["1", 2, 3, 4] # => false
[1, 2, 3, 4] == [1, 2, 3]     # => false
```

## assoc ve rassoc

Elemanları Array olan bir Array içinde, ilk değere göre yakalama yapmaya yarar.

## Array

```
a = ["renkler", "kırmızı", "sarı", "mavi"]
b = ["harfler", "a", "b", "c"]
c = "foo"

t = [a, b, c]
t # => [["renkler", "kırmızı", "sarı", "mavi"], ["harfler", "a", "b", "c"], "foo"]
t.assoc("renkler") # => ["renkler", "kırmızı", "sarı", "mavi"]
t.assoc("foo") # => nil

t.rassoc("kırmızı") # => ["renkler", "kırmızı", "sarı", "mavi"]
```

`rassoc` ise ikinci elemanına bakar, yani "renkler" yerine "kırmızı" kullanabiliriz:

### **slice(başlangıç, boy) ya da slice(aralık)**

Array içinden kesip başka bir Array oluşturmak için kullanılır. **başlangıç** indeks'indeki eleman dahil olmak üzere, boy ya da aralık kadarını kes.

```
[1, 2, 3, 4].slice(0, 2) # => [1, 2] # 0.dan itibaren 2 tane
[1, 2, 3, 4].slice(2..4) # => [3, 4] # 2.den itibaren 2 tane
```

### **first ve last**

Adından da anlaşılacağı gibi, Array'in ilk ve son elemanları için kullanılır:

```
a = [1, 2, 3, 4, 5]
a.first # => 1
a.last # => 5
```

Eğer parametre geçerse, **ilk n** ya da **son n** elemanları alabiliriz:

```
a = [1, 2, 3, 4, 5]
a.first(2) # => [1, 2]
a.last(2) # => [4, 5]
```

### **find (detect), find\_all, index, find\_index**

`find` ile blok içinde koşula uyan ilk Array elemanını, `find_all` ile tümünü alırız:

```
["Uğur", "Yeşim", "Ezel", "Ömer"].find { |n| n.length > 3 } # => "Uğur"
["Uğur", "Yeşim", "Ezel", "Ömer"].find_all { |n| n.length > 3 } # => ["Uğur", "Yeşim", "Ezel", "Ömer"]
```

`detect` ile `find` aynı işi yapar.

`index` , `find_index` ile elemanın index'ini buluruz:

## Array

```
["a", "b", "c", "d", "e"].index("e") # => 4
["Uğur", "Yeşim", "Ezel", "Ömer"].index("Ezel") # => 2
["Uğur", "Yeşim", "Ezel", "Ömer"].find_index("Ezel") # => 2
```

### clear

Array'i temizlemek için kullanılır :)

```
a = [1, 2, 3]
a.clear # => []
a # => []
```

### reverse

Array'i terse çevir.

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

### sample

Array'den **random** olarak eleman almaya yarar. Eğer parametre geçilirse geçilen adet kadar random eleman döner.

```
a = [1, 2, 3, 4, 5]
a.sample # => 3
a.sample(3) # => [5, 1, 3]
```

### shuffle

Array'in içindeki elemanların index'lerini karıştırır :)

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

### sort

Array içindeki elemanları `<=>` mantığıyla sıralar.

```
a = [1, 4, 2, 3, 11, 5]
a.sort # => [1, 2, 3, 4, 5, 11]

b = ["a", "c", "b", "z", "d"]
b.sort # => ["a", "b", "c", "d", "z"]
```

### fill



## Array

Array'in içini ilgili değerle doldurmak için kullanılır. İşlem sonucunda orijinal Array'in değeri değişir. Yani ne ile **fill** ettiyseniz Array artık o değerlerdedir.

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.fill("x")      # => ["x", "x", "x", "x"] # tüm elemanları x yaptı
a                # => ["x", "x", "x", "x"] # artık a'nın yeni değeri bu

a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.fill("y", 2)   # => ["Uğur", "Yeşim", "y", "y"] # 2.den itibaren y ile doldur

a = ["Uğur", "Yeşim", "Ezel", "Ömer"] # 2.den itibaren 1 tane doldur
a.fill("z", 2, 1) # => ["Uğur", "Yeşim", "z", "Ömer"]
```

Keza;

```
a = [1, 2, 3, 4, 5]
a.fill { |i| i * 5 } # => [0, 5, 10, 15, 20]
a                  # => [0, 5, 10, 15, 20]
```

şeklinde de kullanılır.

### flatten

Array içinde Array elemanları varsa, tek harekette bunları düz tek bir Array haline getirebiliriz.

```
[1, 2, ["a", "b", :c], [66, [5.5, 3.1]]].flatten # => [1, 2, "a", "b", :c, 66, 5.5, 3.1]
```

### rotate

Array elemanları kendi içinde kaydırır.

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

a.rotate # => [2, 3, 4, 5, 1] # 1 kaydırdı
a.rotate(2) # => [3, 4, 5, 1, 2] # 2 kaydırdı, ilk 2 elemanı sona koydu!
```

Varsayılan değer **1**'dir.

### zip

```
a = [ 4, 5, 6 ]
b = [ 7, 8, 9 ]

[1, 2, 3].zip(a, b) # => [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
[1, 2].zip(a, b)   # => [[1, 4, 7], [2, 5, 8]]
a.zip([1, 2], [8]) # => [[4, 1, 8], [5, 2, nil], [6, nil, nil]]
```

`[1, 2, 3].zip(a, b)` işlemini yaparken, önce 0. elemanı yani **1**'i aldı, sonra **a**'nın 0. elemanını aldı, sonra da **b**'nin 0. elemanını aldı ve paketledi : `[1, 4, 7]` aynı işi 1. ve 2. elemanlar için yaptı.

`[1, 2].zip(a, b)` yaparken, Array boyları eşit olmadığı için `[1, 2]` sadece 2 elemanlı olduğu için bu işlemi 0. ve 1. elemanlar için yaptı.

Son örnekte index'e karşılık gelmediği için elemanlar `nil` geldi!

## transpose

Array içindeki Array'leri satır gibi düşünüp bunları sütuna çeviriyor gibi algılayabilirsiniz.

```
a = [[1, 2], [3, 4], [5, 6]]
a.transpose # => [[1, 3, 5], [2, 4, 6]]

# [
#   [1, 2],
#   [3, 4],
#   [5, 6]
# ]
# -> [1, 3, 5], [2, 4, 5]
#
```

# Tip Çeviricileri

`to_a` ve `to_ary` kendisini döner, asıl görevi eğer alt sınıftan çağrılmışsa, yani Array'den türeyen başka bir Class'da kullanıldığında direk Array'e dönüştürür.

```
["a", 1, "b", 2].to_a # => ["a", 1, "b", 2]
["a", 1, "b", 2].to_ary # => ["a", 1, "b", 2]
[["a", 1], ["b", 2]].to_h # => {"a"=>1, "b"=>2}
["a", 1, "b", 2].to_s # => "["a", 1, "b", 2]"
["a", 1, "b", 2].inspect # => "["a", 1, "b", 2]"
```

`entries` de aynen `to_a` gibi çalışır:

```
(1..3) # => 1..3
(1..3).entries # => [1, 2, 3]
(1..3).to_a # => [1, 2, 3]
```

## grep

Aslında bu konuları **Regular Expressions**'da göreceğiz ama yeri gelmişken hızla değinelim. Array içinde elemanları **Regex** koşullarına göre filtreleyebiliyoruz:

## Array

```
(1..10).to_a      # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 2'den 5'e kadar (5 dahil)
(1..10).grep 2..5  # => [2, 3, 4, 5]

# sadece .com olan elemanları al
["a", "http://example.com", "b", "foo", "http://webbox.io"].grep(/^http.+\.com/) # =>
["http://example.com"]
```

## pack

Array'in içeriğini verilen direktife göre **Binary String** haline getirir. Uzunca bir [direktif listesi](#) var.

```
# A: String olarak işle, space karakteri kullan
# 5: Uzunluğu 5 karakter olsun
["a", "b", "c"].pack("A5A5A5")      # => "a   b   c   "

# Uzunluğu 5'ten büyük olan kesintiye uğradı
["ali", "burak", "cengiz"].pack("A5A5A5") # => "ali  burakcengi"

# a: String olarak işle, null yani \x00 karakteri kullan
["a", "b", "c"].pack("a3a3a3")      # => "a\x00\x00b\x00\x00c\x00\x00"
```

# İterasyon ve Block Kullanımı

**collect / map { |eleman| blok } → yeni\_array**

Blok içinde gelen kodu her elemana uygular, yeni Array döner:

```
a = [1, 2, 3, 4, 5]
a.collect { |i| i * 2 }      # => [2, 4, 6, 8, 10]
a.collect { |i| "sayı #{i}" } # => ["sayı 1", "sayı 2", "sayı 3", "sayı 4", "sayı 5"]
```

**map** de aynı işi yapar:

```
["Uğur", "Yeşim", "Ezel", "Ömer"].map { |isim| "İsim: #{isim}" } # => ["İsim: Uğur", "İsim: Yeşim", "İsim: Ezel", "İsim: Ömer"]
["Uğur", "Yeşim", "Ezel", "Ömer"].collect { |isim| "İsim: #{isim}" } # => ["İsim: Uğur", "İsim: Yeşim", "İsim: Ezel", "İsim: Ömer"]
```

## select

Blok içinden gelen ifadenin **true** / **false** olmasına göre filtre yapar ve yeni Array döner:

## Array

```
[1, 2, 3, 10, 15, 20].select { |n| n % 2 == 0 } # => [2, 10, 20] # 2'ye tam bölünenler  
[1, 2, "3", "ali", 15, 20].select { |n| n.is_a?(Fixnum) } # => [1, 2, 15, 20] # sadece  
sayılar
```

### reject

`select` in tersidir.

```
[1, 2, 3, 10, 15, 20].reject { |n| n % 2 == 0 } # => [1, 3, 15] # 2'ye tam bölünenleri  
at  
[1, 2, "3", "ali", 15, 20].reject { |n| n.is_a?(Fixnum) } # => ["3", "ali"] # Sayı ola  
nları at
```

### keep\_if

Blok içindeki ifade'den sadece `false` dönenleri atar ve Array'in orjinal değerini bozar, değiştirir.

```
a = [1, 2, 3, 10, 15, 20]  
a.keep_if { |n| n % 2 == 0 } # => [2, 10, 20] # 2'ye bölünemeyenler false geldiği için  
düşüldüler.  
a # => [2, 10, 20] # a artık bu!
```

### combination(n) { |c| blok } → array

Matematikteki kombinasyon işlemidir. 1, 2 ve 3 sayılarının 2'li kombinasyonu:

```
a = [1, 2, 3]  
a.combination(1).to_a # => [[1], [2], [3]]  
a.combination(2).to_a # => [[1, 2], [1, 3], [2, 3]]  
  
a.combination(2) { |c| puts "Olasılıklar: #{c.join(" ve ")}" }  
  
# Olasılıklar: 1 ve 2  
# Olasılıklar: 1 ve 3  
# Olasılıklar: 2 ve 3
```

### permutation

Aynı kombinasyon gibi, matematikteki permutasyon işlemidir.

```
[1, 2, 3].permutation.to_a # => [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2],  
[3, 2, 1]]
```

Eğer parametre geçerse kaçlı permutasyon olduğunu belirtiriz:

```
[1, 2, 3].permutation(1).to_a # => [[1], [2], [3]]
[1, 2, 3].permutation(2).to_a # => [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

## repeated\_combination, repeated\_permutation

`combination` ile `repeated_combination` arasındaki farkı örnekle görelim:

```
[1, 2, 3].combination(1).to_a # => [[1], [2], [3]]
[1, 2, 3].repeated_combination(1).to_a # => [[1], [2], [3]]

[1, 2, 3].combination(2).to_a # => [[1, 2], [1, 3], [2, 3]]
[1, 2, 3].repeated_combination(2).to_a # => [[1, 1], [1, 2], [1, 3], [2, 2], [2, 3], [3, 3]]
```

`combination` olası tekil sonucu, `repeated_combination` pas edilen sayıya göre tekrar da edebilen sonucu döner. Aynısı `repeated_permutation` için de geçerlidir:

```
[1, 2, 3].permutation(1).to_a # => [[1], [2], [3]]
[1, 2, 3].repeated_permutation(1).to_a # => [[1], [2], [3]]

[1, 2, 3].permutation(2).to_a # => [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
[1, 2, 3].repeated_permutation(2).to_a # => [[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
```

## product

Array ve argüman olarak geçilecek diğer Array/lerin elemanlarıyla oluşabilecek tüm alternatifleri üretmenizi sağlar.

```
[1, 2, 3].product # => [[1], [2], [3]]
[1, 2, 3].product([4, 5]) # => [[1, 4], [1, 5], [2, 4], [2, 5], [3, 4], [3, 5]]
[1, 2, 3].product([7, 8, 9]) # => [[1, 7], [1, 8], [1, 9], [2, 7], [2, 8], [2, 9], [3, 7], [3, 8], [3, 9]]
[1, 2, 3].product(["a", "b"], ["x", "y"]) # => [[1, "a", "x"], [1, "a", "y"], [1, "b", "x"], [1, "b", "y"], [2, "a", "x"], [2, "a", "y"], [2, "b", "x"], [2, "b", "y"], [3, "a", "x"], [3, "a", "y"], [3, "b", "x"], [3, "b", "y"]]
```

## count

Az önce method olarak işlediğimiz `count` ile başka ilginç işler de yapabiliyoruz:

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

a.count # => 5 # eleman sayısı
a.count(2) # => 2 # kaç tane 2 var?
a.count { |n| n % 2 == 0 } # => 3 # kaç tane 2'ye tam bölünen var?
```

## cycle(n=nil) { |obje| blok } → nil

Pas edilen blok'u **n** defa tekrar eder.

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

a.cycle(2).to_a # => [1, 2, 3, 1, 2, 3] # 2 defa
a.cycle(4).to_a # => [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3] # 3defa
a.cycle(2) { |o| puts "Sayı #{o}" }

# Sayı 1
# Sayı 2
# Sayı 3
# Sayı 1
# Sayı 2
# Sayı 3
```

Eğer `[1, 2, 3].cycle { |i| puts i }` gibi bir işlem yaparsanız, default olarak `nil` geçmiş olursun ve bu sonsuz döngüye girer, sonsuza kadar 1, 2, 3, 1, 2, 3 .... şeklinde devam eder!

### **drop\_while { |array| blok } → yeni array**

`delete_if` ile aynı işi yapar.

```
notlar = [40, 45, 53, 70, 99, 65]
notlar.drop_while { |notu| notu < 50 } # => [53, 70, 99, 65]
```

Koşula göre Array'den atar gibi düşünebilirsiniz. Not 50'den küçükse bırak.

### **take\_while**

Aynı `drop_while` gibi çalışır ama tersini yapar:

```
notlar = [40, 45, 53, 70, 99, 65]
notlar.take_while { |notu| notu < 50 } # => [40, 45]
```

Koşula göre Array'e ekler gibi düşünebilirsiniz. Not 50'den küçükse sepete ekle! :)

### **each, each\_index, each\_with\_index, each\_slice, each\_cons, each\_with\_object, reverse\_each**

Array ve hatta Enumerator'lerin can damarıdır. Ruby yazarken siz de göreceksiniz `each` en sık kullandığınız iterasyon (*yineleme / tekrarlama*) yöntemi olacak.

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]

a.each # => #<Enumerator: ["Uğur", "Yeşim", "Ezel", "Ömer"]:each>
a.each { |isim| puts "İsim: #{isim}" }

# İsim: Uğur
# İsim: Yeşim
# İsim: Ezel
# İsim: Ömer
```

Array ve içinde dolaşılabilir her nesnede işe yarar. Birde bunun **index**'li hali var;

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]

a.each_index # => #<Enumerator: ["Uğur", "Yeşim", "Ezel", "Ömer"]:each_index>
a.each_index.to_a # => [0, 1, 2, 3]
a.each_index { |i| puts "Index: #{i}, Değeri: #{a[i]}" }

# Index: 0, Değeri: Uğur
# Index: 1, Değeri: Yeşim
# Index: 2, Değeri: Ezel
# Index: 3, Değeri: Ömer
```

ya da bu işi;

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.each_with_index { |eleman, index| puts "index: #{index}, eleman: #{eleman}" }

# index: 0, eleman: Uğur
# index: 1, eleman: Yeşim
# index: 2, eleman: Ezel
# index: 3, eleman: Ömer
```

`each_slice` da Array'i gruplamak, parçalara ayırmak içindir. Geçilen parametre bu işe yarar:

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.each_slice(2) # => #<Enumerator: ["Uğur", "Yeşim", "Ezel", "Ömer"]:each_slice(2)>
a.each_slice(2).to_a # => [["Uğur", "Yeşim"], ["Ezel", "Ömer"]]
a.each_slice(2) { |ikili_grup| puts "#{ikili_grup}" }

# ["Uğur", "Yeşim"]
# ["Ezel", "Ömer"]
```

`each_cons` ise slice gibi ama mutlaka belirtilen miktarda parça üretir.

## Array

```
# 3'lü üret
[1, 2, 3, 4, 5, 6].each_cons(3).to_a # => [[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]

# 4'lü üret
[1, 2, 3, 4, 5, 6].each_cons(4).to_a # => [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

`each_with_object` de ise, iterasyona girerken bir nesne pas edip, o nesneyi doldurabilirsiniz.

```
[1, 2, 3, 4].each_with_object([]) { |number, given_object|
  given_object << number * 2
} # => [2, 4, 6, 8]
```

`number` Array'den gelen eleman (1, 2, 3, 4 gibi), `given_object` ise `each_with_object([])` method'da geçtiğimiz boş Array `[]`.

`reverse_each` aslında Array'i otomatik olarak ters çevirir yani **reverse** eder ve içinde dolaşmanızı sağlar:

```
computers = ["Commodore 64", "Amiga", "Sinclair", "Amstrad"]
computers.reverse_each # => #<Enumerator: ["Commodore 64", "Amiga", "Sinclair", "Amstrad"]:reverse_each>
computers.reverse_each.to_a # => ["Amstrad", "Sinclair", "Amiga", "Commodore 64"]

computers.reverse_each { |c| puts "Bilgisayar: #{c}" }

# Bilgisayar: Amstrad
# Bilgisayar: Sinclair
# Bilgisayar: Amiga
# Bilgisayar: Commodore 64
```

## find\_index

İndeks'i ararken blok işleyebiliriz:

```
computers = ["Commodore 64", "Amiga", "Sinclair", "Amstrad"]
computers.find_index { |c| c == "Amstrad" } # => 3
```

## freeze ve frozen?

Array'i kitlemek için kullanılır. Yani **freeze** (dondurulmuş) bir Array'e yeni eleman eklenemez. Keza `Array#sort` esnasında da otomatik olarak **freeze** olur sort bitince buz çözülür!

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.freeze
a << "Fazilet" # Yeni isim eklemek mümkün değildir!
```



```
RuntimeError: can't modify frozen Array
```

Array'de buzlanma var mı yok mu anlamak için **frozen?** kullanırız:

```
a = ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.freeze # => ["Uğur", "Yeşim", "Ezel", "Ömer"]
a.frozen? # => true
```

## min, max, minmax, min\_by, max\_by ve minmax\_by

`min` ve `max` ile Array elemanlarından en küçük/büyük değeri alırız:

```
a = [6, 1, 8, 4, 11]
a.min # => 1
a.max # => 11
```

Peki sayı yerine metinler olsa ne olacaktı?

```
m = ["a", "ab", "abc", "abcd"]
m.min # => "a"
m.max # => "abcd"
```

peki, `m` Array'i şöyle olsaydı : `m = ["a", "ab", "abc", "abcd", "111111111"]` sonuç ne olurdu?

```
m = ["a", "ab", "abc", "abcd", "111111111"]

m.min # => "111111111" # ?
m.max # => "abcd"
```

Önce **Comparable** mı diyer bakılır, sayılar için çalışan bu yöntem, **String** de `a <=> b` karşılaştırmasına girer ve **Lexicological** karşılaştırma yapar. "111111111" karakter sayısı olarak diğerlerine göre çok olmasına rağmen, `min` değer olarak gelir. Eğer karakter sayına göre karşılaştırma yapmak gerekiyorsa;

```
m = ["a", "ab", "abc", "abcd", "111111111"]
m.min { |a, b| a.length <=> b.length } # => "a"
m.max                                     # => "abcd"
```

Şeklinde yapmak gerekir. Blok kullanabildiğimiz için aynı iş `max` için de geçerlidir. Ya da bu işleri yapabilmek için `min_by` ve `max_by` kullanabiliriz:

## Array

```
m = ["a", "ab", "abc", "abcd", "11111111"]
m.min_by { |x| x.length } # => "a"
m.max_by { |x| x.length } # => "11111111"
```

`minmax` da Array'in minimum ve maximum'unu döner:

```
m = ["a", "ab", "abc", "abcd"]
m.min # => "a"
m.max # => "abcd"
m.minmax # => ["a", "abcd"]
```

Aynı mantıkta `minmax_by` da gerekli şarta göre min, max döner:

```
m = ["a", "ab", "abc", "abcd"]
m.minmax_by { |x| x.length } # => ["a", "abcd"]
```

## all?, any?, one?, none?

Array içindeki elemanları belli bir koşula göre kontrol etmek için kullanılır. Sonuç **Boolean** yani `true` ya da `false` döner. Tüm elemanların kontrolü koşula uyuyorsa `true` uymuyorsa `false` döner.

```
# acaba hayvanlar dizisindeki isimlerin hepsinin uzunluğu
# en az 2 karakter mi?

hayvanlar = ["Kedi", "Köpek", "Kuş", "Kurbağa", "Kaplumbağa"]
hayvanlar.all? { |hayvan_ismi| hayvan_ismi.length >= 2 } # => true

# Acaba ilk harfleri K harfimi?

hayvanlar.all? { |hayvan_ismi| hayvan_ismi.start_with?("K") } # => true

# Elemanların her biri true mu?
[true, false, nil].all? # => false
```

`any?` de yalnızca bir tanesi `true` olsa yeterlidir:

```
# En azından bir hayvan ismi A ile başlıyor mu?

hayvanlar = ["Kedi", "Köpek", "At", "Yılan", "Balık"]
hayvanlar.any? { |hayvan_ismi| hayvan_ismi.start_with?("A") } # => true
```

`one?` da ise sadece bir eleman koşula uymalıdır. Yani bir tanesi `true` dönmelidir. Eğer birden fazla eleman koşula `true` dönerse sonuç `false` olur:

```

hayvanlar = ["Kedi", "Köpek", "At", "Yılan", "Balık", "Kaplumbağa"]

# Sadece bir ismin uzunluğu 6 karaterten büyük olmalı!
hayvanlar.one?{ |hayvan_ismi| hayvan_ismi.length > 6 } # => true

# Uzunluğu 3'ten büyük 5 isim olduğu için false döndü!
hayvanlar.one?{ |hayvan_ismi| hayvan_ismi.length > 3 } # => false

```

none? da ise hepsi false olmalıdır ki sonuç true dönsün:

```

hayvanlar = ["Kedi", "Köpek", "At", "Yılan", "Balık", "Kaplumbağa"]

# Hiçbir ismin uzunluğu 2 karakter olmamalı ? false. At'ın uzunluğu 2
hayvanlar.none?{ |hayvan_ismi| hayvan_ismi.length == 2 } # => false

# C ile başlayan hayvan ismi olmasın! true. Hiçbir isim C ile başlamıyor
hayvanlar.none?{ |hayvan_ismi| hayvan_ismi.start_with?("C") } # => true

```

## inject, reduce\*

inject ve reduce aynı işi yaparlar ve bir tür akümülator işlemi yapmaya yararlar. Blok içinde 2 paramtre kullanılır. Başlama parametresi de alabilir. Örneğin Array [1, 2, 3, 4, 5] ve tüm elemanları birbiriyle toplamak isitiyoruz.

```

[1, 2, 3, 4, 5].inject{ |toplam, eleman| toplam + eleman } # => 15

# işlem şu şekilde ilerliyor
# toplam: 0, eleman: 1
# toplam: 1, eleman: 2
# toplam: 3, eleman: 3
# toplam: 6, eleman: 4
# toplam: 10, eleman: 5
# sona geldiğinde toplam 10, eleman 5 -> 10 + 5 = 15

```

Eğer başlangıç değeri için parametre geçseydik, örneğin 10:

```

[1, 2, 3, 4, 5].inject(10){ |toplam, eleman| toplam + eleman } # => 25

# toplam: 10, eleman: 1
# toplam: 11, eleman: 2
# toplam: 13, eleman: 3
# toplam: 16, eleman: 4
# toplam: 20, eleman: 5
# sona geldiğinde toplam 20, eleman 5 -> 20 + 5 = 25

```

Aynı işi reduce ile de yapabilirdik.

```

[1, 2, 3, 4, 5].reduce(:+) # => 15

```

## Array

Örnekte her elemanın `+` methodu'nu çağırıyoruz ve sanki `x = x + 1` mantığında, kendisini ekleye ekleye sonuca varıyoruz.

```
en_uzun_hayvan_ismi = ["kedi", "köpek", "kamlumbağa"].inject do |buffer, hayvan|
  buffer.length > hayvan.length ? buffer : hayvan
end

en_uzun_hayvan_ismi # => "kamlumbağa"
```

## partition ve group\_by

`partition` Array'i 2 parçaya ayırmaya yarar. Sonuç, blok'ta işlenen ifadeye bağlı olarak `[true_array, false_array]` olarak döner. Yani koşula `true` cevap verenlerle `false` cevap verenler ayrı parçalar halinde döner :)

```
[1, 2, 3, 4, 5, 6].partition{ |n| n.even? } # => [[2, 4, 6], [1, 3, 5]]

# Çift sayılar, true_array yani ilk parça: [2, 4, 6]
# Tek sayılar, false_array yani ikinci parça: [1, 3, 5]

# Sadece çift sayılar gelsin:
[1, 2, 3, 4, 5, 6].partition{ |n| n.even? }[0] # => [2, 4, 6]
```

`group_by` gruptama yapmak için kullanılır. Sonuç **Hash** döner, ilk değer (*key*) blok içindeki ifadenin sonucu, ikinci değer (*value*) ise sonucu verenlerin oluşturduğu gruptur. 1'den 6'ya kadar sayıları 3'e bölünce kaç kaldığını gruplayarak bulalım:

```
[1, 2, 3, 4, 5, 6].group_by{ |n| n % 3 } # => {1=>[1, 4], 2=>[2, 5], 0=>[3, 6]}

# 3'e bölünce kalanı;
# 1 olanlar: [1, 4]
# 2 olanlar: [2, 5]
# 0 olanlar (tam bölünenler) : [3, 6]
```

Notu 50'den büyük olanlar:

```
notlar = [50, 20, 44, 60, 80, 100, 99, 81, 5]
notlar.group_by{ |notu| notu > 40 }[true] # => [50, 44, 60, 80, 100, 99, 81]
```

## chunk

Array elemanları koşula göre gruplar.

```
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5].chunk { |n| n.even? }.to_a

# => [
# [false, [3, 1]],
# [true, [4]],
# [false, [1, 5, 9]],
# [true, [2, 6]],
# [false, [5, 3, 5]]
# ]
```

## slice\_before

Array içinde belli bir elemana ya da kurala göre parçalara ayırmak için kullanılır.

```
[1, 2, 3, 'a', 4, 5, 6, 'a', 7, 8, 9, 'a', 1, 3, 5].slice_before {|i| i == 'a' }.to_a
# => [[1, 2, 3], ["a", 4, 5, 6], ["a", 7, 8, 9], ["a", 1, 3, 5]]
```

## flat\_map, collect\_concat

İkisi de aynı işi yapar.

Önce `map` eder sonra `flatten` yapar.

```
pos_neg = [1, 2, 3, 4, 5, 6].map { |n| [n, -n] }
pos_neg      # => [[1, -1], [2, -2], [3, -3], [4, -4], [5, -5], [6, -6]]
pos_neg.flatten # => [1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6]

# yerine:
[1, 2, 3, 4, 5, 6].flat_map { |n| [n, -n] } # => [1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6]
```

## sort\_by

Aynı `sort` gibi çalışır, Blok kullanır. İfadenin `true` olmasına göre çalışır:

```
hayvanlar = ["kamlumbağa", "at", "eşşek", "kurbağa", "ayıl"]

# isimleri uzunluklarına göre küçükten büyüğe doğru sıralayalım
hayvanlar.sort_by{ |isim| isim.length } # => ["at", "ayıl", "eşşek", "kurbağa", "kamlumbağa"]

# isimleri uzunluklarına göre büyükten küçüğe doğru sıralayalım
hayvanlar.sort_by{ |isim| -isim.length } # => ["at", "ayıl", "eşşek", "kurbağa", "kamlumbağa"]
```

## bsearch

**Binary** arama yapar, `O(log n)` formülünü uygular, buradaki `n` Array'in boyudur. **Find minimum** gibidir, yani koşula ilk uyanı bul gibi...

```
# 2'den büyük 3, 4 ve 5 olmasına rağmen tek sonuç
[1, 2, 3, 4, 5].bsearch{ |n| n > 2 } # => 3

[1, 2, 3, 4, 5].bsearch{ |n| n >= 4 } # => 4
```

## Tehlikeli İşlemler

Başlarda da bahsettiğimiz gibi method ismi `!` ile bitiyorsa bu ilgili nesnede değişiklik yapıyor olduğumuz anlamına gelir. Array'lerde de bu tür method'lar var:

```
[ :reverse!, :rotate!, :sort!, :sort_by!, :collect!, :map!, :select!, :reject!, :slice!,
  :uniq!, :compact!, :flatten!, :shuffle! ]
```

Bu method'lar orijinal Array'i bozar. Yani;

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

# a artık reverse edilmiş halde!
a          # => [5, 4, 3, 2, 1]
```

# Hash

Pek çok dilde **Dictionary** olarak geçen, Array'imsi, hatta **Associative Array** de denir, **Key-Value** çifti barındıran yine Array'e benzeyen başka bir taşıyıcıdır. **Key-Value** dediğimiz şey ise;

```
{key1: "value1", key2: "value2", ....}
```

şeklindedir. Yukarıdaki örnek, yeni syntax'ı kullanır. Ruby programcılarının alışık olduğu örnek:

```
{"key1" => "value1", "key2" => "value2", ...}
```

ya da

```
{:key1 => "value1", :key2 => "value2", ...}
```

şeklindedir. Hepsi aynı kapıya çıkar... Array'deki sıra (*index*) mantığı burada **key**'ler ile oluyor gibi düşünebilirsiniz. Key'ler **unique**'dir yani bir Hash içinde **2 tane aynı key**'den olamaz.

Hash'de sonuçta bir class olduğu için `new` method'u ile Hash'i oluşturabiliriz;

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

Aynı Array'deki gibi Hash'in de hızlı oluşturma yolu var : `h = {}` . Hemen Hash'in nereden geldiğine bakalım:

```
Hash.class # => Class
Hash.class.superclass # => Module
Hash.class.superclass.superclass # => Object
Hash.class.superclass.superclass.superclass # => BasicObject
Hash.class.superclass.superclass.superclass.superclass # => nil
```

Dikkat ettiyseniz Hash'in bir üst sınıfı **Module**. Aynı Array'deki gibi. Peki bu modüller nelermiş?

```
Hash.included_modules # => [Enumerable, Kernel]
```

Eğer Hash'i oluştururken **default** değer geçerse, tanımsız olan **key** için değer atamış oluruz:

```
h = Hash.new("Tanımsız") # => {}
h[:isim] = "Uğur"        # => "Uğur"
h                        # => {:isim=>"Uğur"}
h[:soyad]                # => "Tanımsız"
h.default                # => "Tanımsız"
```

Olmayan bir key'e ulaşmak istediğimizde "Tanımsız" değeri geldi. Eğer bu default değeri atamasaydı ne olacaktı?

```
h = Hash.new            # => {}
h[:isim] = "Uğur"       # => "Uğur"
h[:soyad]              # => nil
```

`nil` gelecekti. Biraz sonra göreceğimiz `fetch` method'unu lütfen aklınızda tutun!

**Default** değer tanımlama mantığında;

```
h = Hash.new { |hash, key| hash[key] = "User: #{key}" }
h["vigo"]      # => "User: vigo"
h["foobar"]    # => "User: foobar"
h["animal"] = "horse" # => "horse"
h              # => {"vigo"=>"User: vigo", "foobar"=>"User: foobar", "animal"=>"horse"}
```

bu tarz ilginç bir yöntem de kullanılabilir. Normalde `vigo` key'ine karşılık **value** yok ama `Hash` in `new` method'unda yaptığımız bir blok işlemi ile olmayan `key` için değer ataması yaptığımız gibi key-value ataması da yapabiliyoruz.

## Hash Class Method'ları

Hash'den bir instance oluşturmadan kullandığımız methodlardır.

**Hash[ key, value, ... ] -> yeni\_hash**  
**Hash[ [ [key, value], ... ] ] -> yeni\_hash**  
**Hash[ object ] -> yeni\_hash**

```
Hash["user_count", 5] # => {"user_count"=>5}
Hash[ [{"user_count", 5}, {"active_users", 2} ] ] # => {"user_count"=>5, "active_users"=>2}
Hash["user_count" => 5, "active_users" => 2] # => {"user_count"=>5, "active_users"=>2}
```

### Hash.new

Zaten ilgili örnekleri başta vermiştik, tekrar edelim:



```
h = Hash.new
h # => {}
h["user_count"] = 5
h # => {"user_count"=>5}

h = Hash.new { |hash, key| hash[key] = "User ID: #{key}" }
h["1"] # => "User ID: 1"
h["2"] # => "User ID: 2"
h # => {"1"=>"User ID: 1", "2"=>"User ID: 2"}
```

### try\_convert(obj) → hash ya da nil

Hash'e dönüşebilme ihtimali olan nesneyi Hash haline çevirir.

```
Hash.try_convert({"user_count"=>5}) # => {"user_count"=>5}
Hash.try_convert("user_count=>5") # => nil
```

## Hash Instance Method'ları

Hash instance'ı oluşturduktan sonra kullanacağımız method'lardır.

Önce klasik değer okuma ve değer atama işlerine bakalım. Zaten bu noktaya kadar kabaca biliyoruz nasıl değer atarız geri okuruz. Ama biraz kafaları karıştırmak istiyorum:

```
h = {username: "vigo", password: "1234"} # => {:username=>"vigo", :password=>"1234"}
```

Yukarıdaki gibi bir **Hash**'imiz var. Dikkat ettiyseniz, key,value olarak baktığımızda `:username` ve `:password` diye başlayan `key` ler var... Hatta:

```
h.keys # => [:username, :password]
```

diye de sağlamasını yaparız. Peki, yeni bir `key` tanımlasak? `h["useremail"] = "vigo@example.com"` . Tekrar bakalım `key` lere:

```
h.keys # => [:username, :password, "useremail"]
```

Bir sonraki bölümde karşımıza çıkacak olan **Symbol** tipi ile karşı karşıyayız. Sadece **symbol** mu? hayır, karışık keyler var elimizde. Hemen sağlamasını yapalım:

```
h.keys.map(&:class) # => [Symbol, Symbol, String]
```

İlk iki key **Symbol** iken son key **String** oldu. Demek ki Hash içine key cinsi olarak karşık atama yapabiliyor. Biraz sıkıntılı bir durum ama genel kültür mahiyetinde aklınızda tutun bunu!

Şimdi genel olarak Hash hangi method'lara sahip hemen bakalım:

```
h = Hash.new
h.methods # => [:rehash, :to_hash, :to_h, :to_a, :inspect, :to_s, :==, :[], :hash, :eq
l?, :fetch, :[]=, :store, :default, :default=, :default_proc, :default_proc=, :key, :i
ndex, :size, :length, :empty?, :each_value, :each_key, :each_pair, :each, :keys, :valu
es, :values_at, :shift, :delete, :delete_if, :keep_if, :select, :select!, :reject, :re
ject!, :clear, :invert, :update, :replace, :merge!, :merge, :assoc, :rassoc, :flatten,
:include?, :member?, :has_key?, :has_value?, :key?, :value?, :compare_by_identity, :c
ompare_by_identity?, :entries, :sort, :sort_by, :grep, :count, :find, :detect, :find_i
ndex, :find_all, :collect, :map, :flat_map, :collect_concat, :inject, :reduce, :partit
ion, :group_by, :first, :all?, :any?, :one?, :none?, :min, :max, :minmax, :min_by, :ma
x_by, :minmax_by, :each_with_index, :reverse_each, :each_entry, :each_slice, :each_con
s, :each_with_object, :zip, :take, :take_while, :drop, :drop_while, :cycle, :chunk, :s
lice_before, :lazy, :nil?, :===, :=~, :!~, :<=>, :class, :singleton_class, :clone, :du
p, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :freeze, :frozen?, :met
hods, :singleton_methods, :protected_methods, :private_methods, :public_methods, :inst
ance_variables, :instance_variable_get, :instance_variable_set, :instance_variable_def
ined?, :remove_instance_variable, :instance_of?, :kind_of?, :is_a?, :tap, :send, :publ
ic_send, :respond_to?, :extend, :display, :method, :public_method, :singleton_method,
:define_singleton_method, :object_id, :to_enum, :enum_for, :equal?, :!, :!=, :instance
_eval, :instance_exec, :__send__, :__id__]
```

Dikkat ettiyseniz method'ların bir kısmı **Array** ile aynı çünkü ikisi de **Enumerable** modülünü kullanıyor.

Şimdi sıradan başlayalım! Önceki **Array** bölümünde anlattığım ortak method'ları pas geçeceğim!

## rehash

Hash'e key olarak Array verebiliriz. Yani `h[key] = value` mantığında `key` olarak bildiğiniz Array geçebiliriz.

```
a = [ "a", "b" ]
c = [ "c", "d" ]
h = { a => 100, c => 300 } # => {"a", "b"=>100, "c", "d"=>300}
```

`h` Hash'inin keyleri nedir?

```
h.keys # => [{"a", "b"}, {"c", "d"}]
```

2 key'i var biri `["a", "b"]` ve diğeri `["c", "d"]` nasıl yani?

## Hash

```
h[a]           # => 100
h[["a", "b"]]  # => 100
h[c]           # => 300
h[["c", "d"]]  # => 300
```

Şimdi işleri karıştıralım. `a` Array'inin ilk değerini değiştirelim. Bakalım `h` ne olacak?

```
a[0] = "v"      # => "v"
a             # => ["v", "b"]
h[a]           # => nil ????????
```

`h[a]` patladı? `nil` döndü. İşte şimdi imdadımıza ne yetişecek?

```
h.rehash       # => {["v", "b"]=>100, ["c", "d"]=>300}
h[a]           # => 100
```

## to\_hash, to\_h, to\_a, to\_s

Tip dönüştürmeleri için kullanılırlar. `to_h` ve `to_hash` eğer kendisi Hash ise sonuç yine kendisi olur. `to_a` ise Hash'den Array yapmak için kullanılır. Tahmin edeceğiniz gibi `to_s` de String'e çevirmek için kullanılır.

```
h = {:foo => "bar"}
h             # => {:foo=>"bar"}
h.to_hash    # => {:foo=>"bar"}
h.to_h       # => {:foo=>"bar"}
["foo", "bar"].respond_to?(:to_h) # => true
[{:foo, "bar"}].to_h      # => {:foo=>"bar"}

[["a", 1], ["b", 2]].to_h # => {"a"=>1, "b"=>2}

h.to_a       # => [{:foo, "bar"}]
h.to_s       # => "{:foo=>\"bar\"}"
```

## == ve eql? Eşitlik

Hash içinde key'lerin sırası eşitlik kontrolünde önemli değildir. İçerik önemlidir. Eşitlik kontrolü için kullanılırlar.

```
h1 = { "a" => 100, "c" => 200 }
h2 = { 70 => 350, "x" => 22, "y" => 11 }
h3 = { "y" => 11, "x" => 22, 70 => 350 }

h1 == h2 # => false
h2 == h3 # => true

h1.eql?(h2) # => false
h2.eql?(h3) # => true
```

`h2` ile `h3` key sıraları farklı olmasına rağmen içerik bazında eşittirler.

## fetch

Hash içinden sorgu yaparken kullanılır. Eğer olmayan key çağırırsanız **exception** oluşur. Bu method güvenli bir yöntemdir. Aksi takdirde `nil` döner ve kompleks işlerde **Silent Fail** yani dipsiz kuyuya düşer bir türlü hatanın yerini bulamazsınız!

```
h = {:user => "vigo", :password => "secret"}
puts h.fetch(:user) # "vigo"
puts h.fetch(:email)

KeyError: key not found: :email
```

Keza eğer key'e karşılık yoksa **default** değer ataması yapabilirsiniz:

```
h = {:user => "vigo", :password => "secret"}
h.fetch(:user) # => "vigo"
h.fetch(:email, "Not found") # => "Not found"
```

**Block** kabul ettiği için artistlik hareketler yapmak da mümkün :)

```
h = {:user => "vigo", :password => "secret"}
h.fetch(:email) { |element| "key: #{element} is not defined!" } # => "key: email is not defined!"
```

## store

Atama yapmanın farklı bir yöntemidir.

```
h = {:user => "vigo", :password => "secret"}
h.store(:email, "vigo@example.com") # => "vigo@example.com"
h # => {:user=>"vigo", :password=>"secret", :email=>"vigo@example.com"}

# ya da

h[:url] = "http://webbox.io" # => "http://webbox.io"
h # => {:user=>"vigo", :password=>"secret", :email=>"vigo@example.com", :url=>"http://webbox.io"}
```

## default, default=

Karşılığı olmayan key ler için varsayılan değer ataması yapılmışsa bunu bulmak için ya da varsayılan değeri atamak için kullanılır. En başta benzer işler yaptık:

## Hash

```
h = Hash.new(10)
h[:user_age]      # => 10
h                 # => {}
h.default         # => 10
h.default(:user_weight) # => 10
```

ya da

```
h = Hash.new
h                 # => {}
h.default = 100  # => 100
h[:user_weight]  # => 100
h[:foo]          # => 100
```

## key

Value'den key'i bulmak için kullanılır. Eğer key'i olmayan bir value kullanırsanız sonuç `nil` döner!

```
h = {:user => "vigo", :password => "secret"}
h.key("vigo")   # => :user
h.key("foobar") # => nil
```

## size, length, count

Aynı işi yaparlar, Arrar gibi Hash'in boyunu / uzunluğunu verir.

```
h = {:user => "vigo", :password => "secret"}
h.length # => 2
h.size   # => 2
h.count  # => 2
```

# Key, Value Kontrolleri

## keys, values, values\_at

Tahmin edeceğiniz gibi `keys` ile Hash'e ait key'leri, `values` ile sadece key'lere karşılık gelen değerleri, `values_at` ile verdiğimiz key'lere ait değerleri alırız.

```
h = {:user => "vigo", :password => "secret", :email => "vigo@foo.com"}
h.keys           # => [:user, :password, :email]
h.values         # => ["vigo", "secret", "vigo@foo.com"]
h.values_at(:user, :password) # => ["vigo", "secret"]
```

## key?, value?, has\_key?, has\_value?

Soru işareti ile biten method'lar bize her zaman **Boolean** yani `true` ya da `false` döner demiştik. Acaba Hash'in içinde ilgili key var mı? ya da value var mı?

```
h = { :user => "vigo", :password => "secret", :email => "vigo@foo.com" }
h.key?(:user)           # => true
h.has_key?(:user)       # => true

h.key?(:full_name)      # => false
h.has_key?(:full_name)  # => false

h.value?("vigo")        # => true
h.has_value?("vigo")    # => true

h.value?("lego")        # => false
h.has_value?("lego")    # => false
```

### include?, member?

`key?` ya da `has_key?` ile aynı işi yapar.

```
h = { :user => "vigo", :password => "secret", :email => "vigo@foo.com" }
h.include?(:user)       # => true
h.member?(:user)        # => true
```

### empty?

Hash'in içinde eleman var mı yok mu?

```
{ :user => "vigo", :password => "secret", :email => "vigo@foo.com" }.empty? # => false
{}.empty? # => true
```

### all?, any?, one?, none?

**Array** bölümünde görmüştük, **Enumerable** modülünden gelen bu özellik aynen Hash'de de kullanılıyor. `all?` da tüm elemanlar, verilen koşuldan `nil` ya da `false` dışında birşey dönmek zorunda, aksi halde sonuç `false` oluyor:

```
# value'su boş olan var mı?
{ :user => "vigo", :password => "secret", :email => "vigo@foo.com" }.all? { |k,v| v.empty? } # => false
{ :user => "", :password => "", :email => "" }.all? { |k,v| v.empty? } # => true
{ :user => "vigo", :password => "", :email => "" }.all? { |k,v| v.empty? } # => false
```

`any?` de içlerinden biri `false` ya da `nil` dönmezse sonuç `true` olur. `one?` da sadece bir tanesi `true` dönmelidir. `none` da ise block'daki işlem sonucu her eleman için `false` olmalıdır.

```
{:is_admin => true, :notifications_enabled => true}.all?{ |option, value| value } # =>
true
{:is_admin => true, :notifications_enabled => false}.any?{ |option, value| value } # =
> true
{:is_admin => true, :notifications_enabled => false}.one?{ |option, value| value } # =
> true
{:is_admin => false, :notifications_enabled => false}.one?{ |option, value| value } #
=> false
{:is_admin => false, :notifications_enabled => false}.all?{ |option, value| value } #
=> false
{:is_admin => false, :notifications_enabled => false}.none?{ |option, value| value } #
=> true
{:is_admin => false, :notifications_enabled => false}.any?{ |option, value| value } #
=> false
```

## shift

Hash'den key-value çiftini silmek için kullanılır. Her seferinde ilk key-value çiftini siler.

```
h = {:user => "vigo", :password => "secret", :email => "vigo@foo.com"}
h.shift # => [:user, "vigo"]
h      # => {:password=>"secret", :email=>"vigo@foo.com"}
h.shift # => [:password, "secret"]
h      # => {:email=>"vigo@foo.com"}
h.shift # => [:email, "vigo@foo.com"]
h      # => {}
```

## delete, delete\_if, keep\_if

Hash'den key kullanarak eleman silmek için `delete` method'u kullanılır.

```
h = {:user => "vigo", :password => "secret", :email => "vigo@foo.com"}
h.delete(:user) # => "vigo"
h              # => {:password=>"secret", :email=>"vigo@foo.com"}
```

Block kullanıldığında, eğer olmayan bir key kullanılmışsa, bununla ilgili işlem yapmamızı sağlar:

```
h.delete(:phone){ |key| "-#{key}- bulunamadı?" } # => "-phone- bulunamadı?"
```

`delete_if` de ise direk block kullanarak koşullu silme işlemi yapabiliyoruz.

```
# 40'dan büyükleri silelim
h = { point_a: 10, point_b: 20, point_c: 50 } # => {:point_a=>10, :point_b=>20, :point
_c=>50}
h.delete_if{ |k,v| v > 40 }                  # => {:point_a=>10, :point_b=>20}
h                                             # => {:point_a=>10, :point_b=>20}
```

`keep_if` ise `delete_if` in tam tersi gibidir. Eğer block'daki koşul `true` ise key-value çiftini tutar, aksi halde siler:

```
# 20'dan küçükleri tutalım sadece
h = { point_a: 10, point_b: 20, point_c: 50 }
# => {:point_a=>10, :point_b=>20, :point_c=>50}
h.keep_if{ |k,v| v < 20 } # => {:point_a=>10}
h                         # => {:point_a=>10}
```

## invert

Hash'in key'leri ile value'lerini yer değiştirmek için kullanılır.

```
h = { "a" => 100, "b" => 200 } # => {"a"=>100, "b"=>200}
h.keys                         # => ["a", "b"]
h.invert                       # => {100=>"a", 200=>"b"}
```

## merge, update, merge!

İki Hash'i birbiriyle birleştirmek için `merge` kullanılır.

```
h1 = { "a" => 100, "b" => 200 }
h2 = { "x" => 1, "y" => 2, "z" => 3 }
h1.merge(h2) # => {"a"=>100, "b"=>200, "x"=>1, "y"=>2, "z"=>3}
h1           # => {"a"=>100, "b"=>200}
```

Dikkat ettiyseniz `h1` ile `h2` yi birleştirdik ama `h1` in orijinal değerini bozmadık. Eğer bu birleşmenin kalıcı olmasını isteseydik ya `update` ya da `merge!` kullanmamız gerekecekti!

```
h1 = { "a" => 100, "b" => 200 }
h2 = { "x" => 1, "y" => 2, "z" => 3 }
h1.update(h2) # => {"a"=>100, "b"=>200, "x"=>1, "y"=>2, "z"=>3}
h1           # => {"a"=>100, "b"=>200, "x"=>1, "y"=>2, "z"=>3}

h1 = { "a" => 100, "b" => 200 }
h2 = { "x" => 1, "y" => 2, "z" => 3 }
h1.merge!(h2) # => {"a"=>100, "b"=>200, "x"=>1, "y"=>2, "z"=>3}
h1           # => {"a"=>100, "b"=>200, "x"=>1, "y"=>2, "z"=>3}
```

## replace

Hash'in içeriğini başka bir Hash ile değiştirmek için kullanılır. Aslında varolan Hash'i başka bir Hash'e çevirmek gibidir. Neden `replace` kullanılıyor? Tamamen hafızadaki adresleme ile ilgili. `replace` kullanıldığı zaman, aynı Hash kullanılıyor, yeni bir Hash instance'i yaratılmıyor.



```
h1 = { "a" => 100, "b" => 200, "c" => 0 }
h1.__id__ # => 70320602334320
```

Hafızadaki `h1` Hash'nin nesne referansı : 70320602334320. Şimdi `replace` ile değerlerini değiştirelim:

```
h1.replace({ "foo" => 1, "bar" => 2 })
h1 # => {"foo"=>1, "bar"=>2}
h1.__id__ # => 70320602334320
```

Referansları aynı : **70320602334320**. Eğer direkt olarak atama yapsakdık `h1` gibi görünen ama bambaşka yepyeni bir Hash'imiz olacaktı.

```
h1 = { "foo" => 1, "bar" => 2 }
h1.__id__ # => 70216424232360
```

## İterasyon ve Block Kullanımı

Aynı Array'lerdeki gibi Hash'lerde de iterasyon ve block kullanmak mümkün.

**each, each\_pair, each\_value, each\_key**

`each` ve `each_pair` kardeş gibidirler:

```
h = { "a" => 100, "b" => 200, "c" => 0 }
h.each { |key, value| puts "key: #{key}, value: #{value}" }
h.each_pair { |key, value| puts "key: #{key}, value: #{value}" }

# key: a, value: 100
# key: b, value: 200
# key: c, value: 0
```

`each_value` sadece **value**, `each_key` de sadece **key** döner.

```
h = { "a" => 100, "b" => 200, "c" => 0 }

h.each_value { |value| puts "value: #{value}" }

# value: 100
# value: 200
# value: 0

h.each_key { |key| puts "key: #{key}" }

# key: a
# key: b
# key: c
```

### each\_entry, each\_slice, each\_cons

Hash'deki key-value çifti Array şeklinde bir **entry** olur:

```
h = { "a" => 100, "b" => 200, "c" => 0 }
h.each_entry{ |o| puts "o: #{o}" }

# o: ["a", 100]
# o: ["b", 200]
# o: ["c", 0]
```

`each_slice` ile **entry**'leri parçacıklara ayırırız:

```
h = { "a" => 100, "b" => 200, "c" => 0 }

# 2'li dilimlere ayırdık
h.each_slice(2){ |s| puts "slice: #{s}" }

# slice: ["a", 100], ["b", 200]
# slice: ["c", 0]
```

`each_cons` ise `each_slice` gibi çalışır ama farkı örnekteki gibidir:

```
h = { "a" => 100, "b" => 200, "c" => 0 }
h.each_cons(2){ |s| puts "grup: #{s}" }

# grup: ["a", 100], ["b", 200]
# grup: ["b", 200], ["c", 0]
```

Neticede, 3 key-value çifti vardı. 2'li grupladık ama sonuç `each_slice` daki gibi dönmedi. `["b", 200]` tekrar etti, çıktı gruplaması mutlaka 2 eleman içerdi.

### default\_proc, default\_proc=

Konunun başında varsayılan değer ataması yaparken şöyle bir örnek vermiştik :

## Hash

```
h = Hash.new { |hash, key| hash[key] = "User: #{key}" }
```

eğer;

```
h.default_proc # => #<Proc:0x007f85f2250fd8@-:7>
```

deseydik, bu Hash'e ait **Proc** u görmüş olurduk. Yani bu Hash için varsayılan işlem prosedürünü tanımlamış olduk aslında. Örneği biraz genişletelim:

```
h = Hash.new {|obj, key| obj[key] = key * 4 } # => {}  
h[1] # => 4  
h[2] # => 8  
h    # => {1=>4, 2=>8}
```

Key olarak sayı veriyoruz, gelen sayıdan da value üretiyoruz otomatik olarak. İşlemin çalışması için bir adet obje ve sayı geçmemiz gerekiyor parametre olarak. Aslında;

```
h.default_proc.call(Array.new, 9) # => 36  
h.default_proc.call([], 9) # => 36  
h.default_proc.call({}, 9) # => 36
```

şeklinde de, Hash'i sanki bir fonksiyon gibi kullanıp işleyebiliyoruz.

Daha sonra, önceden tanımladığımız bu prosedürü değiştirmek istersek `default_proc=` methodunu kullanıyoruz:

```
h = Hash.new { |hash, key| hash[key] = "User: #{key}" }  
h.default_proc # => #<Proc:0x007f85f2250fd8@-:7>  
h[1] # => "User: 1"  
  
# Yeni prosedür veriyoruz  
h.default_proc = proc do |hash, key|  
  hash[key] = "hello #{key}"  
end  
h # => {1=>"User: 1"}  
h[2] # => "hello 2"  
h # => {1=>"User: 1", 2=>"hello 2"}
```

## compare\_by\_identity, compare\_by\_identity?

Hash'in key ve value'leri birbirine benziyor mu?

## Hash

```
h = { "a" => 1, "b" => 2, :c => "c" }
h["a"] # => 1
h.compare_by_identity? # => false
h.compare_by_identity # => {"a"=>1, "b"=>2, :c=>"c"}
h.compare_by_identity? # => true

# acaba key ile value benziyormu?
h["a"] # => nil

# burada benzer :)
h[:c] # => "c"
```

# Symbol

Symbol (*sembol*) Ruby'e ait özel bir nesnedir. Bir tür **placeholder** (*yer tutucu*) görevindedir.

: işareti ile başlayan herşey semboldür.

Sembolün, değişkenden en önemli farkı tekil olmasıdır. Yani sembole atanan değişkenden hafızada 1 adet bulunur.

```
user = "vigo"
user.object_id # => 70122132113780

# şimdi başka değer atayalım
user = "bronx"
user.object_id # => 70122132113340
# object_id değişti!
```

Eğer Symbol kullansaydık:

```
user = :vigo
user.object_id # => 420488

user = :bronx
user.object_id # => 420648
```

`user` değişkeninin değeri **Symbol** cinsinden olduğu için, artık hafızada sabit bir yer ayrılmış oldu bu iş için. Değer değişse bile hafızadaki adreslendiği alan değişmemiş oluyor :)

String olarak atanmış değişkeni de Symbol'e çevirmek mümkün:

```
full_name = "Uğur" # => "Uğur"
full_name.to_sym # => :Uğur
full_name == :Uğur.id2name # => true
user_full_name = :Uğur # => :Uğur
user_full_name.object_id # => 420428

:is_user_admin.id2name # => "is_user_admin"
:is_user_admin.to_s # => "is_user_admin"
```

Symbol'ler, değişkenler gibi direkt atama yöntemiyle yani `:a = 1` gibi bir şekilde çalışmazlar. Eğer bir String'den Symbol üretmek isterseniz `to_sym` methodunu kullanmanız gerekiyor.

Hafızayı idareli kullanmak, boşu boşuna değişken kirliliği yaratmamak gibi konularda tercih edilir. Keza Hash'lerde de **KEY** ataması Symbol olarak yapılıyor bu tür hız / tasarruf işleri için.

```
{:user=>"vigo"}
```

# Class (Sınıf)

Ruby, **Object-Oriented** (OO) bir dil olduğu için, methodları değişkenleri ve benzeri şeyleri içinde barındıran bir tür taşıyıcıya ihtiyaç duyar. İşte bu taşıyıcıya **Class** ya da sınıf diyoruz. Aslında ben **tür** demeyi tercih ediyorum sınıf yerine.

Zaten önceki konularda **Class Methods**, **Instance Methods** gibi kavramlara girmiştik.

Class'lar birbirinden türeyebilir (*Hani `class.superclass` şeklinde analizler yapmıştık*)

Teknik olarak bir dosyada birden fazla Class tanımlaması yapılabilir. Örneğin, `my_class.rb` adlı bir dosya içinde farklı farklı Class tanımlamaları olabilir;

```
class MyClass
end

class OtherClass
end

a = MyClass.new # => #<MyClass:0x007ffa2b09b758>
b = OtherClass.new # => #<OtherClass:0x007ffa2b09b3c0>
```

Class tek başına bir nesne (*Obje*) bu bakımdan **instantiate** etmeseniz bile (*Yani `a = Array.new` gibi*) kullanabileceğiniz bir şeydir.

Class'ların diğer bir özelliği de açık olmasıdır. Ruby'nin bence en harika özelliği, **built-in** yani dilin çekirdeğinden gelen Class'lara bile method / property eklemeniz mümkündür. Bu konuları **Monkey Patching** kısmında detaylı göreceğiz.

En basit tanımıyla Class aşağıdaki gibidir:

```
class Merhaba
  def initialize(isim)
    @isim = isim
  end

  def selam_sana
    "Selam sana, #{@isim}"
  end
end

hey = Merhaba.new "Uğur"
hey.selam_sana # => "Selam sana, Uğur"
```

`initialize` methodu, Class'dan ilk örnek türedildiğinde (*yani bir instance oluşturulduğunda*) tetiklenir ve bir tür Class ile ilgili ön tanımlamaların yapıldığı alan konumundadır. Benzer dillerdeki **class constructor**'ı gibi düşünülebilir.

`@isim` ise **Instance Variable** yani Class'tan türeyen nesneye ait değişkendir.

`selam_sana` ise bu Class'ın bir method'udur. `hey` değişkeni, `Merhaba` Class'ından türemiş bir **Instance**'dir. `Merhaba` sınıfındaki tüm method'lar **inherit** yani miras olarak `hey` nesnesine geçmiştir.

Class'ı tanımlarken ister klasik ister block yöntemini kullanabilirsiniz. Klasik yöntem:

```
class Person
end

jack = Person.new # => #<Person:0x007fb0521a4820>
```

Block ise;

```
Person = Class.new do
end

jack = Person.new # => #<Person:0x007fc42c0c8648>
```

şeklinde. Class isimleri **Büyük** harfle başlar.

## Public Instance Method'ları

Bir Class'tan türeyen şeye **Instance** diyoruz. Ruby'de Class'lar **first-class objects** olarak geçer yani birinci sınıf nesnelerdir. Bu da şu anlama gelir, aslında her Class, Kernel'dan gelen `class` nesnesinden türemiş alt sınıftır :)

**allocate**, **new** ve **superclass**

`superclass` ilgili Class'ın kimden geldiğini / türediğini gösterir. Benzer örnekleri kitabın başında yapmıştık:

```
String.superclass # => Object
Object.superclass # => BasicObject
BasicObject.superclass # => nil
```

`new` ise önce `allocate` method'unu çağırıp hafızada gereken yeri ayırır, yani **instantiate** edeceğimiz Class'ın sınıfını organize eder, sonra oluşacak Class'ın `initialize` method'unu çağırıp varsa ilgili argümanları pas eder. Her `.new` çağırıldığında bu iş olur.

## Private Instance Method'ları

**inherited(subclass)**



İlgili sınıfın alt sınıfı oluşturulduğunda tetiklenir.

```
class Animal
  def self.inherited(subclass)
    puts "Yeni subclass: #{subclass}"
  end
end

class Cat < Animal
end

class Tiger < Cat
end

# Yeni subclass: Cat
# Yeni subclass: Tiger
```

Animal (*hayvan*) sınıfından, Cat (*kedi*) ürettik, Tiger (*kaplan*)'ı da yine Cat (*kedi*)'den ürettik...

## Accessors (getter + setter)

Özel method'lar kullanarak **Meta Programming** mantığıyla Ruby, Instance Variable'ları yönetmeyi kolaylaştırır. Yukarıdaki Merhaba Class'ındaki `@isim` için aslında `get` ve `set` yani oku ve yaz method'ları tanımlamamız lazım ki ilgili değişken üzerinde işlem yapabilelim. Önce uzun yolu, sonra doğru ve kısa yolu görelim:

```
class Person
  def name
    @name
  end

  def name=(name)
    @name = name
  end
end

vigo = Person.new # => #<Person:0x007f903b8d0590>
vigo.name         # => nil
vigo.name = "Uğur" # => "Uğur"
vigo.name         # => "Uğur"
```

`name` method'unu çağırınca bize instance variable olan `@name` dönüyor. İlk anda **set** etmediğimiz yani değer atamadığımız için `nil` geliyor. Daha sonra `vigo.name = "Uğur"` diyerek atama yapıyoruz ve artık değerini belirlemiş oluyoruz. Bu iş için 2 tane method yazdık. `name` ve `name=` method'ları.

İşte bu noktada **accessors** imdadımıza yetişiyor:

```
class Person
  attr_accessor :name
end

vigo = Person.new # => #<Person:0x007fb7c9a4c620>
vigo.name         # => nil
vigo.name = "Uğur" # => "Uğur"
vigo.name         # => "Uğur"
```

`attr_accessor :name` dediğimizde, Ruby, bizim için `name` ve `name=` method'ları oluşturuyor. Keza sadece bununla kalmayıp, pek çok farklı kullanım imkanları sunuyor.

`attr` modülüyle:

- `attr`
- `attr_accessor`
- `attr_reader`
- `attr_writer`

gibi özel getter/setter'lar geliyor. Yukarıdaki örneği `attr` ile yapalım;

```
class Person
  attr :name, true
end

Person.instance_methods - Object.instance_methods # => [:name, :name=]
```

Otomatik olarak 2 method ekledi : `[:name, :name=]` . Aynı şeyi `attr_accessor :name` ile de yapabiliydik:

```
class Person
  attr_accessor :name
end

Person.instance_methods - Object.instance_methods # => [:name, :name=]
```

Eğer sadece `attr_reader` kullansaydık, sadece ilgili instance variable'ını okuyabilir ama değerini set edemezdik!

```
class Person
  attr_reader :name
end

Person.instance_methods - Object.instance_methods # => [:name]

vigo = Person.new
vigo.name # => nil
vigo.name = "Uğur" # => NoMethodError: undefined method `name=' for #<Person:0x007ffe4d0e8528>
```

Gördüğünüz gibi `NoMethodError` hatası aldık çünkü setter yani `name=` method'u oluşmadı! Peki sadece `attr_writer` olsaydı?

```
class Person
  attr_writer :name
end

Person.instance_methods - Object.instance_methods # => [:name=]

vigo = Person.new
vigo.name = "Uğur" # => "Uğur"
vigo.name # => NoMethodError: undefined method `name' for #<Person:0x007fb92b9d45b8 @name="Uğur">
```

Set edebiliyoruz ama get edemiyoruz! Peki `attr_writer` nerede işimize yarar? Örneğin sadece Class'ı initialize ederken değer pas edip sınıf içinde bir değişkene atama yapmak gerektiğinde kullanabilirsiniz:

```
class Person
  attr_writer :name

  def initialize(name)
    @name = name
  end

  def greet
    "Hello #{@name}"
  end
end

vigo = Person.new "Uğur"
vigo.greet # => "Hello Uğur"
```

`@name` değişkenini sadece ilk tetiklenmede set ediceksem ve dışarıdan okuma ihtiyacım yoksa bu şekilde kullanabilirim!

## Class Variables

`@@` ile başlayan değişkenler Class Variable (*Sınıf Değişkeni*) olarak tanımlanır. Yani Ana Class'a ait bir değişkendir. Her yeni instance oluştuğunda bu değer ait olduğu üst sınıftan erişilebilir:

```
class Person
  attr_accessor :name

  @@amount = 0
  def initialize(name)
    @@amount += 1
    @name = name
  end

  def greet
    "Hello #{name}"
  end

  def how_many_people_created
    "Number of people: #{@amount}"
  end
end

user1 = Person.new "Uğur"
user2 = Person.new "Yeşim"
user3 = Person.new "Ezel"

Person.class_variable_get(:@@amount) # => 3
user3.how_many_people_created         # => "Number of people: 3"
```

## Class Methods

İlgili Class'dan türetme yapmadan, direk Class'dan çağırılan özel method'dur. Bu method'u çağırmak için sınıftan herhangi bir türetme yapmaya gerek olmaz, direkt olarak sınıftan çağırılır:

```

class Person
  attr_accessor :name

  @@amount = 0
  def initialize(name)
    @@amount += 1
    @name = name
  end

  def greet
    "Hello #{name}"
  end

  def how_many_people_created
    "Number of people: #{@amount}"
  end

  def self.how_many_people_created
    "We have #{@amount} copie(s)"
  end
end

user1 = Person.new "Uğur"
user2 = Person.new "Yeşim"
user3 = Person.new "Ezel"

Person.how_many_people_created # => "We have 3 copie(s)"

```

`Person.how_many_people_created` direkt olarak çağırılır!

## Singletons

Sınıf içinde `class` komutunu kullanarak method oluşturmak içindir. Buna **Singleton** denir. Sadece bir kere **instantiate** (*tetiklenme diyelim*) olur. Örneğin alan hesabı yapacak bir sınıf düşünüyoruz ve bunun `calculate` method'u olsun. En x Boy bize metrekare'yi versin:

```

class Area
  class << self
    def calculate(width, height)
      width * height
    end
  end
end

Area.calculate(5, 5) # => 25

```

Gördüğümüz gibi hiçbir şekilde `new` ya da benzer bir şey türetme kullanmadık direkt olarak `Area.calculate(5, 5)` şeklinde kullandık. Keza aynı işi;

```
class Area
end

x = Area.new
def x.calculate(width, height)
  width * height
end
x.calculate 5,5 # => 25
```

şeklinde de yapabilirdik.

## Inheritance (Miras)

Aslında bu da bildiğimiz bir şey. Sınıftan türeme yaparken, türettiğimiz sınıfın özellikleri türeyene miras geçer.

```
class Animal
  attr_accessor :name, :kind

  def initialize(name)
    @name = name
  end

  def say_hi
    "Hello! I'm a #{@kind}, my name is #{@name}"
  end
end

class Cat < Animal
end

class Horse < Animal
end

bidik = Cat.new "Bıdık"
bidik.kind = "cat"

zuzu = Horse.new "Zuzu"
zuzu.kind = "horse"

bidik.say_hi # => "Hello! I'm a cat, my name is Bıdık"
zuzu.say_hi  # => "Hello! I'm a horse, my name is Zuzu"
```

Cat ve Horse **Animal** sınıfından < yöntemiyle türedi ve Animal deki tüm method'lar Cat ve Horse 'a geçti.

# Access Level (Erişim): Public, Private, ve Protected Method'lar

Class içindeki method'lar duruma göre erişilebilirlik açısından kısıtlanabilir. `public` olanlar her yerden erişilebilirken (*bu default bir durumdur*), `private` olana sadece içeriden erişilebilir, `protected` olana ise ancak alt sınıftan türeyenden erişilebilir.

```
class User
  def bu_sayede_private_cagirabilirim
    bu_sadece_iceriden
  end

  private
  def bu_sadece_iceriden
    puts "Bu private method. Bu method instance'dan çağırılmaz!"
  end

  protected
  def bu_sadece_subclass_veya_instance_dan
    puts "Bu protected method."
  end
end

u = User.new
u.bu_sadece_iceriden # => NoMethodError: private method 'bu_sadece_iceriden' called for #<User:0x007feb9d0d2560>
```

Gördüğümüz gibi `bu_sadece_iceriden` method'unu `User` dan instantiate ettiğimiz `u` üzerinden çağırıyoruz. `private` olduğu için ancak içeriden çağırılabilir:

```
u.bu_sayede_private_cagirabilirim # => "Bu private method. Bu method instance'dan çağırılmaz!"
```

`public` olan `bu_sayede_private_cagirabilirim` method'u içeriden `private` method olan `bu_sadece_iceriden` 'e erişebildi. Peki ya `protected` ? Eğer direkt olarak çağırmaya kalksaydık:

```
u.bu_sadece_subclass_veya_instance_dan # => NoMethodError: protected method 'bu_sadece_subclass_veya_instance_dan' called for #<User:0x007fff131c60a0>
```

Hemen gerekeni yapalım; `User` Class'ından başka bir Class üretilim:

```
class SuperUser < User
  def initialize
    bu_sadece_subclass_veya_instance_dan
  end
end

y = SuperUser.new # => "Bu protected method."
```

## Method Aliasing

Bazı durumlarda, üst sınıftaki method'u ezmek gerekir. Bu işlemi yaparken aslında üst sınıftaki orijinal method'a da erişmeniz gerekebilir.

```
class User
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def give_random_age
    (20..45).to_a.sample
  end
end

class SuperUser < User
  alias :yedik :give_random_age # üst sınıftaki give_random_age'i sakladık, yedik adını verdik

  def give_random_age
    rnd = self.yedik
    "Kendi yaşı: 43, rnd= #{rnd}"
  end
end

u = User.new "vigo"
u.name # => "vigo"
u.give_random_age # => 29

v = SuperUser.new "Uğur"
v.give_random_age # => "Kendi yaşı: 43, rnd= 44"
```

Örnekte, `SuperUser` Class'ında kafamıza göre `give_random_age` method'unu ezip kendi işlemimizi yaparken, üst sınıftan miras gelen orijinal method'u da yedekliyoruz, `yedik` adı altında.

## Sınıflar Açıktır, Modifiye Edilebilir!



İster Kernel'dan ister başka bir yerden gelsin, her şekilde Class'lar modifiye edilebilir. Detayları **Monkey Patching**'de göreceğiz. Kısa bir örnek yapalım. `String` Class'ına neşemize göre bir method ekleyelim:

```
class String
  def hello
    "Hello: #{self}"
  end
end

"Deneme".hello # => "Hello: Deneme"
```

Tipi `String` olan herşeyin artık `hello` diye bir method'u oldu :)

## Nested Class

Aynı Module'lerde olduğu gibi, iç içe Class tanımlamak da mümkündür. Kimi zaman düzenli olmak için (*Namespace*) kimi zaman da belli bir kuralı uygulamak için kullanılır;

```
class Animal
  attr_reader :name

  def initialize(name)
    @name = name
  end

  class Cat < Animal
  end

  class Horse < Animal
  end

  class Uber
  end
end

horse = Animal::Horse.new "Furry"
horse.name           # => "Furry"
horse.class           # => Animal::Horse
horse.class.superclass # => Animal

cat = Animal::Cat.new "Bıdık"
cat.name              # => "Bıdık"
cat.class              # => Animal::Cat
cat.class.superclass  # => Animal

alien = Animal::Uber.new
alien.respond_to?(:name) # => false
alien.class             # => Animal::Uber
alien.class.superclass  # => Object
```

Cat ve Horse , Animal sınıfından türemiş, Uber ise sadece Animal namespace'i içinde olup kendi başına bir Class'ı temsil etmektedir.

# Module

Class'a benzeyen ama Class gibi **instantiate** edilemeyen şeydir modül. Modül denen şeye Class eklenebilir (*include edilir*) Modülden gelen methodlar artık ilgili Class'ın methodu haline gelir.

Yani düşünün ki bir Class var, bu Class'ın farklı 2-3 Class'tan özellik almasını istiyorsunuz. Bunu başarmak için ilgili Class'a o 2-3 Class'ı Modül olarak ekliyorsunuz!

Module'ler sayesinde **Namespace** ve **Mix-in** fonksiyonallitesi de gelmiş olur.

Tahmin edebileceğiniz gibi `module` kelimesiyle başlarlar ve aynı Class'larda olduğu gibi büyük harfle başlayan module adı ya da Namespace tanımlaması yapılır:

```
module RandomNumbers
  def generate
    rand(10)
  end
end

class DiceGame
  include RandomNumbers
end

class RaceGame
  include RandomNumbers
end

g = DiceGame.new
g.generate # => 3

x = RaceGame.new
x.generate # => 7
```

`RandomNumbers` adında bir Module yaptık, iki farklı Class'ımız var, `DiceGame` ve `RaceGame` diye, `include` ile bu Module'ü 2 farklı Class'a ekledik. Şimdi her iki Class'ın da `generate` adında method'u oldu...

## Namespacing

Module içinde Module tanımlayabilirsiniz. Bu sayede belirlediğiniz Module tanımı altında başka alt Module'ler ve method'lar ekleyebilir, bu sayede tüm fonksiyonalliteyi ortak bir isim altından yürütebilirsiniz:

```
module Framework
  module HttpFunctions
    def self.fetch_url
      "This is url fetcher"
    end
  end
end

Framework::HttpFunctions.fetch_url # => "This is url fetcher"
```

Alt Module'e ulaşmak için `::` kullandık. Aynı kodu şu şekilde de tanımlayabilirdik:

```
module Framework
end

module Framework::HttpFunctions
  def self.fetch_url
    "This is url fetcher"
  end
end

Framework::HttpFunctions.fetch_url # => "This is url fetcher"
```

Bu sayede, başka bir kütüphaneden gelen Module'e ek Module'ler takma şansınız olur. Örneğin [Sinatra](#) için ek bir özellik yapıyorsunuz. Bu durumda;

```
module Sinatra::MyFeature
end
```

Şeklinde kullanabilirsiniz.

## Scope (Kapsama Alanı)

Dikkat ettiyseniz Module'ü kullanırken Class gibi instanciate etmedik. Keza örnekte `self.fetch_url` diye method tanımlaması yaptık. Aslında burada **Singleton** gibi kullandık. Örnekte `fetch_url` methodu için scope olarak `HttpFunctions` vermiş olduk. Yani `fetch_url` sadece `Framework::HttpFunctions.fetch_url` şeklinde erişilebilir oldu.

## Constants (Sabitler)

Module içinde sabit değer tanımlaması da yapmak mümkündür.

```
module A
  SABIT = 5
end

A::SABIT # => 5
```

Eğer **nested** (*iç içe*) yani Module içinde Module yaparsak, sabitlere aşağıdaki gibi erişebiliriz:

```
module A
  SABIT = 5
  module B
    def self.sabit_degeri_ver
      SABIT
    end
  end
end

A::SABIT # => 5
A::B.sabit_degeri_ver # => 5
```

Peki, dışarıda tanımlanmış bir sabit varsa?

```
SABIT = 5 # en dıştaki global
module A
  SABIT = 10 # içerideki
  module B
    def self.sabit_degeri_ver
      "#{::SABIT}, #{SABIT}"
    end
  end
end

A::B.sabit_degeri_ver # => "5, 10"
```

En dıştakini `::SABIT` ile aldık.

## Visibility, Access Level (Erişim):

Aynı Class'lardaki gibi `public`, `private` ve `protected` olayı Module'ler için de geçerlidir.

```

module A
  def sadece_iceriden
    "Bu private method"
  end

  def bu_sayede_private_erisim_olur
    sadece_iceriden
  end

  private :sadece_iceriden
end

class Deneme
  include A
end

c = Deneme.new
c.sadece_iceriden # => NoMethodError: private method 'sadece_iceriden' called for #<Deneme:0x007f8f7c9188c8>
c.bu_sayede_private_erisim_olur # => "Bu private method"

```

## Extend ve Include Durumları

Ruby'de bir Class sadece tek bir Class'tan türeyebildiği için `module` ve `include` çözümlerinden bahsetmiştik:

```

module Person
  attr_accessor :name
  def say_hi
    "Hello #{@name}"
  end
end

Person # => Person

class User
  include Person
  def initialize(name)
    @name = name
  end
end

User # => User

u = User.new("Uğur") # => #<User:0x007fcd42976de8 @name="Uğur">
u.say_hi # => "Hello Uğur"

u.name = "vigo"
u.say_hi # => "Hello vigo"

```

Person modülünden gelen say\_hi method'una;

```
User.new("Ezel").say_hi # => "Hello Ezel"
```

erişebiliyorsunuz ama ;

```
User.say_hi # => undefined method `say_hi' for User:Class (NoMethodError)
```

yaptığımızda olmayan bir method çağırımı yapmış oluruz. Eğer `include` yerine `extend` kullansaydık;

```
module Person
  attr_accessor :name

  def say_hi
    @name ||= "Undefined name"
    "Hello #{@name}"
  end
end

class User
  extend Person

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

```
user = User.new("Yeşim") # => #<User:0x007f87c39702a0 @name="Yeşim">
user.name # => undefined method `name' for #<User:0x007f87c39702a0 @name="Yeşim"> (NoMethodError)
```

Çünkü `Person` a ait özellikleri eklemek (*include*) yerine **extend** (*genişletme*) ettik ve;

```
User.say_hi # => "Hello Undefined name"
User.instance_methods - Object.instance_methods # => [] # boş array
```

`say_hi` artık bir **singleton** haline geldi yani **Instance Method**'u olmak yerine **Class Method**'u oldu. Zaten ilgili sınıfın varolan method'larına baktığımızda boş `Array` döndüğünü görürüz.

Özetle, `include` ile sanki başka bir sınıftan türer gibi tüm özellikleri **instance method** olarak alırken, `extend` kullandığımızda direk **Class** kopyası gibi davranıyor.

Modülden gelen method'ları **Class** içinde **instance method** gibi kullanmak yerine **singleton method** olarak kullanacağınız zaman `extend` kullanabilirsiniz!





# Bölüm 5

Bu bölümde;

- Enumeration ve Iteration
- Ranges
- File System ve IO (*Dosya Sistemi*)
- Exception Handling
- Kernel Modülü

konularını işleyeceğiz.

# Enumeration ve Iteration

Sayılabilen nesneler `Enumerator` sınıfından türemişlerdir ve içinde döngü / tekrar yapılabilen nesneler haline geldikleri için de **Iterable** hale gelmişlerdir. Yani ne demek istiyorum?

```
["a", "b", "c"].class      # => Array
["a", "b", "c"].each.class # => Enumerator
```

`["a", "b", "c"]` aslında bir `Array` iken, `#each` method'unu çağırdığımız anda elimizdeki `Array` birden `Enumerator` haline geldi ve içinde **iterasyon** yapılabilecek yani teker teker dizi içindeki elemanlara erişip istediğimizi gibi kullanabileceğimiz bir hale geldi.

## `each_with_object`, `with_object`

İki method'da aynı işi yapar. Elimizde **Enumerator** varsa, yani bu içinde dolaşılabilen bir nesne ise, bu iterasyona ara elementler takabiliriz.

Bu iki method, `Enumerator`'deki her elemana verilen şeyi takar. Aşağıdaki örnekte `each_with_object("foo")`, `["a", "b", "c"]` dizisindeki her eleman içindir. Dolayısıyla, bu işlem sonrasında ne olduğunu anlamak için `Enumerator` ü `to_a` method'u ile `Array` e çevirdik.

```
enumerator = ["a", "b", "c"].each
enumerator_with_foo = enumerator.each_with_object("foo")
enumerator_with_foo.to_a # => [["a", "foo"], ["b", "foo"], ["c", "foo"]]
```

Keza, bu durumda `enumerator_with_foo` da `each` method'unu kullanarak, `each_with_object` ile pas edilen nesneye de iterasyon esnasına ulaşabiliyoruz;

```
enumerator = ["a", "b", "c"].each
enumerator_with_foo = enumerator.each_with_object("foo")
enumerator_with_foo.each do |element, obj|
  puts "eleman: #{element}, obj: #{obj}"
end

# eleman: a, obj: foo
# eleman: b, obj: foo
# eleman: c, obj: foo
```

Elimizde `Enumerator` varsa, bir şekilde sıra / index bilgisi de var demektir;

```
sayilar = [1, 2, 3, 4].each
sayilar.next # => 1
sayilar.next # => 2
sayilar.next # => 3
sayilar.next # => 4
sayilar.next # => StopIteration hatası!
```

`next` method'unu kullanarak sonraki elemana ulaşabiliyoruz. Hatırlarsanız, **Array**'ler 0 index'lidir. Elimizdeki Array içinde dolaşırken index kullanmak istersek `each_with_index` method'unu kullanırız;

```
["Uğur", "Yeşim", "Ezel", "Ömer"].each_with_index do |name, index|
  puts "İsim: #{name}, index: #{index}"
end
# İsim: Uğur, index: 0
# İsim: Yeşim, index: 1
# İsim: Ezel, index: 2
# İsim: Ömer, index: 3
```

Keza `for` kullanarak da aşağıdaki gibi bir işlem yapabiliriz;

```
isimler = ["Uğur", "Yeşim", "Ezel", "Ömer"]
for isim in isimler
  puts "isim: #{isim}"
end
# isim: Uğur
# isim: Yeşim
# isim: Ezel
# isim: Ömer
```

### `next_values`, `peek`, `peek_values`, `rewind`

Aynı `next` gibi çalışır fakat geriye `Array` döner ve ilgili index pozisyonunu ileri taşır. Sona geldiğinde de `StopIteration` hatası verir (*Exception*)

```
a = [1, 2, 3, 4].each
a.next # => 1
a.next_values # => [2]
a.next_values # => [3]
```

`peek` ile `next` den sonraki değeri görürüz. Eğer sona gelinmişse yine `StopIteration` raise edilir.

```

a = [1, 2, 3, 4].each
a.next      # => 1
a.peek      # => 2

a.next      # => 2
a.peek      # => 3

a.next      # => 3
a.peek      # => 4

a.next      # => 4
a.peek      # => StopIteration: iteration reached an end

```

aynı `next_values` gibi `peek_values` de bize Array olarak bilgi verir `next` sonrasında kalan elemanları.

`rewind` ile pozisyonu başa alırız, yani kaydı geri sararız :)

```

a = [1, 2, 3, 4].each
a.next      # => 1
a.next      # => 2
a.next      # => 3
a.rewind    # => #<Enumerator: [1, 2, 3, 4]:each>
a.peek_values # => [1]
a.next      # => 1

```

## Diğer Nesnelerdeki Enumeration Durumları

### Fixnum

#### **upto, downto, times**

Bir sayıdan yukarı doğru sayarken `upto` , aşağı doğru sayarken `downto` ve kaç defa aynı işlemi yaparken de `times` kullanırız.

```
# 10'dan 5'e sayıyoruz, 10'da 5'de dahil..
10.downto(5){ |i| puts "Sayı: #{i}" }
# >> Sayı: 10
# >> Sayı: 9
# >> Sayı: 8
# >> Sayı: 7
# >> Sayı: 6
# >> Sayı: 5

# 5'den 10'a sayıyoruz, 10'da 5'de dahil..
5.upto(10){ |i| puts "Sayı: #{i}" }
# >> Sayı: 5
# >> Sayı: 6
# >> Sayı: 7
# >> Sayı: 8
# >> Sayı: 9
# >> Sayı: 10

# 3 defa block içindeki kod çalışsın
# 0'dan 3'e kadar 3 hariç :)
3.times{ |i| puts "#{i}" }
# >> 0
# >> 1
# >> 2
```

## String

Aynı mantıkta `upto` ve `downto` ilginç bir şekilde `String` için de kullanılır. Örneğin A'dan itibaren M'ye kadar demek için:

```
"A".upto("M"){ |s| puts s }
# >> A
# >> B
# >> C
# >> D
# >> E
# >> F
# >> G
# >> H
# >> I
# >> J
# >> K
# >> L
# >> M
```

ya da, "AB", "AC", "AD" gibi sekans olarak gitmek gerektiğinde de;

```
"AB".upto("AE"){ |s| puts s }  
# >> AB  
# >> AC  
# >> AD  
# >> AE
```

tam tersi için `downto` kullanılır.

# Ranges

Başı, sonu olan, tanımlanan belli bir aralıktaki değerleri gösteren nesneler `Range` sınıfındadır. Örneğin 0'la 5 arasındaki sayılar **Range** olarak ifade edilebilir;

```
(0..5)          # => 0..5
(0..5).class    # => Range
(0..5).to_a     # => [0, 1, 2, 3, 4, 5]
(0..5).to_a.join # => "012345"
(0..5).each     # => #<Enumerator: 0..5:each>
```

`(0..5)` ifadesinde 0 ve 5 dahil olmak üzere bir aralık tanımladık. İstedığımız gibi işleyebiliriz.

```
(-10..0).to_a    # => [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0]
(0..-10).to_a    # => []
("a".. "e").to_a # => ["a", "b", "c", "d", "e"]
```

Eğer `..` yerine `...` kullanırsak, yani `(0..5)` dersek, 5 hariç demiş oluruz;

```
(0...5).to_a     # => [0, 1, 2, 3, 4]
("a"... "e").to_a # => ["a", "b", "c", "d"]
```

Son değer hariç mi dahil mi anlamak için `exclude_end?` method'unu kullanırız;

```
(5..10).exclude_end? # => false
(5...10).exclude_end? # => true
```

`Range` aslında bir `class` 'dır ve her Class gibi;

```
r = Range.new(0,2) # => 0..2
r.to_a             # => [0, 1, 2]
```

kullanılabilir. `==` ya da `eq1?` method'u ile karşılaştırılabilir;

```
(0..2) == (0..2)      # => true
(0..2).eq1?(0..2)    # => true
(0..2) == (0...2)    # => false
(0..2) == Range.new(0,2) # => true
(0..2).eq1?(Range.new(0,2)) # => true
```

**begin, first, cover?, include?, member?, end, last**

## Ranges

Aralığın başlama değerini almak için `begin` ya da `first` kullanabildiğimiz gibi, `first` 'e parametre geçerek, ilk N değeri de okuyabiliriz;

```
(5..10).begin      # => 5
(5..10).first      # => 5
(5..10).first(2)   # => [5, 6] # ilk 2 değeri ver
```

Belirlediğimiz aralık içinde sorgu yapmak için `cover?` ya da `include?` ya da `member?` method'unu kullanırız. `cover?` kullanırken eğer verdiğimiz değer aralık içindeyse sonuç `true` döner;

```
(5..10).cover?(6)   # => true
(5..10).cover?(4)   # => false
(5..10).cover?(11)  # => false
(5..10).cover?(9)   # => true
("a".. "z").cover?("b") # => true
("a".. "z").cover?("1") # => false
("a".. "z").cover?(1) # => false
("a".. "z").cover?("abc") # => true
```

`include?` ile `cover?` arasındaki fark ise şudur; `cover?` a verilen parametredeki değer, örnekteki `abc` teker teker range içinde var mı? yani `a` var mı? `b` var mı? `c` var mı? şeklinde olurken, `include?` da `abc` var mı? şeklindedir;

```
("a".. "z").cover?("abc") # => true
("a".. "z").include?("abc") # => false

("a".. "z").cover?("a") # => true
("a".. "z").include?("a") # => true
```

Tahmin edeceğiniz gibi, `end` ile son değeri alırız.

```
(5..10).end      # => 10
(5...10).end     # => 10
```

`last` ile de aynı `first` deki gibi son ya da son N değeri okuruz;

```
(5..10).last      # => 10
(5..10).last(3)   # => [8, 9, 10]
```

## min, max, size

`min` ve `max` ile tanımlı aralıktaki en büyük/küçük değeri alırız, `size` bize boyu verir;



## Ranges

```
(5..10).min      # => 5  
(5..10).max      # => 10  
(5..10).size     # => 6
```

**step** ile kaçar kaçar artacağını veririz;

```
r = Range.new(0, 5)  
r.step(2) # => #<Enumerator: 0..5:step(2)>  
r.step(2).to_a # => [0, 2, 4]
```

# File System ve IO (Dosya Sistemi)

## File

`IO` sınıfından özellikler içeren `File` sınıfı, fiziki dosyalarla işlem yapmamızı sağlayan özellikleri sunar bize. Ruby'nin üzerinde çalıştığı işletim sistemine göre de **file permission** yani dosya üzerindeki yetki sistemi de devrededir.

Default olarak gelen Constant'ları:

```
File.constants # => [:Separator, :SEPARATOR, :ALT_SEPARATOR, :PATH_SEPARATOR, :Constants, :Stat, :WaitReadable, :WaitWritable, :EAGAINWaitReadable, :EAGAINWaitWritable, :EWOULDBLOCKWaitReadable, :EWOULDBLOCKWaitWritable, :EINPROGRESSWaitReadable, :EINPROGRESSWaitWritable, :SEEK_SET, :SEEK_CUR, :SEEK_END, :RDONLY, :WRONLY, :RDWR, :APPEND, :CREATE, :EXCL, :NONBLOCK, :TRUNC, :NOCTTY, :BINARY, :SYNC, :DSYNC, :NOFOLLOW, :LOCK_SH, :LOCK_EX, :LOCK_UN, :LOCK_NB, :NULL, :FNM_NOESCAPE, :FNM_PATHNAME, :FNM_DOTMATCH, :FNM_CASEFOLD, :FNM_EXTGLOB, :FNM_SYSCASE]

File::ALT_SEPARATOR # => nil # Bu Ruby'nin çalıştığı platforma özeldir
File::PATH_SEPARATOR # => ":"
File::SEPARATOR # => "/"
File::Separator # => "/"
```

Örneğin Windows'da çalışan Ruby'de `SEPARATOR` ters slash `\` şeklinde gelecektir.

## Public Class Method'ları

### `absolute_path`, `expand_path`, `join`, `split`

String olarak verilen path bilgisini **absolute path**'e çevirir. Eğer ikinci parametre verilmezse **CWD** (*current working directory*) yani o an için içinde çalıştığınız directory bilgisi kullanılır.

```
File.absolute_path("~/") # => "/"
File.absolute_path(".gitignore", "~/") # => "~/../.gitignore"
```

`expand_path` de bir nevi absolute path'e çevirir:

```
File.expand_path("~/../.gitignore") # => "/Users/vigo/.gitignore"
```

Keza daha kompleks path bulma işlerinde de kullanılır. Bu durumda `__FILE__` sabiti hayatımızı kolaylaştırır. O an Ruby script'inin çalıştığı dosyanın path'i `__FILE__` sabitindedir. Örneğin aşağıdaki gibi bir directory yapısı olsa:

```

proje/
├── lib
│   └── users.rb
└── main.rb

```

ve `lib/users.rb` içinden, dışarıda bulunan `main.rb` dosyasının path'ine ulaşmak istesek;

```
File.expand_path("../../main.rb", __FILE__)
```

şeklinde kullanırız.

`join` kullanarak, Ruby'nin çalıştığı işletim sistemine bağlı olarak, `File::SEPARATOR` kullanarak geçilen string'leri birleştiririz:

```
File.join("usr", "local", "bin") # => "usr/local/bin"
```

Dizin ve dosya ayırıştırmasını da `split` ile yaparız:

```
File.split("usr/local/bin/foo") # => ["usr/local/bin", "foo"]
```

### **atime, ctime, mtime**

Dosyaya son erişilen tarihi `atime` ile, dosyada yapılmış olan son değişiklik tarihini de `ctime` ile, son değişiklik zamanını da `mtime` ile alırız.

```

File.atime("/Users/vigo/.gitignore") # => 2014-11-05 11:45:10 +0200
File.ctime("/Users/vigo/.gitignore") # => 2014-08-04 11:33:14 +0300
File.mtime("/Users/vigo/.gitignore") # => 2014-10-29 15:05:15 +0200

```

### **basename, dirname, extname**

Path içinden dosya adını almak için `basename` kullanırız. Eğer parametre olarak atacağımız şeyi (örneğin *extension* olarak `.gif`, `.rb` gibi) geçerse bize sadece dosyanın adını verir.

```

File.basename("/Users/vigo/test.rb") # => "test.rb"
File.basename("/Users/vigo/test.rb", ".rb") # => "test"

```

Bu işin tersini de `dirname` ile yaparız, yani directory adı gerekince:

```
File.dirname("/Users/vigo/test.rb") # => "/Users/vigo"
```

şekinde kullanırız. Dosyanın extension'ını öğrenmek için `extname` kullanırız.

```
File.extname("test_file.rb")      # => ".rb"
File.extname("/foo/bar/test_file.rb") # => ".rb"
File.extname("test_file")         # => ""
```

## chmod, chown, lchmod, lchown

Her iki komut da Unix'den gelir. **Change mod** ve **Change owner** işlerini yapmamızı sağlar.

`chmod` ile Unix izinlerini ayarlarız:

```
-rw-r--r-- 1 vigo wheel 0 Aug 30 19:19 file-01.txt
|||||
|||||+--- Others, Execute    (x) 1 = 2^0
|||||+---- Others, Write     (w) 2 = 2^1
|||||+----- Others, Read    (r) 4 = 2^2
|||||+----- Group, Execute   (x) 1 = 2^0
|||||+----- Group, Write     (w) 2 = 2^1
|||||+----- Group, Read      (r) 4 = 2^2
|||+----- Owner/User Execute (e) 1 = 2^0
||+----- Owner/User Write    (w) 2 = 2^1
|+----- Owner/User Read      (r) 4 = 2^2
+----- Is Directory?        (d)
```

`file-01.txt` dosyasında, User (yani dosyanın sahibi) **Read** ve **Write** hakkına sahiptir. Group ve Others ise sadece **Read** hakkına sahiptir. Bu durumda varolan bu dosyanın **chmod** değeri:

```
Owner/User : Read, Write => 4 + 2 = 6
Group      : Read        => 4      = 4
Others     : Read        => 4      = 4
-----
644 unix file permission
```

şeklinde. Hatta Terminal'den; `stat -f '%A' file-01.txt` yaparsak **644** olduğunu da görebiliriz. Şimdi bu dosyayı Ruby ile sadece sahibi tarafından okunur ve yazılır yapıp, başka hiçbir kimse tarafından okunamaz ve yazılamaz hale getirelim:

```
File.chmod(0600, "file-01.txt")
```

Keza dosyanın sahibini de düzenlemek için `chown` kullanırız. Aynı terminaldeki gibi **KULLANICI:GRUP** şeklinde, toplamda üç parametre geçeriz. İlki kullanıcıyı belirler. `nil` ya da `-1` geçtiğimiz taktirde ilgili şeyi set etmemiş oluruz. Yani sadece grubu değiştireceksek kullanıcı için `nil` ya da `-1` geçebiliriz.

```
File.chown(nil, 20, "/tmp/file-01.txt") # => 1
```

Grup ID olarak 20 geçtik, OSX'deki id 20 karşılık olarak **staff** grubuna denk gelir.

`lchmod` ve `lchown` ile normal `chmod` , `chown` farkı, `1` ile başlayanlar sembolik linkleri takip etmezler.

### **ftype, stat, lstat, size**

Dosyanın ne tür bir dosya olduğunu `ftype` ile anlarız:

```
File.ftype("/tmp/file-01.txt") # => "file"
File.ftype("/usr/")            # => "directory"
File.ftype("/dev/null")       # => "characterSpecial"
```

`stat` ile aynen biraz önce shell'den yaptığımız (`stat -f '%A' file-01.txt`) gibi aynı işi Ruby'den de yapabiliriz:

```
File.stat("/tmp/file-01.txt")      # => #<File::Stat dev=0x10000004, ino=1540444, mode
=0100600, nlink=1, uid=501, gid=20, rdev=0x0, size=4, blksize=4096, blocks=8, atime=20
14-11-12 14:41:49 +0200, mtime=2014-11-12 14:40:13 +0200, ctime=2014-11-12 14:45:28 +0
200>
File.stat("/tmp/file-01.txt").uid  # => 501
File.stat("/tmp/file-01.txt").gid  # => 20
File.stat("/tmp/file-01.txt").mtime # => 2014-11-12 14:40:13 +0200
```

`lstat` da aynı işi yapar fakat aynı `lchmod` ve `lchown` daki gibi sembolik linkleri takip etmez!

Dosyanın byte cinsinden büyüklüğünü almak için `size` kullanırız:

```
File.size("/Users/vigo/.gitignore") # => 323
```

### **delete, unlink, link, rename, readlink, symlink**

Her ikisi de dosya silmeye yarar. Eğer dosya başarıyla silinirse `1` döner, aksi halde hata alırız!

```
File.delete("/tmp/foo.txt") # => 1 yani silindi
File.delete("/tmp/foo1.txt") # => No such file or directory
```

`link` ile **HARD LINK** oluşturuyoruz. Bunu dosyanın bir kopyası / yansıması gibi düşünebilirsiniz. Orijinal dosya değiştiğinde linklenmiş dosya da güncel içeriğe sahip olur. Link'in hangi dosyaya bağlı olduğunu da `readlink` ile okuruz:

```
File.link("orijinal_dosya", "linklenecek_dosya")
File.readlink("linklenecek_dosya") # => "orijinal_dosya"
```

Sembolik link yani `symlink` için;

```
File.symlink("foo.txt", "bar.txt") # => 0
File.readlink("bar.txt")          # => "foo.txt"
```

Komut satırından bakınca;

```
-rw-r--r--  1 vigo wheel   0 Dec 13 16:08 foo.txt
lrwxr-xr-x  1 vigo wheel   7 Dec 13 16:08 bar.txt -> foo.txt
```

şeklinde `bar.txt` dosyasının `foo.txt` dosyasına linklendiğini görürüz.

Dosya ismini değiştirmek için `rename` kullanırız.

```
File.rename("/tmp/file-01.txt", "/tmp/file-01.txt.bak") # => 0
```

**file?**, **directory?**, **executable?**, **exist?**, **identical?**, **readable?**, **size?**,

Dosya gerçekten fiziki bir dosya mı? ya da directory mi? ya da bu dosya var mı?

```
# file?
File.file?("/tmp/file-01.txt") # => true (evet)
File.file?("/tmp/file-02.txt") # => false

# directory?
File.directory?("/tmp/file-02.txt") # => false
File.directory?("/tmp/test_folder") # => true (evet)

# dosya var mı?
File.exist?("/tmp/file-01.txt") # => true (var)
File.exist?("/tmp/file-02.txt") # => false
```

Daha önce dosya izinlerinden bahsetmiş, bazı dosyaların **executable** olduğunu söylemiştik. Acaba dosya çalıştırılabilir yani executable mı?

```
File.executable?("/tmp/file-01.txt") # => false
-rw-r--r--  1 vigo wheel   0 Nov 15 10:39 file-01.txt

File.executable?("/tmp/executable_file") # => true
-rwxr-xr-x  1 vigo wheel   0 Nov 15 10:43 executable_file
```

Executable olan dosyada `x` flag aktif gördüğünüz gibi :)

**new**, **open**

`open` method'u ile `new` aynı işi yapar. Dosya açmaya yarar. Dosyayı açarken hangi duruma göre açacağımızı yani okumak için mi? yazmak için mi? yoksa varolan dosyaya ek yapmak için mi? belirtmemiz gerekir.

```
f = File.new("/tmp/test.txt", "w")
f.puts "Merhaba"
f.close
```

/tmp/test.txt adlı bir dosya oluşturup içine puts ile **Merhaba** yazdık. Eğer cat /tmp/test.txt yaparsanız kontrol edebilirsiniz. Dikkat ettiyseniz mode olarak "w" kullandık. Bu mode'lar neler?

Mode	Açıklama
r	Read-only, sadece okumak için. Bu default mode'dur.
r+	Read+write, hatta read + prepend, pointer'ı başa alır, yani bu method'la bişi yazarsanız, yazdığınız şey dosyanın başına eklenir.
w	Write-only, sadece yazmak içindir. Eğer dosya yoksa hata verir!
w+	Read+Write, hatta read + append, pointer'ı dosyanın sonuna alır ve yazdıklarınızı sona ekler. Eğer dosya yoksa hata verir!
a	Write-only, Eğer dosya varsa pointer'ı sona alır ve sona ek yapar, dosya yoksa sıfırdan yeni dosya üretir.
a+	Read+write, Aynı a gibi çalışır, sona ek yapar, hem okumaya hem de yazmaya izin verir.
b	Binary mode
t	Text mode

### fnmatch, fnmatch?

**File Name Match** yani dosya adı eşleştirmek. RegEx pattern'ine göre dosya adı yakalamak / kontrol etmek için kullanılır. 2 zorunlu ve 1 opsiyonel olmak üzere 3 parametre alabilir. Pattern, dosya adı ve opsiyonel olarak Flag'ler...

```
File.fnmatch('foo', 'foobar.rb') # => false
File.fnmatch('foo*', 'foobar.rb') # => true
File.fnmatch('*foo*', 'test_foobar.rb') # => true
```

Şimdi;

```
File.fnmatch('*.rb', './main.rb') # => false
```

Bu işlemin true dönməsi için FNM\_DOTMATCH flag'ini kullanacağız:

```
File.fnmatch('*.rb', './main.rb', File::FNM_DOTMATCH) # => true
```

0:0	1:0
FNM_DOTMATCH	Nokta ile başlayan dosyalarda * kullanımına izin ver
FNM_EXTGLOB	{a,b,c} gibi paternlerde global aramaya izin ver
FNM_PATHNAME	Path ayraçlarında * kullanımını engelle
FNM_CASEFOLD	Case in-sensitive yani büyük/küçük harf ayırt etme!
File::FNM_NOESCAPE	ESCAPE kodu kullan

```
File.fnmatch('*', '/', File::FNM_PATHNAME)      # => false
File.fnmatch('F00*', 'foo.rb', File::FNM_CASEFOLD) # => true
File.fnmatch('f{o,a}o*', 'foo.rb', File::FNM_EXTGLOB) # => true
File.fnmatch('f{o,a}o*', 'fao.rb', File::FNM_EXTGLOB) # => true
File.fnmatch('\foo*', '\foo.rb')                # => false
File.fnmatch('\foo*', '\foo.rb', File::FNM_NOESCAPE) # => true
```

## IO

Tüm giriş/çıkış (Input/Output) işlerinin kalbi burada atar. `File` sınıfı da `IO` 'nun alt sınıfıdır. Binary okuma/yazma işlemleri, multi tasking işlemler (*Process spawning, async işler*) hep bu sınıf sayesinde çalışır.

Ruby 101 seviyesi için biraz karmaşık olsa dahi, sadece fikriniz olması açısından, en bilinen ve kullanılan birkaç method'a değinmek istiyorum.

### binread

Binary read, yani byte-byte okuma işlemi için kullanılır. Opsiyonel olarak geçilen 2.parametre, kaç byte okumak istediğimizi, 3.parametre de offset yani kaç byte öteden okumaya başlamak gerek bunu bildirir. Yani elinizde bir dosya olsun, dosyanın ilk 100 byte'ını 20.byte'tan itibaren okumanız gerekirse kullanacağınız method budur :)

```
# 1 byte atlayarak 3 byte okuduk ve
IO.binread("test.png", 3, 1) # => "PNG"
```

### binwrite

Tahmin edeceğiniz gibi `binread` in tersi, yani Binary olarak yazma işini yapan method. Aynı şekilde opsiyonel 2 ve 3.parametreleri kullanabilirsiniz.

### copystream

Birebir kopya yapmaya yarar. İlk parametre SOURCE yani neyi kopyalayacaksınız, ikinci parametre DESTINATION yani nereye kopyalayacaksınız, eğer kullanırsanız 3.parametre kopyalanacak byte adedi, eğer 4.parametre kullanırsanız aynı read/write daki gibi offset değeri olarak kullanabilirsiniz.



**foreach**

Elimizde `test-file.txt` olsun ve içinde;

```
satır 1
satır 2
```

yazsın... Satır-satır içinde dolaşmak için;

```
IO.foreach("test-file.txt"){|x| print "bu satır: ",x}
```

Dediğimizde çıktı;

```
bu satır: satır 1
bu satır: satır 2
```

şeklinde içeriye block pas edip kullanabiliriz.

**popen**

Subprocess yani alt işlemler açmak için kullanılır. Özellikle Ruby üzerinden SHELL komutları çağırmak için çok kullanılan bir yöntemdir. Asenkron işler.

```
# Bu işlem asenkron/alt işlem olarak çalışır...
IO.popen("date") do |response|
  system_date = response.gets
  puts "system_date: #{system_date}"
end
```

`/tmp/` dizinini listeleyelim:

```
p = IO.popen("ls /tmp/")
p.pid      # => 52389
p.readlines # => ["D8D75028-234B-4F49-9358-C4C4775B4A08_IN\n", "D8D75028-234B-4F49-9358-C4C4775B4A08_OUT\n", "F7C71944B49B446081C0603DE90E4855_IN\n", "F7C71944B49B446081C0603DE90E4855_OUT\n", "KSOutOfProcessFetcher.501.0laJUhhgKAnFsX7fZ0FyXTFxIgg=\n", "com.apple.launchd.4H4RVax25p\n", "com.apple.launchd.Kn8Wcx4NQX\n", "fo\n", "lilo.12159\n", "swtag.log\n", "test-file.txt\n"]
```

Gördüğümüz gibi **pid** yani Process ID : 52389, eğer shell'den;

```
ps ax | grep 52389
```

derseniz;

```
52389  ??  Z      0:00.00 (ls)
```

gibi ilgili işlemi görürsünüz.

# Exception Handling

# Kernel Modülü

## Bölüm 6

Bu bölümde;

- [Monkey Patching](#)
- [Regular Expressions](#)
- [Time ve Date Nesneleri](#)
- [Ruby Paketleri: RubyGems](#)
- [Paket Yöneticisi: Bundler](#)
- [Komut Satırı \(Command-Line\) Kullanımı](#)
- [Meta Programming](#)

konularını işleyeceğiz

# Monkey Patching

Ruby'nin en şaibeli özelliklerinden biridir. Kimileri için müthiş bir şey kimileri için de çok tehlikeli bir özelliktir. 7.7'de bahsettiğim **Meta Programming** konusu ile de çok yakından alakalıdır.

Ruby'deki tüm sınıflar açıktır. Yani Kernel'dan gelen herhangi bir sınıfı modifiye etmek mümkündür. Bu durum yanlış ellerde çok tehlikeli olabilir.

Yani, `String` sınıfındaki herhangi bir method'u bozmak mümkündür. Örneğin, `String#length` methodunu değiştirelim:

```
"Hello".length # => 5 # Bu normali

# Monkey Patching yapıyoruz ve length method'unu değiştiriyoruz.
class String
  def length
    "Uzunluk: #{self.size} karakterdir."
  end
end

"Hello".length # => "Uzunluk: 5 karakterdir."
```

Normal şartlar altında `length` methodu `Fixnum` dönmesi gerekirken, bozduğumuz method bize `String` döndü. Anlatabilmek için bu denli abartı bir örnek vermek istedim.

Düşünsenize, kullandığınız herhangi bir kütüphane, kafasına göre, standart olan herhangi bir method'u bu şekilde bozsa?

Tüm kodunuz çorbaya döner ve içinden çıkamaz bir hale gelir. Peki asıl kullanım amacı bu mudur? Tabii ki değil. Bize kolaylık sağlayan işlerde kullanmamız gerekiyor.

Örneğin, basit bir matematik işlemi için, **5 kere 5** önermesini kullanmak istiyoruz:

```
class Fixnum
  def kere(n)
    self * n
  end
end
```

`Fixnum` içine `kere` diye bir method taktık. Haydi kullanalım:

```
5.kere(5)          # => 25
5.kere(5).kere(2)  # => 50
```

İşte bu tür bir **Monkey Patching** işe yarar ve kullanılabilirliği yüksek olan bir yöntemdir. Keza Ruby on Rails webframework'ü neredeyse bu mantık üzerine kurulmuştur.

Örneğin **5 gün önce** şeklinde bir önerme yapmak istiyoruz.

```
class Fixnum
  def gün
    self * 24 * 60 * 60
  end

  def önce
    Time.now - self
  end

  def sonra
    Time.now + self
  end
end
```

Şimdi şöyle birşey yapalım:

```
Time.now      # => 2015-02-09 12:55:33 +0200
5.gün.önce    # => 2015-02-04 12:55:33 +0200
1.gün.sonra   # => 2015-02-10 12:55:33 +0200
```

`Fixnum` yani basit sayılara `.gün.önce` ve `.gün.sonra` gibi iki tane method ekledik :)

# Regular Expressions

wip...



# Time ve Date Nesneleri

Zaman, tarih ve saat gibi işlemleri yapmak için Ruby ile birlikte gelen `Time` ve `Date` sınıflarından bahsedeceğim. En basit tanımıyla saatin kaç olduğunu ya da `now` yani **şimdi** / **şu anda** yı bulmak için:

## Time

```
Time.now # => 2015-04-28 08:49:20 +0300
```

yapmamız yeterlidir. Saniye bazında ekleme ya da çıkartma yaparak başka zamanları da bulabiliriz. **1 saat önceyi** bulmak için **60 saniye \* 60 dakika** yapmamız yeterli.

```
Time.now - (60 * 60) # => 2015-04-28 07:51:01 +0300
```

Hemen Ruby'sel bir hareketle ufak birşey yapalım:

```
class Fixnum
  def seconds
    self
  end
  def minutes
    self * 60
  end
  def hours
    self * 60 * 60
  end
  def days
    self * 60 * 60 * 24
  end
end

# 10 gün sonrayı bulalım
Time.now + 10.days # => 2015-05-08 08:54:02 +0300
```

Saat farkları yüzünden çeşitli **Time Zone**'lar mevcut. An itibariyle (28 Nisan 2015) yerel zamana baktığımda:

```
Time.local(2015, 4, 28, 8, 54) # => 2015-04-28 08:54:00 +0300
```

Peki **GMT**'ye göre durum ne?

```
Time.gm(2015, 4, 28, 8, 54) # => 2015-04-28 08:54:00 UTC
```

Şimdi kendi istediğimiz bir zamanı oluşturalım. Doğduğum yıl ve ayı kullanarak bir zaman oluşturup kabaca hangi güne denk geldiğini bulalım. Sadece **YIL** ve **AY** kullanıyoruz!

```
t = Time.new(1972, 8) # => 1972-08-01 00:00:00 +0300
t.monday? # => false
t.tuesday? # => true
```

Ruby bize otomatik olarak ayın birini işaret etti ve 1 Ağustos 1972'nin salı gününe denk geldiğini `tuesday?` method'u ile anladık.

Tahmin edebileceğiniz gibi, İngilizce olarak, günleri kontrol edebiliyoruz. Yani `sunday?` , `monday?` ... gibi

Eğer UNIX'in epoch zaman formatında istersek, yapmamız gereken `to_i` method'u ile integer'a çevirmek:

```
Time.new(1972, 8).to_i # => 81464400
```

Eğer elimizde epoch cinsinden bir zaman varsa:

```
Time.at(81464400) # => 1972-08-01 00:00:00 +0300
Time.at(81464400).year # => 1972
```

şeklinde de kullanabiliriz. Bunlara ek olarak;

```
Time.now.zone # => "EEST"
Time.now.day # => 29
Time.now.wday # => 3 # çarşamba
Time.now.utc? # => false
Time.now.gmt? # => false
```

Zamanlar arasındaki farkı bulmak için de aynı matematik işlemi gibi yaparmış gibi davranabilirsiniz.

```
birth_day = Time.new(1972, 8) # Ağustos 1972
Time.now.to_i # => 1430284388
birth_day.to_i # => 81464400

Time.now.to_i - birth_day.to_i # => 1348819988 saniye
1348819870 / (60 * 60 * 24) # => 15611 gün
1348819870 / (60 * 60 * 24 * 30) # => 520 ay
1348819870 / (60 * 60 * 24 * 30 * 12) # => 43 yıl
```

Aynı şekilde karşılaştırma işlemleri de matematik işlemleri gibi. Doğduğum yıl ile bugünü karşılaştıralım:

```
birth_day = Time.new(1972, 8)
now = Time.now
tomorrow = now + (60 * 60 * 24)

now.to_i           # => 1430284645
tomorrow.to_i      # => 1430371045
birth_day.to_i     # => 81464400

now.to_i > birth_day.to_i    # => true      bugün > doğum tarihi
now.to_i > tomorrow.to_i    # => false     bugün < yarın
tomorrow.to_i > now.to_i     # => true      yarın > bugün
```

## Zamanı Formatlı Şekilde Göstermek

Tarih bilgisini istediğimiz şekilde format ederek çıktı alabiliriz.

```
t = Time.now
t.strftime("Bugün %d %B %Y, %A, saat: %H:%M") # => "Bugün 01 May 2015, Friday, saat: 12:35"
```

Dikkat ettiyseniz İngilizce olarak çıktıyı aldık. Ne yazık ki Ruby'de **locale** kavramı yok. Bu yüzden Türkçe çıktı almak için `i18n` gem'ini kullanmamız gerekiyor.

```
gem install i18n
```

Daha sonra herhangi bir dizin altında;

```
cd ~
mkdir i18n-works
cd i18n-works/
mkdir locales
cd locales/
curl -O https://raw.githubusercontent.com/svenfuchs/rails-i18n/master/rails/locale/tr.yml
cd ../../
touch run.rb
```

şimdi `run.rb` dosyası içine;

```
require 'i18n'

I18n.load_path = Dir['./locales/*.yaml']
I18n.locale = :tr
puts I18n.locale
t = Time.now
puts I18n.localize t, :format => "Bugün %d %B %Y, %A, saat: %H:%M"
```

yaptığımızda çıktı:

```
tr
Bugün 01 Mayıs 2015, Cuma, saat: 12:55
```

şeklinde olacaktır.

Kullanımda: `%<flags><width><modifier><conversion>` şeklinde bir yöntem bulunmakta.

## Flag'ler

```
- don't pad a numerical output
_ use spaces for padding
0 use zeros for padding
^ upcase the result string
# change case
: use colons for %z
```

Hemen örneklerle görelim, ilk olarak `-`, `_` ve `0` kullanımına bakalım:

```
t = Time.now      # => 2015-05-02 11:35:26 +0300
t.strftime("%d")  # => "02"

# şimdi - ile yapıyoruz
t.strftime("%-d") # => "2"   # 0'la doldurmadı...

# şimdi _ ile yapıyoruz
t.strftime("%_d") # => " 2"  # SPACE karakteri ile doldurdu...

# şimdi 0 ile yapıyoruz
t.strftime("%0d") # => " 02" # 0 ile doldurdu...
```

Şimdi `^`, `#` ve `:` flag'lerine bakalım:

```
t.strftime("%A")      # => "Saturday"
t.strftime("%^A")     # => "SATURDAY" # upcase yaptı
t.strftime("%#A")     # => "SATURDAY" # change case demek, upcase ise down, downcase
ise up yapmak demek.

                        # saturday downcase geldi, upcase oldu
t.strftime("%z")      # => "+0300"
t.strftime("%:z")     # => "+03:00" # : ile ayırdı
```

**width**

```
t.strftime("%d")    # => "02"
t.strftime("%10d")   # => "0000000002" # 10 basamak yaptı.
```

**Formatlama**

İşaret	Açıklama
%Y	4 dijital yıl
%C	yıl/100, yüzyıl için
%y	yıl mod 100, 2015 için <b>15</b> gelir.
%m	Yılın ayı. (01..12)
%B	Ayın tam adı (January)
%b ya da %h	Ayın kısa adı (Jan)
%d	Ayın günü, 0 eklemeli (01..31)
%e	Ayın günü, SPACE karakteri eklemeli ( 1..31)
%j	Yılın günü (001..366)
%H	Saat, 24-saat formatında 0 eklemeli (00..23)
%k	Saat, 24-saat formatında SPACE karakteri eklemeli ( 0..23)
%I	Saat, 12-saat formatında 0 eklemeli (01..12)
%l	Saat, 12-saat formatında SPACE karakteri eklemeli ( 1..12)
%P	Meridyen göstergesi, küçük harf ( am ya da pm )
%p	Meridyen göstergesi, büyük harf ( AM ya da PM )
%M	Dakika (00..59)
%S	Saniye (00..60)
%L	Milisaniye (000..999)
%N	Kesirli saniye, varsayılan 9 dijital
%z	saat ve dakika offsetli UTC zaman kuşağı (time zone)
%Z	Zaman kuşağının harfsel karşılığı
%A	Haftanın günü, tam yazım
%a	Haftanın günü, kısa yazım
%u	Haftanın kaçınıcı günü, Pazartesi 1
%w	Haftanın kaçınıcı günü, Pazar 0
%c	"%a %b %e %T %Y" şeklinde
%D	"%m/%d/%y" şeklinde

%F	ISO 8601 - "%Y-%m-%d"
%v	VMS - "%e-%^b-%4Y"
%x	%D ile aynı
%X	%T ile aynı
%r	12 saat cinsinden saat "%l:%M:%S %p"
%R	24 saat cinsinden saat "%H:%M"
%T	23 saat cinsinden saat "%H:%M:%S"

## Örnek

```

t = Time.now          # => 2015-05-16 14:29:43 +0300
t.strftime("%N")      # => "691659000"
t.strftime("%3N")     # => "691" # milisaniye, 3 dijit
t.strftime("%6N")     # => "691659" # mikrosaniye, 6 dijit

t.strftime("%Z")      # => "+0300"
t.strftime("%Z")      # => "EEST"
# Eastern European Summer Time yani
# yaz saati :)

t.strftime("%A")      # Sunday
t.strftime("%a")      # Sun
t.strftime("%u")      # "7"
t.strftime("%w")      # "0"

```

@wip

# Ruby paketleri: RubyGems

Ruby, benzeri diğer dillerdeki gibi kendine ait bir paket yöneticisine sahiptir. Ruby paketlerine **Gem** denir. Gem'ler aslında tekrar tekrar kullanılacak Ruby kodlarının paketlenmiş halleridir ve herhangi bir Ruby uygulamasına çok kolay entegre edilebilir.

Genelde tüm paketler <http://rubygems.org> sitesinde sunulur. İster local (*yere/*) ister özel repository isterseniz de RubyGems sitesinden bu paketleri kurabilirsiniz.

Ruby kurulumunda `gem` adında bir komut eklenir sisteme. Eğer `gem --help` dersanız, ilgili kullanımları ve komutları listeleyebilirsiniz.

RubyGems, efsane isim [Jim Weirich](#) tarafından yazılmıştı. Kendisi 2014 Şubat'ta aramızdan ayrıldı.

Herhangi bir paketi kurmak için `gem install PAKET_ADI` şeklinde yazmak yeterli fakat genelde Ruby'nin kurulu olduğu yer sistem dosyalarının kurulu olduğu yerde olduğu için eğer Ruby versiyon yöneticisi (rvm ya da rbenv) kullanmıyorsanız `sudo` ile işlem yapmanız gerekir:

```
sudo gem install PAKET_ADI .
```

Tavsiyem [Rbenv](#) ya da [RVM](#) gibi bir paket yöneticisi kullanmanız. Bir sonraki bölümde göreceğimiz [Bundler](#) aracıyla hem sisteminizi gereksiz gem'lerden korumuş olacağız hem de istediğimiz projede istediğimiz gem versiyonunu kullanmış olacağız.

# Paket yöneticisi: Bundler

wip...



# Komut Satırı (Command-Line) Kullanımı

wip...

# Meta Programming

Ruby'deki **Meta Programming**, yazılan kodun **run-time**'da pek çok şeyi değiştirmesi, Kernel'dan gelen sistem fonksiyonlarını manipule etmesi (*Class, Module, Instance ile ilgili şeyler*) şeklindedir.

Hatta bazen yazdığınız programı restart etmeden bile kodu değişikliği yapmak mümkün olur.

Bazı komutlar, kullanım şekilleri gerçekten de çok tehlikeli olabilir! Özellikle dış dünyadan gelecek input'ların run-time'da yorumlanması pek de önerilen bir yöntem değildir. Yani burada göreceğimiz bazı yöntemleri bilelim ama gerçek dünyada pek fazla **uygulamayalım!**

## Class'lar Değiştirilebilir!

İster Kernel ister dışarıdan eklenen, her tür Class modifiye edilebilir:

```
class String
  def foo
    "foo: #{self}"
  end
end

a = "hello"
a.foo # => "foo: hello"
```

`String` Class'ına kafamıza göre `foo` method'u ekledik.

## Class'ların Birden Fazla `initialize` Yöntemi Olabilir!

Bu `Class Overloading` yani Class ya da method'u ezmek olarak düşünülebilir. Class'ın bir tane `initialize` method'u olduğu için, koşullu olarak Class'a başlangıç seviyesinde müdahale edebiliriz:

```
# Dortgen.new([sol_ust_x, sol_ust_y], boy, en)
# Dortgen.new([sol_ust_x, sol_ust_y], [sag_alt_x, sag_alt_y])
class Dortgen
  def initialize(*args)
    if args.size < 2 || args.size > 3
      "Bu sınıf en az 2 en fazla 3 parametre alır"
    else
      "Doğru parametre kullanımı"
    end
  end
end
Dortgen.new([0, 0], 10, 10) # => #<Dortgen:0x007fe3ea1330f0>
Dortgen.new([0, 0], [10, 10]) # => #<Dortgen:0x007fe3ea132d58>
```

İster 2, ister 3 parametre ile `initialize` ettiğimiz `Dortgen` sınıfı, parametre kullanımına göre farklı çıktılar üretebilir.

## Anonim Class

Anonim Class'lar, **Singleton**, **Ghost** ya da **Metaclass** diye de adlandırılır. Aslında her Ruby Class'ı kendine ait anonim bir sınıfa ve method'lara sahiptir. Tek farkı kendisine ait olmasıdır.

```
class Developer
  class << self
    def personality
      "Awesome"
    end
  end
end

Developer.new # => #<Developer:0x007fc9048a2738>
Developer.personality # => "Awesome"

a = Developer.new
a # => #<Developer:0x007fc9048a2120>
a.class # => Developer
a.class.personality # => "Awesome"

a.personality # => undefined method `personality' for #<Developer:0x007fd3ca0a0e10> (NoMethodError)
```

`Developer` sınıfının kendi `class` 'ına anonim bir method taktık. Class'dan instance üretmeden `Developer.personality` şeklinde erişebilirken, `a` instance'ından gitmek istediğimizde yani `a.personality` dediğimizde hata mesajı aldık. Oysa o methods sadece `a.class` `a` ait :)

Yaptığımız iş aslında bir **Singleton** oluşturma oldu.

### define\_method

Class içinde **run-time** yani dinamik olarak method oluşturabilirsiniz:

```
class Developer
  define_method :personality do |arg|
    "You are #{arg} developer!"
  end
end

Developer.new.personality("an awesome") # => "You are an awesome developer!"
a = Developer.new
a.personality("an awesome") # => "You are an awesome developer!"
a.class.instance_methods(false) # => [:personality]
```

### send

`send` method'u `Object` sınıfından gelen bir method'dur. Sınıfa göndereceğimiz mesaj ilk parametre olup bu da aslında çağıracağımız method adıdır.

```
class Developer
  def hello(*args)
    "Hello #{args.join(" ")}"
  end
end

d = Developer.new
d.send(:hello, "vigo", "how are you?") # => "Hello vigo how are you?"
```

Unutmayın, sadece `public` method'lara erişebilirsiniz!

### remove\_method ve undef\_method

Adından da anlaşılacağı gibi method'u yoketmek için kullanılır ama eğer `remove_method` ile iptal edilmek istenilen method, türediği üst sınıfında var ise ne yazık ki yok edilemez. Bu durumda da `undef_method` devreye girer:

```
class Developer
  def method_missing(m, *args, &block)
    "#{m} is not available!"
  end

  def hello
    "Hello from class Developer"
  end
end

class TurkishDeveloper < Developer
  def hello
    "Hello from class TurkishDeveloper"
  end
end

d = TurkishDeveloper.new
d.hello # => "Hello from class TurkishDeveloper"

class TurkishDeveloper
  remove_method :hello
end

d.hello # => "Hello from class Developer" # üst sınıfta varolduğu için çalıştı!
```

Eğer;

```
class TurkishDeveloper
  undef_method :hello
end

d.hello # => "hello is not available!"
```

yaparsak, method komple uçar ve `method_missing` ile yakaladığımız kod bloğu çalışır.

### eval

Pek çok programlama dilinde **evaluate** etmekten gelen, yani `String` formundaki metnin çalışabilir kod parçası haline gelmesi olayıdır `eval` :

```
eval("5 + 5")          # => 10
eval("'Hello'.downcase') # => "hello"
```

Aslında çok tehlikelidir. Yani programatik hiçbir kontrol olmadan dümdüz metnin **executable** hale getirilmesidir ve hiçbir zaman önerilmez. Güvenlik zafiyeti doğurabilir.

### instance\_eval

Yazılan kod bloğunu sanki Class'ın bir method'uymuş gibi çalıştırır:

```
class Developer
  def initialize
    @star = 10
  end
end

d = Developer.new
d.instance_eval do
  puts self
  puts @star
end

# #<Developer:0x007fe549a91e70>
# 10
```

ya da;

```
class Developer
end

Developer.instance_eval do
  def who
    "vigo"
  end
end

Developer.who # => "vigo"
```

şeklinde kullanılır. Aynı şekilde sadece `public` olan method'lar için geçerlidir.

### **module\_eval** ve **class\_eval**

İkisi de aynı işi yapar. Dışarıdan Class değişkenlerine erişmek için kullanılır:

```
class Developer
  @@geek_rate = 10
end

Developer.class_eval("@@geek_rate") # => 10
```

Aynı şekilde method tanımlamak için;

```
class Developer
end

Developer.class_eval do
  def who
    "vigo"
  end
end

Developer.new.who # => "vigo"
```

### **class\_variable\_get** ve **class\_variable\_set**

Class konusunda Class ve Instance Variables arasındaki farkı görmüştük. Bu iki method yardımıyla sınıf değişkenine erişmek ve değerini değiştirmek mümkün:

```
class Developer
  @@geek_rate = 10
end
Developer.class_variable_set(:@@geek_rate, "100") # => "100"
Developer.class_variable_get(:@@geek_rate)        # => "100"
```

### instance\_variable\_get ve instance\_variable\_set

Aynı önceki gibi, bu method'lar da sadece Instance Variable için çalışır:

```
class Developer
  def initialize(name, star)
    @name = name
    @star = star
  end

  def show
    "Name: #{@name}, Star: #{@star}"
  end
end

d = Developer.new("vigo", 10)
d.instance_variable_get(:@name) # => "vigo"
d.instance_variable_get(:@star) # => 10
d.show # => "Name: vigo, Star: 10"

d.instance_variable_set(:@name, "lego") # => "lego"
d.show # => "Name: lego, Star: 10"
```

### const\_get ve const\_set

Constant yani sabitleri Class ve Module konusunda görmüştük. `const_set` ile Class'a sabit değer atıyoruz, `const_get` ile de ilgili değeri okuyoruz:

```
class Box
end

Box.const_set("NAME", "web") # => "web"
Box.const_get("NAME")       # => "web"
Box::NAME                   # => "web"

a = Box.new
a.class.constants           # => [ :NAME ]
a.class::NAME               # => "web"
```

# Kod Yazma Tarzı (Style Guide)

Aşağıdaki kurallar, kodun doğru çalışmasından ziyade, kullanıcı tarafından doğru okunup algılanması için düşünülmüş, kabul edilmiş kurallardır.

Ruby'e ait resmi bir durum olmasa da, genelde tüm kullanıcılar bu kurallara uymaya çalışır.

Bu kuralları ben [GitHub](#)'dan aldım.

- **soft-tabs** yani `TAB` karakteri yerine **2 adet space** karakteri ile girinti yapılmalı
- Mümkünse satır uzunluğu **80** karakteri geçmesin!
- Satır sonlarında boş karakter **white-space** bırakmayın!
- Her `rb` dosyası ya da Ruby kodu içeren dosya boş bir satırla bitsin.
- Operatörler, virgöl, iki nokta, noktalı virgöl, `{` ve `}` lerin etrafında mutlaka boşluk `space` karakteri olsun!

## Yanlış

```
a=1
a,b=1,3
1>2?true:false;puts "Merhaba"
[1,2,3].each{|n| puts n}
```

## Doğru

```
a = 1
a ,b = 1, 3
1 > 2 ? true : false; puts "Merhaba"
[1, 2, 3].each{ |n| puts n }
```

- Parantez ve Köşeli parantez kullanırken, ne öncesine ne de sonrasına boş karakter `space` koyma!

## Doğru

```
my_method(arg1, arg2).other
[1, 2, 3].length
```

- Ünlenden sonra boş karakter `space` kullanma `!array.include?(element)`
- `when` ve `case` kullanırken girinti durumunu aşağıdaki gibi yap:



```

case
when song.name == "Misty"
  puts "Not again!"
when song.duration > 120
  puts "Too long!"
when Time.now.hour > 21
  puts "It's too late"
else
  song.play
end

kind = case year
  when 1850..1889 then "Blues"
  when 1890..1909 then "Ragtime"
  when 1910..1929 then "New Orleans Jazz"
  when 1930..1939 then "Swing"
  when 1940..1950 then "Bebop"
  else "Jazz"
end

```

- Method'lar arasında **1 satır boşluk** ver, gerekiyorsa mantıklı bir şekilde içeride ayırım yap!

```

def some_method
  data = initialize(options)

  data.manipulate!

  data.result
end

def some_method
  result
end

```

## Syntax

- Eğer method'a parametre geçiyorsan parantez yaz!

```

def some_method
  # argümansız
end

def some_method_with_arguments(arg1, arg2)
  # argümanlı
end

```

- `for` kullanımına dikkat edin, kafanıza göre her yerde kullanmayın:

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

# kötü örnek
for i in arr do
  puts i
end

# iyi örnek
arr.each { |i| puts i }
```

- İç içe **ternary** kullanmaktan kaçının! Okunabilirliği azaltıyor!

```
# kötü
some_condition ? (nested_condition ? nested_something : nested_something_else) : something_else

# iyi
if some_condition
  nested_condition ? nested_something : nested_something_else
else
  something_else
end
```

- `and` ve `or` yerine `&&` ve `||` kullanın
- Mümkün oldukça `if` leri tek satır şeklinde kullanın:

```
# kötü
if some_condition
  do_something
end

# iyi
do_something if some_condition
```

- `unless` içinde `else` kullanmaktan kaçının:

```
# kötü
unless success?
  puts "hata"
else
  puts "ok"
end

# good
if success?
  puts "ok"
else
  puts "hata"
end
```

- Tek satırlık blok işlerinde `{ }` , çok satırlık blok işlerinde `do` `end` kullanın.
- `return` kelimesini gerekmedikçe kullanmayın.
- Method'larda parametre olarak `default` değer atarken `=` etrafında **space** kullanın.

```
# kötü
def some_method(arg1=:default, arg2=nil, arg3=[])
  # kod...
end

# iyi
def some_method(arg1 = :default, arg2 = nil, arg3 = [])
  # kod...
end
```

- Varlık operatörü kullanmaktan çekinmeyin!

```
# eğer isim nil ya da false ise isim değişkenine "vigo" ata
isim ||= "vigo"
```

- **Boolean** değerler için `||=` kullanmayın!

```
# kötü
enabled ||= true

# iyi
enabled = true if enabled.nil?
```

- Parantezli kullanımda method'dan sonra **space** kullanmayın:

```
# kötü
f (3 + 2) + 1

# iyi
f(3 + 2) + 1
```

- **Block** içinde kullanmayacağınız değişken için değer ataması yapmayın:

```
# kötü, k boşa gitti
result = hash.map { |k, v| v + 1 }

# iyi
result = hash.map { |_, v| v + 1 }
```

- Veri tipi kontrolü için `===` kullanmayın! `is_a` ya da `kind_of?` kullanın.

## Naming (İsimlendirmeler)

- Method ve değişken isimleri için `snake_case` kullanın.
- Class ve Modül için `CamelCase` kullanın.
- Constant için `SCREAMING_SNAKE_CASE` kullanın.
- Boolean sonuç dönen method'lar `?` ile bitmeli: `User.is_valid?`
- Tehlike, nesneyi modifiye eden / değiştiren method'lar `!` ile bitmeli! `User.delete!`

## Class

- Singleton tanımlarken `self` kullanın:

```
class TestClass
  # kötü
  def TestClass.some_method
    # kod
  end

  # iyi
  def self.some_other_method
    # kod
  end
end
```

- `private`, `public`, `protected` olan method'larda girintileme method adıyla aynı hizada olsun ve ilgili method'un bir üst satırı boş kalsın:

```
class SomeClass
  def public_method
    # ...
  end

  private
  def private_method
    # ...
  end
end
```

## Exceptions

- Akış kontrolü için kullanmayın!

```
# kötü
begin
  n / d
rescue ZeroDivisionError
  puts "0'a bölünme hatası!"
end

# iyi
if d.zero?
  puts "0'a bölünme hatası!"
else
  n / d
end
```

## Diğer

- String'leri concat ederken interpolasyon kullanın:

```
# kötü
email_with_name = user.name + " <" + user.email + ">"

# iyi
email_with_name = "#{user.name} <#{user.email}>"
```

- Gerekmedikçe değer atamalarında tek tırnak `'` kullanmayın, çift tırnağı tercih edin `"`
- String concat işlerinde **Array**'e ekleme tekniğini kullanabilirsiniz, hızlı da olur:

```
html = ""
html << "<h1>Page title</h1>"

paragraphs.each do |paragraph|
  html << "<p>#{paragraph}</p>"
end
```

# Gerçek Hayat Ruby Örnekleri

Bu bölümde, gerçek dünyadaki Ruby konularına değineceğim. Bunların başında da **Neden Ruby?** sorusuna cevap vermeye çalışacağım. Bence Ruby dünyasını öne çıkartan en büyük farklılık **Test Driven Development** metodolojisinin çok gelişmiş olması.

Onlarca test kütüphanesi, test suite ve benzeri şeyler bence diğer dillerde bu kadar ileri seviyede değil. Yaptığınız uygulamayı **END TO END** yani A'dan Z'ye test etmek, tek tek tüm senaryoları çıkartmak ve neredeyse sıfır hata ile iş yapmak mümkün.

**Behaviour Driven Development (BDD)**, **Continues Integration (CI)**, test ve build otomasyonu da çok sık kullanacağımız şeylerden biri!

Genel anlamda konu başlıklarımız;

- [Neden Ruby?](#)
- [Ruby ve TDD/BDD/CI](#)
- [Kendi Rubygem'imizi yapalım!](#)
- [Sinatra ve Web](#)

# Neden Ruby?

Yazılım hayatıma [Commodore 16](#) BASIC dili ile başladım. Daha sonra [Commodore 64](#)'e geçtim, Assembly (Makine Dili), daha sonra da [Amiga](#)'da da Assembly'ye devam ettim. Bir gün geldi ve makine dili çöp oldu :) Teknoloji değişti :)

90'lı yıllarda [Windows](#) ve [ASP](#) diliyle tanıştım. ASP'nin ilk günleri, [VBScript](#) :) Uzunca bir süre ASP ile yola devam ettikten sonra [PHP](#) ile tanıştım ve "oooh be DÜNYA varmış" dedim.

PHP ile hatırı sayılır pek çok proje yaptım. Daha sonra, sevgili kardeşim [Firat Can Başarır](#)'ın yönlendirmesiyle [Python](#)) ve [Django](#)) ile tanıştım. Özellikle web development işleriyle uğraşan biri, PHP'den sonra Python ve Django'yu görünce hakikatten akli duruyor!

Özellikle Django ile birlikte gelen **Admin Panel** olayı insanın tabiri caizse dibini düşürüyor! Daha sonra anlaşıyor ki, bu panel, sadece **developer** için. Yani sadece geliştirme amaçlı. İlk anda "ooof, bununla hemen hızlıca işleri çıkartırım" diyorsunuz haklı olarak. Teoride mümkün de.

Ancak büyük bir işe kalkıştığınızda bu Admin Panel kabusunuz oluyor ve işi gücü bırakıp sadece bu paneli **customize** (yani *özelleştirme*) etmekle uğraşıyorsunuz.

Keza Python 2 mi? 3 mü? gibi durumlar da söz konusu. Python 3 neredeyse tamamen farklı. An itibariyle (30 Kasım 2014) halen [Django](#) ve [Python3](#) desteği resmi olarak gelmiş değil.

Bunun dışında Python topluluğu çok ağır hareket ediyor. Yeni çıkan bir servisin modül olarak hazırlanması ya da en son 3 sene önce güncellenmiş, yüzlerce pull-request'in beklediği GitHub projeleri mi istersiniz?

Bu konular beni geliştirici olarak zorladı. Amacım hızla işimi düzgün bir şekilde yapıp yoluma devam etmekten başka bir şey değil.

## Ruby is Fun!

Yani **Ruby eğlencelidir!** ne demek bu? İlk bakışta insan düşünüyor, programlama dilinin neresi eğlenceli olabilir ki? Neticede bilimsel bir işlem diye düşünüyor insan. Taa ki Ruby ile uğraşmaya başlayana kadar...

İlk dikkat çeken şey, Ruby dilinin insan diline çok benzemesi. Yani İngilizce bilen biri için okunması çok kolay. Hiç Ruby bilmeyen biri bile rahatlıkla ne tür bir işlem olduğunu anlayabilir.

Daha önce hiçbir programlama dilinde rastlamadığım bir `if` kullanımı:

```
puts "vigo" if a > 2
```

## Neden Ruby?

Bu şu demek, eğer `a` 'nın değeri `2` 'den büyükse ekrana `vigo` yaz. Asp, Python, Php, Perl, C gibi dillerde genelde `if` bloğu içinde ya da **oneline** yani tek satırda ifade şeklinde olurken ilk kez Ruby'de `if` koşulunun bu şekilde kullanıldığını gördüm.

`unless` kelime anlamı olarak **if not** anlamındadır. Yani eğer `x` 'in değeri `false` ise şunu yap derken:

```
puts "vigo" unless x
```

gibi bir kullanım söz konusu. yani aşağıdaki gibi de kullanılabilir ama mantıksal olarak tercih edilmez:

```
puts "vigo" if !x
```

Keza method isimleri, standart kütüphane ile gelen özellikler gayet akılda kalıcı ve mantıklı. Ruby sizin adınıza pek çok şeyi düşünüp hazır kullanıma sunuyor.

Acaba bugün günlerden cuma mı?

Hemen bakalım:

```
Time.now.friday? # => true # evet
```

Aslında çok basit ama çok işe yarayan bir özellik. Bu tarz konulara **Syntactic Sugar** deniliyor.

## Çok Güçlü ve Çalışkan Ruby Topluluğu

Ne yazık ki ülkemizde çok da bilinen ya da tercih edilen bir dil olmamasına rağmen, dünyada durum çok farklı. Pek çok tanınmış proje Ruby, Ruby on Rails, Sinatra gibi Ruby dünyasının araçlarını kullanmakta.

Python'dan gelen bir geliştirici olarak, yaşadığım en büyük sıkıntılardan biri de yavaş ilerleme durumuydu. Python'u ben Alman Mühendisliğine benzetiyorum. Her şey inanılmaz kurallı, süper sistemli olmak zorunda. Tamam bu çok güzel bir yaklaşım, kabul ediyorum ama bazen ağır kalıyor.

Bazen öyle bir Python modülüne ihtiyacınız oluyor ve bir bakıyorsunuz son 3 yıldır güncelleme yapılmamış, GitHub'da bekleyen 50 tane Pull Request, kodu kimin maintain ettiği belli değil, uzay boşluğunda kendi kendine giden bir durumda kaderini bekliyor.

Ruby topluluğu inanılmaz derecede üretken. Yeni bir API'mı çıktı? Hemen gem'ini bulmanız mümkün! Ruby mi öğrenmek istiyorsunuz? Tonlarca ücretsiz/ücretli online videolar, eğitimler var! Kitap, kod, konferans aklınıza ne gelirse...



Ne yazık ki bu denli aktif bir dünyayı ben diğer uğraştığım dillerde göremedim.

## Bir işi yapmanın pek çok farklı yolu olabilir!

Son olarak, Ruby'deki en çok hoşuma giden mentaliteden bahsetmek istiyorum. Bir işi doğru yapmanın birden fazla yolu olabilir.

Örneğin Python'da sadece "tek bir yol" varken Ruby'de farklı farklı yöntemlerle, komutlarla aynı işi değişik şekillerde yapabilirsiniz ve hepsi de doğrudur!

Neredeyse hiçbir programlama dili **DSL** (yani Domain Specific Language) yapmak için bu denli müsait değil. Yani Ruby kullanarak kafanıza göre Ruby üzerinde çalışan başka bir dünya yapabilirsiniz!

Neticede hayatta herşey tercihler ve zevklerle ilgilidir. Ben kendi nedenlerimi belirtiyorum, bunlar size uymayabilir, hatta nefret bile edebilirsiniz. Eğer böyle bir durum varsa, sizden ricam sakın bir şekilde Ruby dünyasını incelemeniz.

An itibariyle ne ile uğraşıyorsanız aynı şeyi Ruby ile yapmaya çalışmanız. Mutlaka deneyin. Denedikten, tadına baktıktan sonra karar verin.

Sevgili annem küçükken hep derdi "önce bi tadına bak ondan sonra istemiyorsan yemeği yeme" diye :)

# Ruby ve TDD/BDD/CI

wip...

# Kendi Rubygem'imizi yapalım!

wip...

# Sinatra ve Web

Belkide en popüler Ruby kütüphanesi micro-webframework [Sinatra](#) ile hızlıca mini web uygulamaları yapmak, basit API servisleri hazırlamak ve sunucuya deploy etmek konularını işleyeceğiz!

wip...

## Yazar Hakkında

1972'de İstanbul'da doğdum. Rahmetli anne-annem'in satın aldığı [Commodore 16](#) ile bilgisayar hayatımı değiştirdi. Uzun yıllar [Commodore 64](#) ve [Amiga](#) kullandım. İnternetin, Google'ın ve Stackoverflow'un olmadığı bir dünyada [Assembly](#) dili ile **code** yazdım.

1990'ların başında, televizyonlarda fırtına gibi esen [Dinozorus](#), Küp Küp, Sokak Dövüşçüsü gibi oyunları Amiga platformunda kodlamıştım. 1997'nin ortalarına kadar Amiga ile devam ettim hayatıma. 1998-2007 arasında Windows/PC kullanmak zorunda kaldım. [Asp 3.0](#) ile başlayan web programlama maceram, sırasıyla **JavaScript**, **Php**, **Perl**, **Python** ve son olarak **Ruby** ile devam etmekte.

2003 yılında evlendim, 2011'de dünyanın en güzel hediyesini verdi eşim. Kızım **Ezel** dünyaya geldi! Bazen beraber Amiga'da oyun oynuyoruz :)

Uzun yıllar [İstanbul Bilgi Üniversitesi](#)'nde çalıştıktan sonra 2013'te [webBox.io](#) şirketini kurdum. [Kod.io 2013](#), [Kod.io 2014](#), [Codefront.io](#), [Jsist 2014](#) gibi konferanslar düzenledik.

[Failconf](#), [Hack-ing](#) gibi organizasyonlar ve çeşitli meetup'lar yaptık. Ocak 2015'de webBox.io'nun faaliyetlerini bitirdik. Mayıs 2015'ten beri tekrar eski görevime, İstanbul Bilgi Üniversitesi'ne geri döndüm.

Halen, büyük bir heyecanla, **Amiga** ve **Commodore 64** makine dili programlama ile uğraşıyorum. Sizlere de tavsiye ederim :)