3

Ruby Sınıfları

Bir programlama dilini öğrenirken izlenebilecek en iyi yol yoktur. Konuları sistematik olarak öğrenmeye kalkarsanız, öğrenme olgusunun pedagojik kısmı yok olur. En öne alınması gereken basit giriş çıkış bilgileri bile, o dilin sonraki kavramlarıyla ilgilidir. O nedenle, kitapta ya da derste henüz açıklanmadan kullanılan kavram ve deyimleri olduğu gibi belleğimize alıp kullanmaya başlamaktan başka yol yoktur. Bu iş, bir çocuğun ana dilini öğrenmesi gibidir; taklit et ve belleğe yerleştir.

Bu bölümde, Ruby dilini öğrenirken karşılaşacağımız temel kavramlar, ayrıntıya inmeden, listelenecektir.

3.1 Ruby Yorumlayıcı Bir Dildir

Ruby yorumlayıcı bir dildir. Yazacağımız bir satırlık deyimleri, her işletim sisteminde var olan etkileşimli komut penceresinden (interactive console) yazacak ve Ruby'nin ona verdiği yanıtı gene o pencereden göreceğiz. Etkileşimli pencerelere kabuk (shell) de denilir. İşletim sisteminin komutlarının doğrudan yazıldığı penceredir.

Ruby 1.9 sürümü, öncekilere göre önemli yenilikler getirmiştir. Onlardan birisi `unicode` kullanmasıdır. Bir başkası da, üstsınıftaki bir metodun altsınıfa daha kolay çağrılmasıdır.

3.2 Ruby Kabuğu

Ruby kabuğu irb adını alır. Her işletim sisteminin kabuğu içinden açılabilir. Bunun için, işletim isteminizin kabuğunu açınız ve oraya irb yazıp entere basınız. Eğer, ruby'nin yüklendiği yeri sisteminiz biliyorsa, Ruby kabuğu açılacak ve ekranda

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

ye benzer bir görüntü gelecektir. Artık Ruby etkileşimli penceresi (Ruby kabuğu) açılmıştır. İstediğiniz Ruby komutunu (>) simgesinin karşısına yazabilirsiniz.

Bunun dışında, sisteminizde kısa yollar da yaratabilirsiniz. O zaman Ruby'yi gösteren ikona tıklamanız yetecektir. Ruby'nin yüklendiğini ve yolların (path) ayarlandığını varsayıyoruz. Bunu henüz yapmadıysanız hemen yapmalısınız.

Windows'ta Ruby Kabuğu

Etkileşimli pencereyi açınız. Windows 8 de Windows tuşuna basınız Masa üstüne gelen uygulamalar arasında Siyah zeminli *Komut İstemi* ya da Mavi zeminli *Windows Power Shell* kabuklarından birisini seçebilirsiniz. Yazacağımız satırlara yer açmak için önce `cd \` komutu ile kök dizinine gidelim. Sonra `irb` yazarak Ruby'nin etkileşimli penceresini açalım. Bütün bu basit işlerden sonra, Ruby kabuğu açılacaktır. Kabuğun görüntüsü sistemden sisteme değişik olabilir. Bizim için önemli olan $irb(main) : 001 : 0 >$ satırıdır. O satır, Ruby'nin sistemimizde yüklü olduğunu ve etkileşimli pencerenin *Ruby* komutu almaya hazır olduğunu gösterir. Aslında işletim sistemimizin kabuk penceresi Ruby'nin kabuk penceresine dönüşmüştür. Başka bir deyişle yorumlayıcı Ruby'nin etkileşimli ortamına girmiş durumdayız. Bu ortamdan çıkmadan, kabuk penceresine Ruby komutu dışında bir şey yazamayız. Ruby kabuğundan işletim sisteminin kabuğuna dönmek için

`>exit` ya da `Ctrl + D`

yazınız.

3.3 Gömülü Metotlar

Ruby'nin hemen her nesneye uygulanan gömülü komutları (metot) vardır. Bunlara `çekirdek` (kernel) metotları denir. Bunların hepsini birden ezberlemeye kalkışmayınız. Kullandıkça önemlileri zaten aklınızda kalacaktır.

Başlarda yazacağımız komutların Ruby'nin çekirdeğinde olduğunu, onları başka yerlerden çağırmaya gerek kalmadan doğrudan kullanabileceğimizi bilelim. Tabii, ileride kendimiz metot yazacak ya da çekirdekte olmayan metotları da kullanacağız. O zaman gerekenleri söyleyeceğiz.

3.4 Kaynak Program

Ruby kabuğu yazdığımız tek satırlık komutlara hızlı ve doğru yanıtlar veriyor. Yazdığımız satırlar bellekte tutuyor ve yukarı/aşağı ok tuşlarına basıldığında istediğimiz satırı tekrar görüntüye getiriyor. Bellekte tutulan satır sayısı sistemden sisteme değişebilir. Ancak, bilgisayar kapanınca bunların hepsi silinir. Onları yeniden yazmaktan başka bir yol yoktur. Çok kullanılan komutlar için bu durum komutu tekrar tekrar yazmayı gerektirir. Hem tekrarlardan sakınmak hem birden çok komut satırı yazabilmek için, kodlarımızı içeren dosyalar hazırlayıp kaydederiz. Bu tür dosyalara *kaynak program (source)* denilir.

Kaynak programı yazıp kaydetmek çok kolaydır. Bunun için basit bir editör (kelime işlemci) gerekir. Basit editör deyince, karakterleri yazan ve yazılan metni kaydedebilen, kaydedilen metinleri yeniden çağırabilen bir editörü anlayacağız. Bilgisayar programı yazmak için *MSWord* gibi gelişkin editörler kullanılmaz. Çünkü onlar derleyicimizin anlamayacağı biçemleme kodları koyabilirler. Windows'taki *Notepad* kaynak program yazmak için uygundur. Her işletim isteminde böyle basit editörler vardır. Örneğin Linux'ta *Gedit*, Mac OS X'de *TextWrangler* gibi editör kullanılabilir. Kullanılan Programlama dilinin anahtar sözcüklerini farklı reklerle gösteren ve kod tamamlama niteliklerine sahip editörler de vardır. O tür editörler program yazmayı kolaylaştırır. Ama şimdilik bize Notepad yetecektir. Anahtar sözcüklerin renkli görünmesini istiyorsanız, Windows'ta Notepad yerine *Notepad++* kullanabilirsiniz. Ücretsiz olarak internetten indirilebilir. Öteki işletim sistemleri için de benzer editörler vardır.

Önce kaynak programlarımızı içine koyacağımız bir dizin yaratalım. Windows işletim sisteminde

Liste 3.1.

```
md c:\RubyKaynak

cd c:\RubyKaynak
```

deyimlerinin ilki *c* : sürücüsünde kök dizinine bağlı *RubyKaynak* adlı dizin yaratır. İkinci deyim, o dizini etkin duruma getirir.

Şimdi *Notepad* 'i açalım ve şunları yazıp *RubyKaynak* dizini içine *ilkprg.rb* adıyla kaydedelim.

Liste 3.2.

```
#!ruby2.0.0
# encoding: UTF-8

>puts "Ruby öğreniyorum."
```

Birinci satır kullanılan Ruby sürüm numarasını gösteriyor. Yazılması gerekli değildir. İkinci satır, karakter kodlama sisteminin `UTF-8` olduğunu Ruby'ye bildiriyor. Programımızda *ascii* dışında karakterler varsa, bu satırı yazmalıyız. Üçüncü satır, kaynak programımızın yapmasını istediğimiz şeydir. *"Ruby öğreniyorum"* stringini konsola yazacaktır. Kaynak programa istediğiniz adı verebilirsiniz. Ama program adının sonuna *.rb* uzantısını eklemelisiniz. Bu uzantı, dosyanın bir Ruby kaynak programı olduğunu belirtir. Dosyayı derleyebilmesi için, Ruby'nin bu uzantıyı görmesi gerekir.

Şimdi bu programı işletim sisteminin etkileşimli kabuğunda derleyip koşturmak için

Liste 3.3.

```
ruby ilkprogram.rb
```

komutunu yazalım. Konsolda

```
Ruby öğreniyorum.
```

yazısını göreceğiz. Ruby'de kaynak program yazıp koşturmak bu denli kolaydır. Programın kısa ya da uzun olması önemli değildir. Çok iş için çok komut, az iş için az komut yazıyoruz. Uzun ya da kısa kaynak programlarını derleyip koşturmak için hep aynı işleri yapacağız.

3.5 Ruby Kaynak Programlarını Koşturma

Şimdi açık penceremizde ilk komutumuzu yazabiliriz. Her dile selam ile başlamak gelenektir. Biz de bu geleneğe uyalım:

Liste 3.4.

```
irb(main):001:0>puts 'Merhaba Ruby!'
```

`puts` Ruby'nin bir çekirdek (gömülü) fonksiyonudur. Bazı dillerdeki *println* yerine geçer. Parametresini string olarak yazar ve yeni satıra geçer. Parametre olarak her nesneyi alabilir; çok hünerlidir. Kabuk penceresinde

Liste 3.5.

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

satırlarını görürüz. Bunlardan ilki puts metoduna yazdıracağımız ve tek tırnak içine aldığımız metindir. O metni çift tırnak içine de alabilirdik. İkinci satır Ruby'nin verdiği yanıttır. İstenen metni, tırnak kullanmadan yazmıştır. Bu Ruby programının çıktısıdır. Üçüncü satırdaki $=>nil$ deyimi, çıktının nil olduğunu belirtiyor. Son satır ise, Ruby'nin etkileşimli kabuğunun tekrar komut almaya hazır olduğunu belirtiyor. `NilClass` sınıfı daha sonra ele alınacaktır.

Ruby'de metinleri sınırlamak için tek tırnak ve çift tırnak yanında başka simgeler de kullanırız. Çift tırnak (" ") tek tırnağa (' ') göre daha işlevseldir. Ayrıntıları ileride ele alacağız. Şimdilik metinleri (string) tek ya da çift tırnak içinde yazacağız.

Ruby'de metinler `String` sınıfından türetilir. String sınıfının, metinlerle ilgili işlemleri yapan metotları vardır. Örneğin, yukarıdaki metni büyük harfe dönüştürmek için `upcase` metodunu kullanırız.

Liste 3.6.

```
irb(main):001:0> puts( "Merhaba Ruby".upcase)
MERHABA RUBY
=> nil
irb(main):002:0>
```

Görüldüğü gibi, Ruby, metni ekrana büyük harflerle yazmıştır. Yazdırılacak metni, `puts` metodunun parametresi sayıp, yukarıdaki gibi ( ) parantezi içine alabiliriz. Ama bu Ruby için gerekli değildir. Komutu okuyanın kolay algılaması içindir. Üstelik, usta Ruby programcıları metot parametrelerini parantez içinde yazmazlar. Biz gene de algılanırlığı kolaylaştırmak için yeri geldikçe blokları parantezlerle sınırlayacağız.

Yukarıdaki ilk satırda `"Merhaba Ruby".upcase` yazdık. Burada çift tırnak içindeki *"Merhaba Ruby"*, String sınıfından üretilen bir nesnedir. Ruby'deki adı object'tir. `upcase` ise o nesneye uygulanan bir metottur. Nesne ile metot adı arasına nokta (.) bağlacı konulduğuna dikkat ediniz.

Bir sınıf bir şablon gibidir. Örneğin, *otomobil* dediğimizde, bir eşya sınıfını kasdederiz; belirli bir otomobili değil. Ama Erol'un otomobili dediğimizde, somut bir varlığı kastediyor oluruz. Otomobil bir sınıftır, ama

Erol'un otomobili, otomobil sınıfından türetilmiş somut bir nesnedir (instance). String (metin) dediğimizde de belirli bir stringi kastetmiyoruz. Ama *"Merhaba Ruby"* dediğimizde somut bir string kastediliyor (bkz ([7]).

Sınıflar, nesneleri üreten şablonlardır. Bir sınıftan üretilen nesne, ana bellekte *heap* denilen kısımda yer alır. Ona çekirdek metotlarını ve varsa sınıfın metotlarını uygulayabiliriz. Heap'te yaratılan nesne, işi bitene kadar orada kalır. İşi bitince ana bellekten silinir. Bu özelik, büyük programlarda ana belleğin dolup programın kitlenmemesi için iyi bir yöntemdir.

3.6 Metotlar

Bilgisayarlar verileri işler. Metotlar, her programlama dilinde verileri işleme araçlarıdır. Onlar, istediğimiz verileri içeren değişkenleri işleyen kodlar ile istediğimiz başka işleri yapan kodları kendi içinde barındıran program nesneleridir. Bir metot çağrılınca, onun içindeki kodlar çalışır. Böylece onun içindeki komutları tek tek yazmaktan kurtulmuş oluruz. Bir metodu programda istediğimiz yerlerde tekrar tekrar çağırabiliriz. Üstelik, gerekli ise başka programlara da taşırız. Bu saydıklarımız, metotların niteliklerinin az bir kısmıdır. İleride metotların başka hünerlerini örnekler üzerinde göreceğiz.

Ruby'de metot `def` anahtar sözcüğünü ile ile başlar `end` anahtar sözcüğü ile biter. `def` sözcüğünü *metot adı* izler.

Örnekler:

Liste 3.7.

```
1 def selam
    puts "Merhaba dostum!"
  end
```

Bu selam veren kısa bir metottur. İlk satır ile son satır arasında kalan bölgeye *metodun gövdesi* denilir. Bu metodun gövdesi bir tek deyim içeriyor. Ama uzun metotlar da böyle yazılır. Aralarındaki fark, gövdeye daha çok deyim yazılmasıdır.

Liste 3.8.

```
  def selam
2   puts "Merhaba dostum!".downcase
  end
```

Metot adından sonra varsa metodun parametreleri yazılır. Parametreleri tırnak içine alma zorunluğu yoktur ; aynı satıra yazılırlar, aralarına (,) konulur.

3.6.1 Metot Parametreleri ve Yerel Değişkenler

Bir metot tanımında iki türlü değişken kullanılabilir: *parametreler* ve *yerel değişkenler*. Parametrelere argüman (argument) da denilir. Bu kavramları gene bir örnek üzerinde açıklayacağız.

Liste 3.9.

```
def gelir_vergisi_hesapla (gelir)
  gelir_vergisi_orani = 35
  brut_gelir = gelir
  p gelir_vergisi = brut_gelir * gelir_vergisi_orani / 100.0
  p net_kazanc = brut_gelir - gelir_vergisi
end

gelir_vergisi_hesapla(50_000)
```

Metot adından sonra () içinde yazılan *gelir* adlı değişkene bilgisayar jargonunda parametre ya da argument denilir. Birden çok parametre varsa, onlar arasına virgül konulur. Fonksiyonun gövdesinde yer alan *gelir_vergisi_oranı, brüt_gelir, gelir_vergisi* ve *net_kazanç* değişkenlerine yerel değişkenler denilir.

Parametreleri parantez içine almak gerekmez; ama metot adının bulunduğu satıra yazılmalıdır.

Parametreli metot çağrılırken, parametrelere gerçek değerler konulur. Örneğin, yukarıdaki metodu çağırırken *gelir_vergisi_hesapla (50000)* yazılırsa, 50000 liralık brüt kazanç için, gövdede istenen gelir_vergisi ve net_kazanç hesaplanır. Parametre değer(ler)i, gerekiyorsa gövde içindeki yerel değişken(ler)e de atanabilir ya da gövde içindeki ifadelrde kullanılabilir. Bu örnekte, `gelir` parametresinin değeri *brüt_gelir* adlı yerel değişkene aktarılmaktadır. Ruby metotlarında `return` anahtar sözcüğünü kullanma zorunluğu yoktur. Son deyimdeki değer, metodun `return` değeridir. Ama öteki dillerden alışkanlık kazananlar `return` kullanabilirler.

Basamak sayısı çok olan sayıların kolay algılanırlığını sağlamak için, sayının binlik blokları arasına alt çizgi (_) konulabilir. Örneğin, yukarıdaki metodu çağırırken *gelir_vergisi_hesapla (50000)* yerine *gelir_vergisi_hesapla (50_000)* yazılmıştır.

3.6.2 Metot Çağırma

Anlık (instance) metotlar, ait oldukları nesne içinden çağrılabilir. Module metotları module adını izleyen nokta bağlacı ve onu izleyen metot adı ile çağrılır. Varsa metodun parametreleri yerine gerçek literal değerler yazılır:

```
module_adı.metot_adı

module_adı.metot_adı("Ankara")
```

Bazen kısaltmalar yapılabilir. Örneğin,

```
foo(*[1,2,3])
```

deyimi

```
foo(1,2,3)
```

yerine geçer.

3.7 Sınıf Bildirimi

Sınıf bildirimi için söz dizimi basittir. *class* anahtar sözcüğünü izleyen *sınıf_adı* yazılır. Sınıf gövdesi oluşturulduktan sonra end satırı yazılarak sınıf bildirimi sonlandırılır. Örneğin

Liste 3.10.

```
class Tamtakır
end
```

deyimleri boş bir sınıf yaratır. Boş sınıf, doğal olarak, bize yararlı işler yapmaz. Ama, her sınıf Ruby'de *Object* sınıfının altsınıfıdır. Dolayısıyla, yukarıdaki Tamtakır sınıfı, Object sınıfının özniteliklerine ve metotlarına kalıtsal olarak sahiptir. Tamtakır sınıfından bir nesne yaratmak için *new* operatörünü kullanırız.

Liste 3.11.

```
boş = Tamtakır.new
```

Bu deyim, *Tamtakır* sınıfına ait bir nesne (instance) yaratmıştır. Nesne yaratmak demek, teknik olarak, ana bellekte ona yetecek kadar bir yerin tahsis edilmesi demektir. Bir değişkene ana bellekte yer ayrılmasına benzer.

Sınıflardan üretilen bütün nesneler ana bellekte *heap* denilen bölgeye konulurlar. Ruby'nin heap düzenlemesi bu kitabın konusu değildir. Programcı sınıftan nesne yaratınca, Ruby onu ana bellekte uygun yere konuşlandırır.

Her nesnenin bir adı vardır. O ad, aslında, nesnenin ana bellekteki adresini gösteren bir işaretçidir *(reference, pointer)*. Bir sınıftan istenildiği kadar nesne türetilebilir. Onlar heap'te farklı adreslere konulur. Her nesnenin kendi pointeri vardır.

Yukarıdaki *Tamtakır* sınıfından türetilen nesnenin ana bellekteki adresini gösteren pointerin adına *boş* dedik. Pointere istediğimiz adı verebiliriz. Pointer aynı zamanda nesnenin adı rolünü de üstlenir. Programcı, nesnenin adresini bilmek zorunda değildir. Nesnenin pointeri, daima işaret ettiği nesnenin adresini bilir.

Bir sınıftan yaratılan bütün nesneler heap'te konuşlanır. Büyük programlarda nesne gerektiğinde yaratılır ve işi bitince silinir. Böylece heap'te gereksiz yer işgali olmaz.

Ruby'de sınıf adı bir `sabit`'tir. Sabitler büyük harfle başladığı için, sınıf adları daima büyük harfle başlar.

Yukarıda yarattığımız *boş* nesnesi, her nesne gibi, Object sınıfının her şeyine kalıtsal olarak sahiptir. *class* metodu Object sınıfının bir metodudur. Nesnenin hangi sınıftan üretildiğini bildirir. Örneğin,

Liste 3.12.

```
puts (boş.class)  #=> Tamtakır
```

deyiminin çıktısı *boş* pointerinin işaret ettiği nesnenin *Tamtakır* sınıfından üretildiğini söylüyor.

3.8 Ruby'de Sınıflar ve Nesneler

Ruby nesne tabanlı bir dildir. *"Ruby'de her şey nesnedir"* sözünü sık sık duyarız. Burada *"Peki ama sınıflar da nesne midir?"* sorusu akla gelmelidir. Evet, Ruby'de sınıflar da birer nesnedirler. Çünkü sınıflar da sıradüzenin (hierarchy) en tepesinde yer alan `BasicObject` nesnesinden türetilir. `BasicObject`'in altında `Object` nesnesi yer alır. Örneğin, *"abc"* stringi de *String* sınıfı da birer nesnedir. Aralarındaki fark *String*'in ayrıca sınıf olarak davranıyor olmasıdır. Başka bir deyişle,

Ruby'de her sınıf bir nesnedir, ama her nesne bir sınıf değildir.

`Sınıf` nesne tabanlı programlamanın başlıca aracıdır. Her sınıf bir veri tipidir. Her veri tipi bir sınıftır. Ruby'de her şey bir nesne olduğuna göre, her nesneyi türeten bir sınıf vardır.

Genel olarak, nesne tabanlı dillerde *sınıf, altsınıf* ve *üstsınıf* kavramlarını Bölüm 2 'te açıklamıştık. Bu bölümde sınıf, altsınıf ve üstsınıf kavramları ile kalıtım özeliklerini Ruby sözdizimini kullanarak basitçe açıklayacağız.

Örnekler

Ruby'de *sınıf* bildiriminin `class` anahtar sözcüğü ile başlayıp `end` anahtar sözcüğü ile bittiğini söyledik. `class` anahtar sözcüğünü sınıf adı izler. Sınıf adı büyük harfle başlar. Büyük harfle başlayan adlar Ruby'de sabit'tir. O nedenle, sınıf adı Ruby'de sabittir. Örneğin,

Örnek 3.1.

```
class Bitki
  def ad_ver (isim)
      @bitki_adı = isim
  end
end
```

Liste 3.1, `Bitki` adlı bir sınıf bildirimidir. Bildirimin ilk satırı ile son satırı arasındakilere sınıfın gövdesi denilir. Bu örnekte sınıf gövdesinde yalnızca *ad_ver* adlı bir metot bildirimi vardır. *ad_ver* metodunun tek parametresi vardır: *isim*. Metodun görevi, bu parametrenin değerini *@bitki_adı* adlı değişkene aktarmaktır.

Anlık Değişken Önünde @ simgesi olan değişkenlere anlık (üretik, üretik instance) değişkenler denilir. Anlık değişken, nesne yaratılırken nesne içinde yaratılır, nesne yok olurken yok olur. *Anlık* denmesinin nedeni budur. Öte yandan, sınıftan üretilen bir nesne içindedir. O nedenle *üretik* değişken diyoruz.

Anlık değişkenlere, ait oldukları nesne dışından erişilemez. Her anlık değişken işlevini ait olduğu kendi nesnesi içinde yapar.

3.9 Sınıftan Nesne Yaratma

Bir sınıftan nesne yaratmak için `new` operatörü kullanılır. Alışılmış olduğu için *operatör* diyoruz. Ama her operatör bir metottur ve bir Ruby nesnesidir. `new` metodu, bütün sınıflara uygulanabilir; yani her sınıftan nesne üretebilir.

Örnek 3.2.

```
b1 = Bitki.new
```

Liste 3.2, `Bitki` sınıfından `b1` adlı bir nesne üretiyor. `b1` üretilen nesnenin adıdır. Daha doğrusu, nesnenin ana bellekteki adresini işaret eden *pointer*'dir.

Bir sınıftan birden çok nesne üretilebilir:

Örnek 3.3.

```
b2 = Bitki.new
```

Liste 3.3, `Bitki` sınıfından `b2` adlı başka bir nesne üretiyor. Bunların farklı olduğunu görmek için

```
b1 == b2 # => false
```

deyiminin *false* değer verişinden anlarız. Gerçekten onlar ana bellekte farklı adreslere konuşlanmışlardır.

`b1` ve `b2` nesneleri birbirlerinden farklı olduğuna göre, her birisinin kendi gövdelerindeki anlık değişkenlere farklı değerler atanabilir. Örneğin,

Örnek 3.4.

```
b1.ad_ver(elma)
b2.ad_ver(kiraz)
```

deyimleri `b1` ve `b2` nesneleri içinde bitkilere, sırasıyla, *elma* ve *kiraz* adlarını verirler.

Şimdi bu deyimleri *bitkiler.rb* adlı bir dosya haline getirip, programlarımızı içeren *RubyKaynak* dizinine kaydedelim.

Program 3.1.

```
class Bitki
  def ad_ver (isim)
    @bitki_adı = isim
  end
end

b1 = Bitki.new
b2 = Bitki.new

  b1.ad_ver('elma')
  b2.ad_ver('kiraz')
```

Program 3.1 ile verilen kaynak programı derleyip koşturmak için, daha önce yaptığımız gibi,

```
ruby bitkiler.rb
```

komutunu yazalım. Konsolda metotlarımızın çalıştığını gösteren bir işaret görmeyiz. Ama, her şeyi doğru yaptıysak, Ruby bize bir hata uyarısı da göndermez. Böyle oluşu, programımızın çalıştığını gösterir. Gerçekte, programımız istenilen işleri yapmıştır. Konsola bir şey yazılmasını istemediğine göre, olan biten her şeyi olağan saymalıyız.

Şimdi sınıfımıza yeni metotlar ekleyeceğiz.

Bir sınıfın içinde istediğimiz kadar değişken ve metot bildirimi yapabiliriz. Örneğin, Liste 3.1 ile bildirilen `Bitki` sınıfına *ad_yaz* adlı yeni bir metot ekleyelim:

Örnek 3.5.

```
class Bitki
  def ad_ver (isim)
      @bitki_adı = isim
  end

  def ad_yaz
      return @bitki_adı
  end
end
```

Burada, `Bitki` sınıfına *ad_yaz* adlı yeni bir metot eklenmiştir. Bu metot *ad_ver* metodunun *@bitki_adı* adlı anlık değişkene atadığı *isim* değerini yazacaktır. Bu metodun parametresi yoktur; çünkü işlevine bakarsak, metot çağrılırken bir parametreye gerek kalmıyor. Başka metotlar için parametre gerekebilir.

Ruby'de `return` değerini yazma zorunluğu olmadığını söylemiştik. Bir metodun gövdesinde bulunan son değer o metodun return değeridir. Ama algılanırlığı kolaylaştırmak için, burada metodun bize vereceği değeri `return` anahtar sözcüğü ile belirliyoruz. İyice ustalaşana kadar böyle yapmakta sakınca yoktur.

Şimdi yaptığımız eklerle programı yeniden yazıp *bitkiler2.rb* adıyla kaydedelim. Verdiğimiz bitki adlarını yazdırabilmek umuduyla, programın sonuna `puts` metodunu elma ve kiraz için ayrı ayrı ekleyelim.

Program 3.2.

```
class Bitki
  def ad_ver (isim)
      @bitki_adı = isim
  end

  def ad_yaz
      return @bitki_adı
  end
```

```
end

b1 = Bitki.new
b2 = Bitki.new

  b1.ad_ver('elma')
  b2.ad_ver('kiraz')

puts ( b1.ad_yaz )
puts ( b2.ad_yaz )
```

Program 3.2'yi koşturunca Ruby konsola

Örnek 3.6.

```
  elma
  kiraz
```

stringlerini yazar. Son iki satırdaki `puts` metodunun `b1` nesnesindeki *ad_yaz* metodunu çağırdığını; son satırdaki `puts` metodunun da `b2` nesnesindeki *ad_yaz* metodunu çağırdığını görüyoruz.

initialize Metodu

Program 3.3 ile bildirimi yapılan Bitki sınıfına ait nesneler yaratılınca, anlık değişkenlere değer atamak için *ad_ver* metodunu kullandık. Nesne tabanlı programlamada anlık değişkenlere değer atayan metoda `verici` (setter), değer okuyan metoda da `alıcı` (getter) denilir. Anlık değişkenlerin her birisi için bu işin tek tek yapılması mümkün ama zaman alıcıdır. Bunun yerine, Ruby `initilize` metodunu kullanarak anlık değişkenlere nesne yaratılırken değerleri atayabilir. Anlık değişkenlere değer atanmadan kullanılmalarını da önlediği için `initilize` metodu tercih edilmelidir.

Program 3.3'de `Bitki` ve `Hayvan` adlı iki sınıf bildirimi var. İlk sınıftaki *anlık* değişkenlere *verici* yöntemiyle değer atanırken, ikinci sınıfın nesnelerine *initialize* yöntemi ile değer atanmaktadır. İkinci yöntemin daha kolay olduğu görülüyor.

Program 3.3.

```
# Ruby örnek programları
# iki sınıf bildirimi ve onlardan nesne üretme

class Bitki
  def ad_ver (isim)
      @bitki_adı = isim
  end
```

```
  def ad_yaz
      return @bitki_adı
  end
end

class Hayvan
      def initialize( hAd, hNitelik )
        @hayvan_adı       = hAd
        @nitelik    = hNitelik
      end

      def to_s # default to_s metodunu baskıla (override)
      "Hayvanın adı: #{@ad} , özelliği: #{@nitelik}\n"
      end
end


bitki1 = Bitki.new
bitki1.ad_ver( "Gül" )
puts bitki1.ad_yaz

h1 = Hayvan.new("At", "Çağlar boyunca insana dost olmuştur.")
h2 = Hayvan.new("Aslan", "Vahşi bir hayvandır.")
puts h1.to_s
puts h2.to_s

# inspect metodu nesnenin içini gösterir
puts "Birinci hayvanın özeliği: #{h1.inspect}"
puts "İkinci hayvanın özeliği : #{h2.inspect}"
```

Bu programın çıktısı konsolda şöyle görünür:

Liste 3.13.

```
>ruby bitkiVehayvan.rb
Gül
Hayvanın adı: At , özelliği: Çağlar boyunca insana dost olmuştur.
Hayvanın adı: Aslan , özelliği: Vahşi bir hayvandır.
Birinci hayvanın özeliği: #<Hayvan:0x007fb5aa2195f0 @ad="At",
 @nitelik="Çağlar boyunca insana dost olmuştur.">
İkinci hayvanın özeliği : #<Hayvan:0x007fb5aa219578 @ad="Aslan",
 @nitelik="Vahşi bir hayvandır.">
```

inspect Metodu

`inspect` metodu bir nesne içine bakmaya yarayan bir metottur. Program 3.3'nin son iki satırında metodun kullanılışını görüyoruz. Metodun verdiği sonuç Liste 3.13'nin 5-8.satırlarında görülüyor. Çıktı, sınıf adından sonra (:) yi izleyen bir sayı ve onun arkasından da nesne ile ilgili nitelemeyi verir. Sözkonusu sayısal değer sistemden sisteme değişir; sistemin sınıfa verdiği numaradır. Pointer rolünü oynar.

to_s Metodu

to_s metodu her nesneye uygulanır ve onu string olarak yazar. Örneğin,

Program 3.4.

```
1.to_s => "1"
[1,2,3].to_s => "[1,2,3]"
class Kimlik;end
Kimlik.new.to_s => "#<Kimlik:0x007fa8f4173ed8>"
```

3.10 Singleton

Bir sınıftan yalnızca bir tek nesne üretilmesini sağlar. Örneğin, `true, false` ve `nil` nesneleri böyledir.

Program 3.5.

```
require 'singleton'

class Deneme
  include Singleton
end
```

sınıfı tanımlandıktan sonra, nesne üretmek için izlediğimiz yolu izleyip,

```
Deneme.new
```

deyimi ile bir nesne üretmek istersek,

```
# => NoMethodError: private method `new' called for Deneme:Class
```

uyarısı alırız. İstediğimiz nesneyi yaratabilmek için, `instance()` metodunu kullanmalıyız.

Program 3.6.

```
require 'singleton'

class Deneme
  include Singleton

  def veri_ver(v)
     @veri = v
  end

  def veri_al ; @veri ; end

  def version
    '1.0.0'
```

```
14    end
  end

  obj1 = Deneme.instance
  obj2 = Deneme.instance
19 puts obj1 == obj2

  obj1.veri_ver(123)
  puts obj1.veri_al
  puts obj1.version
24
  obj2.veri_ver(456)
  puts obj2.veri_al
  puts obj2.version

29 puts obj1.veri_al
```

```
1 /**
  true
  123
  1.0.0
  456
6 1.0.0
  */
```

Açıklamalar: 19.satır, `Deneme` sınıfından `obj1` adlı bir nesne üretiyor. 20.satır, `Deneme` sınıfından `obj2` adlı bir nesne üretiyor. (Singleton sınıf olduğundan, ikinci bir nesne üretilememeli.) 21.satır. üretilen iki nesnenin aynı olup olmadığını denetliyor. Çıktı *true* oluyor. Bu demektir ki, *obj1* ile *obj2* aynı nesnelerdir. O halde, ikinci bir nesne üretilememiştir. İkinci nesne birinci nesne ile çakışmıştır. 23.ve 27. satırlar nesnede `veri` değişkenine farklı değerler atıyor. 24. ve 28. satırlar bu değerleri yazdıryor. 25. ve 29. satırlar nesnede version (sürüm) metodunu çalıştırıyorlar. Aynı sonucu alıyorlar. 28.satırda obj1'e atanan değer yazdırılıyor. Ama bu değerin 24.satırda yapılan değişikliği yansıttığını görüyoruz. Bu da singleton sınıftan iki nesne üretilemediğinin başka bir kanıtıdır.

3.11 Sınıfların Sıradüzeni

Program 3.3 'de `Bitki` ve `Hayvan` sınıflarının bildirimini yaptık. Her iki sınıfta anlık *bitki_adı* ve *hayvan_adı* değişkenlerini kullandık. Bu adlara değer atamak için *verici* (setter) metotlarını ve verilen adlara ulaşmak için *alıcı* (getter) metotlarını her sınıf için ayrı ayrı yazdık. Sonuçta yazdığımız programlar çalıştı. Bir sorun olmadı. Ama her iki sınıfta yapılan işler birbirinin aynıdır. Acaba bu işleri iki kez yapmaktansa, bir kez yapmanın bir yolu olabilir mi? sorusu aklımıza geliyor. Nesne tabanlı programlamada bu mümkündür; üstelik nesne tabanlı programlamada kalıtım (inheritance)

diye bilinen çok önemli bir niteliktir.

3.12 Altsınıf Bildirimi

Bir A sınıfının B adlı alt sınıfını tanımlamak için

class B < A end

sözdizimi kullanıldığını söylemiştik.

Program 3.7.

```
class A
end

class B < A
end

B.is_a? A
=> false

B.superclass == A
=> true

B < A # => true
A < A # => false

or use the <= operator

B <= A # => true
A <= A # => true

B.ancestors.include? A
```

Açıklamalar: 1 ve 2.satırlar boş A sınıfını tanımlıyor. 4. ve 5. satırlar B altsınıfını tanımlıyor. 7.satır B nin A olup olmadığını soruyor ve *false* yanıtını alıyor. 10.satır B nin üst sınıfının A olup olmadığını soruyor ve *true* yanıtını alıyor. 13. satır B nin A nın altsınıfı olup olmadığını soruyor ve *true* yanıtını alıyor. 14.satır A nın B sınıfının altsınıfı olup olmadığını soruyor ve *false* yanıtını alıyor. 18.satır satır B nin A nın altsınıfı ya da A ya eşit olup olmadığını soruyor ve *true* yanıtını alıyor. 19.satır satır A sınıfının, A nın altsınıfı ya da A ya eşit olup olmadığını soruyor ve *true* yanıtını alıyor. 21.satır B nin atasının A yı içerip içermediğini soruyor ve *true* yanıtını alıyor.

Program 3.8'de bir sınıfın kendisiyle ilişkisi açıklanıyor:

Program 3.8.

```
B.ancestors
#=> [B, A, Object, Kernel, BasicObject]

B < B
#=> false

B.ancestors.include? B
#=> true
```

Açıklamalar:

1.satır B nin atalarını listeliyor 4.satır b nin kendi kendisinin bir altsınıfı olmadığını söylüyor. 7.satır, B nin atalarının B yi içerdiğini söylüyor.

Kalıtımın Özelikleri

- Kalıtım, sınıflar sıradüzeninde (hierarchy), ataların bütün öğelerinin oğullara geçmesi olayıdır.
- Bir sınıfın öğeleri deyince, o sınıf içindeki değişkenleri ve metotları anlayacağız.
- Bir sınıf bir altsınıf doğurabilir. Doğurulan sınıfa `altsınıf`, doğuran sınıfa `üstsınıf` ya da `super` sınıf denilir.
- Altsınıf, üstsınıfın bütün öğelerine sahip olur. Buna `kalıtım` (inheritance) özeliği denilir.
- Altsınıf, üstsınıftan gelen öğelere ek olarak kendi değişkenlerini ve metotlarını tanımlayabilir. Tanımlanan yeni öğeler oğullarına aynen geçer, ama atalarına geçmez. Sosyal yaşamdaki *miras* gibidir.
- Bir sınıf birden çok altsınıf doğurabilir. Ama bir sınıfı doğuran üstsınıf bir tanedir.
- Altsınıf başka altsınıflar doğurabilir. Böylece sınıflar arasında yukarıdan aşağıya doğru giden bir sıradüzen (hierarchy) oluşur. Bu sıradüzende, bir sınıfın üstünde yer alanlara `ata` sınıflar (ancestor), altında yer alan sınıflara da `oğul` sınıflar (decendant) denilir.

Daha önce tanımladığımız, *Bitki* ve *Hayvan* sınıfları için `Varlık` adlı bir üstsınıf tanımlayalım:

Program 3.9.

```
class Varlık
  def ad_ver (isim)
      @ad = isim
  end

  def ad_yaz
      return @ad
  end
end
```

Bitki ve *Hayvan* sınıflarının öğelerini Varlık sınıfına taşıdığımıza göre, onları tekrarlamaya gerek yoktur. Ancak, altsınıflarda gerekiyorsa başka öğeler tanımlayabiliriz. Bir sınıftan başka bir sınıfı doğurtmak için küçük (<) karekteri kullanılır. Örneğin,

$$B < A$$

deyimi, A sınıfından B sınıfını doğurur. A üstsınıf, B altsınıf olur. Üstsınıfa *super* adı da verilir. Buna göre, şimdi Varlık sınıfının Bitki ve Hayvan sınıflarını doğurması için aşağıdaki kodları yazalım:

Liste 3.14.

```
class Bitki < Varlık
  def ad_ver(isim)
    super
  end

   def ad_yaz
       super
    end

  def meyve_verir_mi(b)
      @meyve = b
  end

  def cevap
    return @meyve
  end
end
```

Liste 3.14'de yapılanları açıklayalım. 2-4.satırlar `Varlık` üstsınıfında bildirimi yapılan *ad_ver* metodunun tanımının üstsınıftan (super) aynen alınacağını belirtiyor. 6-8.satırlar aynı eylemi *ad_yaz* metodu için yapmaktadır. 10-12.satırlar Bitki altsınıfının *meyve_verir_mi* adlı yeni bir metodunu tanımlanıyor. 14-17.satırlarda ise *cevap* adlı yeni metodu tanımlanıyor.

`Hayvan` altsınıfı için de benzer eylemleri yapalım.

Liste 3.15.

```
class Hayvan < Varlık
```

```
  def ad_ver(isim)
3   super
  end

   def ad_yaz
       super
8    end

  def eti_yenir_mi(e)
      @et = e
  end
13
  def yanıt
    return @et
  end
end
```

Şimdi bunları bir araya getirerek programımızı Program 3.10 gibi yazabiliriz.

Üstsınıftan Metot Çağırma

Program 3.10.

```
# encoding: UTF-8
# üst sınıftan metot çağırma (ruby 1.9 öncesi)
3
class Varlık
  def ad_ver (isim)
      @ad =  isim
  end
8
  def ad_yaz
      return @ad
  end
end
13
class Bitki < Varlık
  def ad_ver(isim)
    super
  end
18
   def ad_yaz
       super
     end

23 def meyve_verir_mi(b)
      @meyve = b
  end

  def cevap
28  return @meyve
  end
end
```

```
class Hayvan < Varlık
  def ad_ver(isim)
    super
  end

   def ad_yaz
       super
    end

  def eti_yenir_mi(e)
      @et = e
  end

  def yanıt
    return @et
  end
end

bitki  = Bitki.new
hayvan = Hayvan.new

bitki.ad_ver('elma')
bitki.meyve_verir_mi('evet')
print "Bitki adı : "
puts (bitki.ad_yaz)
print "Meyvesi yenir mi? "
puts (bitki.cevap)

hayvan.ad_ver('at')
hayvan.eti_yenir_mi('hayır')
print "Hayvan adı :"
puts (hayvan.ad_yaz)
print "Eti yenir mi? "
puts (hayvan.yanıt)
```

Programın çıktısı şöyledir:

```
Bitki adı : elma
Meyvesi yenir mi? evet
Hayvan adı :at
Eti yenir mi? hayır
```

Program 3.11.

```
#! encoding UTF-8
# üst sınıftan metot çağırma (ruby 1.9 sonrası)

class Varlık
  def ad_ver (isim)
      @ad =  isim
  end

  def ad_yaz
      return @ad
  end
end
```

```
class Bitki < Varlık
  @meyve
  def meyve_verir_mi (bool)
      @meyve = bool
    return @meyve
  end
end

class Hayvan < Varlık
  def eti_yenir_mi (mantıksal)
      @yenir = mantıksal
  end
end

b = Bitki.new
h = Hayvan.new

b.ad_ver('Elma Ağacı')
h.ad_ver('At')

print (" Adı nedir ? ")
puts (b.ad_yaz)
print "Meyve verir mi ? "
puts b.meyve_verir_mi('Evet')

print "Adı nedir ? "
puts h.ad_yaz
print "Eti yenir mi ? "
puts h.eti_yenir_mi('Hayır')
```

Bu programın çıktısı şöyledir:

```
>ruby varlik2.rb
 Adı nedir ? Elma Ağacı
Meyve verir mi ? Evet
Adı nedir ? At
Eti yenir mi ? Hayır
```

3.13 Baskılama (overriding)

Ata sınıftan gelen bir metodun oğul sınıfta değiştirilmesine *baskılama* (overriding) denilir. *to_s* metodu nesneyi string olarak yazan bir çekirdek metodudur. Her nesneye uygulanabilir. Atadan gelen her metot için olduğu gibi, *to_s* metodu da baskılanabilir. Program 3.3'nin 21.satırında bildirimi yapılan *to_s* metodu, atadan gelen aynı adlı metodu baskılıyor. Baskılanan metodun yerini, daima baskın metot alır. Ama ata sınıftakini kullanmak istersek, ona özel çağrı gerekir. Çağrı için, Ruby-1.9 ve sonraki sürümlerinde Program 3.10'de yapılanlara gerek yoktur. Üstsınıfın metotlarını altsınıfta çalıştırmak için Program 3.11'deki gibi yazmak yeterlidir. Bunun ne kadar kolaylık getirdiği, iki programın karşılaştırılmasıyla görülebilir.

Program 3.12.

```
# Baskılamaya örnek

class Hazine
      def initialize( ad, nitem )
        @isim        = ad
        @nitelik     = nitem
      end

      def to_s # öntanımlı to_s metodunu baskılar
           " #{@isim} hazinesi tarihte  #{@description}\n"
      end
end


x = "Selam olsun Bolu Bey'ine"
y = 123
z = Hazine.new( "Karun Hazinesi", "definecileri kendisine çekmiştir."
  )

p( x )
p( y )
p( z )
```

```
/*
"Selam olsun Bolu Bey'ine"
123
#<Hazine:0x007f84c38824e0 @isim="Karun Hazinesi", @nitelik="
 definecileri kendisine çekmiştir.">
*/
```

Buradaki `p` metodu `print` metodunun kısa adıdır.

3.14 Uygulamalar

3.14.1 initialize

`initialize` metodununu daha önce açıklamıştık. Tekrar edersek, anlık değişkenlere değer atamanın kolay bir yoludur. Atamaları nesne yaratılırken kendiliğinden yapar. Ayrıca `initialize` metodunun çağrılmasına gerek yoktur. Program 3.11' de kullanılan verici (setter) metodunun işlevini yapar. Üstelik, anlık değişkenlere nesne yaratılırken atama yapması, programda onlara değer atanmasının unutulması tehlikesini de ortadan kaldırır.

Program 3.13.

```
# mobilya.rb

class Mobilya
```

```
    def ad_ver( abc )
      @ad = abc
    end

    def ad_al
      return @ad
    end
end

class Koltuk
      def initialize( abc, belirle )
        @ad         = abc
        @niteleme   = belirle
      end

      def to_s # default to_s metodunu baskıla
            "Bu #{@ad} #{@niteleme}. \n"
      end
end

m = Mobilya.new
m.ad_ver( "İtalyan yapımı" )
puts m.ad_al

t1 = Koltuk.new("Koltuk", "18.Lui stilidir")
t2 = Koltuk.new("Kaplama", "Geyik derisidir")
puts t1.to_s
puts t2.to_s
#  inspect metodu nesnenin içine bakmamızı sağlar
puts "Birinci koltuğun özelikleri: #{t1.inspect}"
```

Programın çoktısı şöyledir:

```
İtalyan yapımı
Bu Koltuk 18.Lui stilidir.
Bu Kaplama Geyik derisidir.
Birinci koltuğun özelikleri: #<Koltuk:0x000000027f83f0 @ad="Koltuk",
 @niteleme="18.Lui stilidir"
```

Açıklamalar:

Mobilya üst sınıf, koltuk altsınıftır. Üst sınıfın ver (set) ve al (get) metotları var. 24-26.satırlar üstsınıfın metotlarını çalıştırıyor.

Alt sınıftaki initialize metodu üst sınıftaki set metodunun işlevini görüyor. 28. ve 29.satırlar alt sınıfın iki nesnesini yaratıyor. İnitialize metodu nesne yaratılırken kendiliğinden çalışır. Dolayısıyla anlık @ad değişkenine, sırasıyla, *Koltuk* ve *Kaplama* değerlerini atıyor. Bu işlem üstsınıftaki verici (ad_ver) metodunu kullanmaya gerek bırakmadığı gibi, nesne yaratılırken atama yapıldığı için kolaylık sağlıyor. Alt sınıfın ikinci metodu, her sınıfta öntanımlı var olan *to_s* metodunu baskılıyor. Aslında *to_s* metodu sınıfın adını yazması gerekirken, burada baskılanırken yapılan 20.satırı yazıyor. Son satırdaki *inspect* metodu sınıf ile ilgili bilgileri verir. Verdiği bilgiler çıktının son satırında görülüyor. Sınıf adını izleyen numara, sistemin sınıfı

tanıması için verdiği numaradır. Bir tür pointer işlevini görür. Son olarak, initialize metodunun tanımladığı @niteleme adlı anlık değişkene atanan değeri yazıyor.

3.14.2 Baskılama (overriding)

Oğul sınıf, atadan gelen bir metodu aynen kullanabileceği gibi, isterse değiştirerek kullanır. Atadan gelen bir metodun oğul içinde değiştirilmesine *metodun baskılanması* (method overriding) denilir. Günlük yaşamda mirasçının kendisinine miras kalan bir şeyi, örneğin bir evi, değiştirmesine benzer.

Program 3.14.

```
class Yazar
  def işi_nedir
    puts "Yazarlar yazı yazar"
  end
end

class Ozan < Yazar
  def işi_nedir
    puts "Ozanlar şiir yazar"
  end
end

class Romancı < Yazar
  def işi_nedir
    puts "Romancılar roman yazar"
  end
end

ozan   = Ozan.new
romancı = Romancı.new

puts ozan.işi_nedir
puts romancı.işi_nedir
```

Programın çıktısı şudur:

```
Ozanlar şiir yazar
Romancılar roman yazar
```

Açıklama: Program 3.14'de Yazar sınıfının *işi_ne* adlı bir metodu var. Bu metot iki alt sınıfında ayrı ayrı baskılandı (override). Çıktılardan, değişikliğin altsınıflara yansıdığı görülüyor.

Program 3.15.

```
3 class Kuş
    def ne_söyler
      puts "Her sabah gagamla tüylerimi temizlerim."
    end
    def ne_yapar
8     puts "Gökyüzünde özgürce uçarım!"
    end
  end

  class Kartal < Kuş
13   def ne_söyler
      puts "Yuvamı yükseklerde yaparım."
    end
  end

18 k = Kartal.new
  k.ne_söyler
  k.ne_yapar
```

Programın çoktısı şöyledir:

```
Yuvamı yükseklerde yaparım.
Gökyüzünde özgürce uçarım!
```

Açıklama: Kuş üstsınıfının iki metodu var. *ne_söyler* adlı metodu altsınıfta baskılandı. Ama *ne_yapar* metoduna dokunulmadı. Çıktıdan görüldüğü gibi, bsakılanan metot yeni işlevini yaparken, baskılanmayan metot üstsınıftaki işlevini yapıyor.

3.15 Üst Sınıftaki Metodu Çağırma

Altsınıfta baskılanan (override) metot yerine baskı geçer. Ama bazan baskı yerine üstsınıftaki asıl metodu kullanmak isteyebiliriz. O zaman super anahtar sözcüğünü kullanırız.

Examples:

Liste 3.16.

```
# encoding UTF-8
2
  class Hayvan
    def eylem
      "Ben yürürüm!"
    end
7 end

  class Kuş < Hayvan
    def eylem
      "Ben uçarım"
```

```
12    end
  end

  puts Hayvan.new.eylem
  puts Kuş.new.eylem
```

```
  /**
  Ben yürürüm!
  Ben uçarım!
4 */
```

İsterseniz 11.satır yerine

super + " ve ben uçarım"

koyabilirsiniz. O aman super metodu da çalışır. Deneyiniz.

3.16 Operatörler

Operatörler, matematikte alışkın olduğumuz işlemlere verilen addır. Doğal olarak, programlama dillerindeki veri tiplerine de uygulanıyor. Örneğin,

4+5*6/7

ifadesi toplama, çarpma ve bölme operatörleriyle yapılan bir işlemdir. Sonucu 8 dir.

Ruby operatörleini Bölüm 6'de daha ayrıntılı olarak inceleyeceğiz. Burada, yalnızca Ruby'nin alışkanlıklarımız dışında saylabilecek bazı operatörlerini listelemekle yetineceğiz:

Liste 3.17.

```
1         ::
            []
            **
            -(birli,unary)  +(birli,unary)  !  ~
            *, /, %
6           +, -
            <<, >>
            &
            |, ^
            >, >=, <, <=
11          <=>, ==, ===, !=, =~, !~
            &&
            ||
            .., ...
            =(+=, -=...)
16          not
          and, or
```

Operatörlerin çoğu metot'tur. Bazıları sözdizimlerine gömülürler ve onlar metot sayılmazlar:

Liste 3.18.

```
=, .., ..., !, not, &&, and, ||, or, !=, !~
```

+= gibi bazı operatörler dile gömülüdür; kullanıcı onları tanımlayamaz.

3.17 Yerleştirme

Değer Gömme, string interpolation

String İçdeğerlemesi (string interpolation) diye anılan bu yöntemi bir örnek üzerinde açıklayacağız.

Liste 3.19.

```
def ad_sor
  print( "Adınız nedir?")
  ad = gets()
  puts( "Merhaba #{ad}")
end
```

Liste 3.19 ile verilen *ad_sor* metodu 2.satırdaki *print* metodu ile kullanıcının adını soruyor. Burada `print` metodu yerine `puts` metodu da kullanılabilirdi. `puts` metodu stringi konsola yazdıktan sonra satırbaşı (new line) yapar. Oysa, `print` metodu metni yazdıktan sonra, yazdığı son karakterin sağında durur; satırbaşı yapmaz. 3.satır, `gets` metodu ile kullanıcının konsola gireceği adı alıyor ve onu ad değişkenine atıyor. Bu satır iki işlem yapıyor: Konsola girilen adı akuma ve onu değişkene aktarma.

4.satır önceki metotlardan bildiğimiz işi farklı biçimde yapıyor. `puts` metodunun parametresi "Merhaba #{ad}" stringi'dir. `puts` metodu parametresindeki stringi konsola yazmakla görevlidir. Ama string içinde ilk kez karşılaştığımız #{ad} simgeleri yer alıyor. #{...} simgesi ad değişkenine atanan değişken değerini içeren simgedir. Bu simge çift tırnak (" ") içinde verilen bir string içinde kullanılınca, onun tuttuğu değişken değeri, stringde yer aldığı konuma yazılıyor. Bu özelik Ruby'de string *yerleştirme (string interpolation)* diye anılır. Buna *değer gömme* de diyebiliriz. Bu yöntem yalnız string değerleri değil, satırbaşı, tab gibi yazılamaz karakterleri, sayıları, matematiksel ifadeleri ya da program parçalarını da gömebilir.