Programming in Scala

Ch2

`val numNames=Array("zero","one","two")`
aslında `Array.apply(...)`
Array mutable

List

List immutable
`val m=List[String]` olmaz, çünkü değişmez
`val m=List(1,2,3)`
`val m=1::2::Nil`
prepend, append'den daha hızlı. başa eklemek hiç maliyeti artmaz.
`ListBuffer, reverse, toList`
`:::` concatenate
`m(2)` aslında `m.apply(2)`
`drop, isEmpty, last, head, length, mkString, reverse`
`count, exists, filter, forall, foreach, map, remove, sort`: fonksiyon argümanlar alır

Tuples:

```
val pair=(99,"luft")
println(pair._2)
Tuple2[Int,String]
```
`pair(0)` çalışmaz, çünkü apply metodu hep aynı tipi döndürür

Set ve Map:

hem mutable hem immutable türler
`set+="ali"`
varsayılan immutable. `Set("ali", "mehmet")`
```
m=Map[Int,String]()
m+=(1->"ali")
Map(1->"ali",2->"mehmet")
```

functional style:

constants: val
daha okunaklı kod yazımını sağlar
yan etkisiz
Unit döndürmek-> yan etkili olmanın göstergesi
```
Source.fromFile(name).getLines.toList

var maxWidth=0
for(line<-lines)
      maxWidth.max(widthOfLength(line))
```
=
```
val widths=lines.map(widthOfLength)
```

```
val maxWidth=widths.reduceLeft((a,b)=>a.max(b))
```
reduceLeft: applies the passed function to the first two elements in widths, then applies it to the result of the first call and the next element in widths
```
val padding=" "*numSpaces
```

Ch3

method parameters are vals not vars
last value in the function body is returned
birden çok return ifadesi bulunmamalı. bu şekilde metotlar kısa olmak zorunda olur
tek satırlık metotlarda küme parantezlerine gerek yok:
```
def add(b:Byte):Unit=sum+=b
```
Unit tipinden sonuç döndüren metotlar, yan etkileri için icra edilir
varsayılan importlar: `java.lang`, `scala` ve singleton `Predef`
dosya ismiyle class ismi aynı olmak zorunda değil

Application trait
```
object MyApp extends Application {
      println("something")
}
```
Bu obje doğrudan çalıştırılabilir, main metodu Application tarafından sağlanır.

Ch5 Basic Types and Operations

`"""` ile ham metinler aynen girilir
ön boşlukları silmek için pipe `|` kullanılır
```
println("""|merhaba
       |ah canım""".stripMargin)
```
Symbol literal
dinamik dillerde, undeclared field identifier yerine geçer. örnek:
```
def updateRecordByName(r:Symbol,value:Any)...
updateRecordByName('favoriteAlbum,"ah my world")
val s='aSymbol
s.name

val sum=1+2
=
val sum=(1).+(2)
```
infix operator notation:
```
s indexof 'o'     // = s.indexOf('o')
s indexof ('o',5)       // = s.indexOf('o',5)
```
prefix operatorlar:
```
-7
=
(7).unary_-
```
prefix operatörler: `+,-,!,~`
postfix operatörler: parametre almayan metotlar
```
7 toLong
```
genel uzlaşı: eğer yan etki varsa parantez () kullan

Eşitlik:

```
null == List(1,2,3)
List(1,2,3)==List(1,2,3)
```

javadan fark: == operatörü referans tipleri için, referans eşitliğini test eder, scala ise değer eşitliğine bakar
operatör önceliği, operatörlerin harf sıralamasına bağlı:

Table 5.3 · Operator precedence

(all other special characters)
* / %
+ -
:
= !
< >
&
^
|
(all letters)
(all assignment operators)

birleşme özelliği: `:` ile biten operatörler, sağ işlenene (operand) bağlanır:

```
a:::b        // = b.:::(a)
```

gruplanma:

```
a:::b:::c          // = a:::(b:::c)
a*b*c // = (a*b)*c
```

basic type değişkenlere, rich wrapper ile yeni metotlar eklenir

```
Byte > scala.runtime.RichByte
```

Ch6 Functional Objects

class parameters:

```
class Rational(n:Int,d:Int)
```

primary constructor:

```
class Rational(n:Int,d:Int) {
      println("created " + n+ "/"+d)
}
```

toString:

```
class Rational(n:Int,d:Int){
      override def toString = n+"/"+d
}
```

precondition:

```
class Rational(n:Int,d:Int) {
      require(d!=0)
```

`require, println` gibi Predef içinde tanımlıdır

fields: nesnelerin iç veriye erişimi için gerekir

```
class Rational(n:Int,d:Int)
      val numer:Int=n
```

`r.numer`
auxiliary constructors
```
      def this(n:Int)=this(n,1)
```
`new Rational(5)`
üst constructorları yalnızca primary constructor çağırabilir

implicit conversion
`2*r` normalde yasak olurdu fakat
`implicit def intToRational(x:Int)=new Rational(x)`
bunlar kaynak kodunda açıkça bulunmaz, compiler tarafından yürütülür

Ch7 Built-in Control Structures

`if, while, for, try, match`, function callls
bütün kontrol yapıları bir değer döndürür. geçici değişkenleri kaldırır
```
val filename=
      if(!args.isEmpty)args(0)
      else "default.txt"
```
böylece val kullanabilmiş olduk.
val'ın faydası, değerin değişmeyeceğinden eminsin yanlışlıkla.
val'ın diğer faydası, equational reasoning. değerin kendisi = değeri hesaplayan ifade
`while` ve `do-while` sadece döngü, ifade değil. döndürdükleri `Unit` yani `()`
`greet()==()`
javada assignment, değer döndürür, scalada unit döndürür
while döngüleri, fonksiyonel paradigmaya çok uygun değil. bunun yerine ifadeler (for veya fonksiyon) kullanılmalı.

```
def gcd(x:Long,y:Long):Long=
      if(y==0)x else gcd(y,x%y)
```

For:

`for(file<-files)`
Range: `1 to 4` veya `1 until 4`

filtering:
```
for(file <- files if file.getName.endsWith(".scala"))
for(
      file<-files
      if file.isFile;
      if file.getName.endsWith(".scala")
}
```
nested iteration:
```
for(
      file<-files
      if file.getName.endsWith(".scala")
      line<-fileLines(file)
      if line.trim.matches(pattern);
}
```
midstream variable
```
for(
      trimmed=line.trim
```
burada tanımlanan değişken bir val değişkenidir

yeni küme oluşturmak:
```
def scalaFiles=
for{
      file<-files
      if ..
} yield file {...}
```
for clause içinde küme parantezi kullanınca ; kullanma gereği kalmaz
```
for clauses yield body
```

throwing exception

```
val half=
      if (n%2==0)
            n/2
      else
            throw new RuntimeException("n must be even")
```
half değişkeninin tipi, `n/2`'nin tipidir. throw ifadesinin tipi `Nothing`

catching:
```
try{ ...
} catch {
      case ex:FileException=> ....
```
scala'da checked exceptionların yakalanması zorunluluğu yok
try-catch cümleciği (clause) değer döndürür, finally döndürmez, eğer açıkça return kullanılmamışsa.

match ifadesi:

switch gibi. _ varsayılan
```
arg match {
      case "salt"=> println("salt")
      case _=>println("hul")
}
```
javadan fark: sadece enum ve int ile sınırlı değil. break belirtmeye gerek yok. değer döndürür
scala compiler özyinelemeli fonksiyonları aslında while döngülerine çevirir, çünkü kuyruk çağrısı optimizasyonu henüz yok

güzel kod parçaları

```
var forLineLengths=
for {
      file<-files
      if file.getName.endsWith(".scala")
      line<-getFileLines(file)
      trimmed=line.trim
      if trimmed.matches("for")
} yield trimmed.length
```

çarpım tablosu:
```
def makeTable()={
   val tableSeq=
```

```
      for(row <-1 to 10)
         yield makeRow(row)
   tableSeq.mkString("\n")
}
def makeRow(row:Int)=makeRowSeq(row).mkString
def makeRowSeq(row:Int)=
   for(col <-1 to 10) yield {
      val prod=(row*col).toString
      val padding=" "*(4 -prod.length)
      padding+prod
   }
println(makeTable())
```

yield Seq nesnesi döndürür. mkString çağrısı bu yüzden

Ch8 Functions and Closures

local function:
```
def m(){
      def g(){}
}
```

function literal
`(x:Int)=>x+1`
function value
`val increase=(x:Int)=>x+1`
function call:
`increase(10)`

tüm koleksiyonlarda `foreach` metodu bulunur. bu metot parametre olarak fonksiyon alır.
`Iterable trait`'inde `foreach` tanımlıdır. bu da tüm koleksiyonların üst sınıfıdır.

function literalleri kısaltmak:

target typing
`someNumbers.filter((x)=>x>0)`
someNumbers int tipinden olduğundan, x argümanı da int tipindendir çıkartır.
parantezleri kaldırmak
`someNumbers.filter(x=>x>0)`

placeholder
```
someNumbers.filter(_>0)        // = x=>x>0
val f=(_:Int)+(_:Int)
```
iki tane _ = iki parametre alır

partially applied functions

```
nums.foreach(println _)        // nums.foreach(x=>println(x))
```
@soru: neden `println(_)` değil de `println _`
```
def sum(a:Int,b:Int)=a+b
val a=sum _
```
a fonksiyonuna yapılan bir çağrı, sum fonksiyonuna yönlendirilir
```
val b=sum(1,_:Int,3)
```

```
nums.foreach(println)   // nums.foreach(println _)
```
burada println fonksiyonunun tüm parametrelerini boş bıraktığımızdan böyle yapabildik

```
drop(n:Int)...
println(drop(3))  // result
println(drop)     // hata verir
println(drop _)   // function
```
_ sembolünü eğer argüman olarak fonksiyon bekleniyorsa, ihmal edebilirsin.

closures:

```
(x:Int)=>x+more
```
yukarıda more değişkeni, function literal (simgedeğer) içinde tanımlanmıyor. dışarıdan geliyor.
more, free variable. x, bound variable.
serbest değişkeni olmayan, fonksiyon simgedeğerleri closed term (kapalı terim) diye adlandırılır
```
(x:Int)=>x+1
```
terim kaynak kodunun bir bitidir.
açık değişkenlerde değişiklik olursa, closure bunu görür. tersi de doğrudur.
```
nums=List(-2,3)
var sum=0
nums.foreach(sum+= _)
```
+1
birden fazla aynı değişkenden varsa: hangi scopeda closure oluşturulmuşsa ona bağlanır:
```
def makeIncreaser(more:Int)=(x:Int)=>x+more
val inc1=makeIncreaser(1)
inc1(10)
```
11
more değişkenini farklı scopelarda farklı değerlere bağlamak mümkün
@soru: sum+=_ ifadesi bir function value olarak nasıl yazılır? placeholder ne olacak?

repeated parameters
son değişken tekrarlanabilir
```
def echo(args:String*)=for(arg<-args)println(arg)
```
aslında `String* =Array[String]` fakat array tipinden bir argüman göndermek için:
```
echo(arr:_*)
```

tail recursion:
```
def approximate(guess:Double):Double=
      if(isGoodEnough(guess)) guess
      else approximate(improve(guess))
```
veya
```
def approximate(guess:Double):Double= {
      var guess=initialGuess
      while(!isGoodEnough(guess))
            guess=improve(guess)
      guess
}
```
bu iki algoritma hemen hemen aynı hızda çalışır. ilki fonksiyonel.
son eylem olarak kendi kendisini çağıran fonksiyonlar: tail recursive
kuyruk özyinelemesi yapılması için, son fonksiyon çağrısından sonra hiçbir şey yapılmamalı.

sorular

1. hangileri doğru:

```
filesEnding(".scala").foreach{x=>println(x.getName)}
filesEnding(".scala").foreach{println _.getName}
filesEnding(".scala").foreach{println(_.getName)}
```

2. hangileri doğru

```
filesMatching((String=>Boolean=_.endsWith(query)))
filesMatching(_.endsWith(query))
filesMatching((x:String)=>x.endsWith(query))
```

yanıt:

1 a

2 b,c, bunlar function literal

Ch9 Control Abstraction

reducing code duplication:

```
object FileMatcher{
   private val files=(new java.io.File(".")).listFiles
   private def filesMatching(matcher:String=>Boolean)=
      for(file<-files;if matcher(file.getName)) yield file
   def filesEnding(query:String)=
      filesMatching(_.endsWith(query))
   def filesContaining(query:String)=
      filesMatching(_.contains(query))
   def filesReqex(query:String)=
      filesMatching(_.matches(query))
   def main(args:Array[String]){
      filesEnding(".scala").foreach{x=>println(x.getName)}
   }
}
```

`_.endsWith(query)` closure

`_.endsWith(_)` closure değil. böyle de yapılabilir.

`filesMatching(matcher:String=>Boolean)` function value when matcher=f.literal

`matcher(file.getName)` function call

`filesMatching(_.matches(query))` function literal

simplifying client code

```
def containsNeg(nums:List[Int])=nums.exists(_<0)
nums.exists(_%2==1)
```

currying:

```
def curriedSum(x:Int)(y:Int)=x+y
curriedSum(1)(2)
=
def first(x:Int)=(y:Int)=>x+y
val second=first(1)
```

veya

```
val second=curriedSum(1)_
```

writing new control structures

```
def twice(op:Double=>Double,x:Double)=op(op(x))
```

```
twice(_+1,5)
7

object Writer{
   def withPrintWriter(file:File,op:PrintWriter=>Unit){
      val writer=new PrintWriter(file)
      try{
         op(writer)
      } finally {
         writer.close()
      }
   }
   def main(args:Array[String]){
      withPrintWriter(
         new File("date.txt"),
         _.println(new java.util.Date)
      )
   }
}
```

veya

```
      withPrintWriter(
         new File("date.txt"),
         x=>x.println(new java.util.Date)
      )
```

tek parametreli fonksiyonlarda () yerine {} kullanılabilir. amaç fonksiyon simgedeğişmezi olan argümanların görsel olarak ayrıştırılabilmesi

```
def withPrintWriter(file:File)(op:PrintWriter=>Unit)...
withPrintWriter(file){
      _.println(new ...)
}
```

by-name parameter

if, while gibi kontrol yapıları

`def myAssert(predicate: ()=> Boolean)=...`

`myAssert(()=>5>3)`

`myAssert(5>3)` şeklinde kullanmak için argümanın tipi `=>Boolean` olmalı

```
def myAssert(predicate:=>Boolean)=
      if(assertionsEnabled&&!predicate())
            throw new AssertionError
```

farklılık: `predicate:=>Boolean`, değil: `predicate:()=>Boolean`

hatalar

block must end in result expression, not in definition

C:\projects\cinar-agaci-01\ari-kovani-01\scala-01\test-02\chap02>scala -classpath . Writer.scala

(virtual file):1: error: block must end in result expression, not in definition

çözüm:

C:\projects\cinar-agaci-01\ari-kovani-01\scala-01\test-02\chap02>scala -classpath . Writer

Ch10 Composition and Inheritance

abstract classes
```
abstract class Element {
      def contents:Array[String]
}
```
contents soyut bir metot

parameterless methods:
```
def height:Int=contents.length
def height():Int
```
empty-paran methods
her iki durumda da height kullanılır. fakat eğer bir işlem yapılıyorsa, () kullanılması tavsiye ediliyor

extending:
```
class ArrayElement(conts:Array[String]) extends Element {
      def contents:Array[String]=conts
}
```
private olmayan tüm üyeler miras alınır
```
scala.AnyRef =java.lang.Object
```

overriding
field, parantezsiz metotları iptal edebilir
alan ve metot aynı isimlere sahip olamaz
bunun sebebi iki tane namespace bulunması:
- values (fields, methods, packages, singleton objects)
- types (classes, traits)

parametric fields
class parameter ve field aynı isme sahip olması
```
class ArrayElement(
      val contents:Array[String]
) extends Element
```

superclass constructor
```
class LineElement(s:String) extends ArrayElement(Array(s)) {
ArrayElement(Array(s))
```

override modifier
üst somut sınıfların üyelerini iptal etmede zorunlu -> yazım hatalarını önlemek için
soyut sınıfların üyeleri için seçmeli

final ve polymorphism java ile aynı

array concatenate: `this.contents++that.contents`

```
new ArrayElement(
      for(
            (line1,line2)<- this.contents zip that.contents
```

```
      ) yield line1 + line2
)
```

`Array(1,2,3) zip Array("a","b")`
sonuç:
`Array((1,"a"),(2,"b"))`
`((line1,line2)<-...)` çiftin (pair) iki öğesini isimlendirmeyi sağlayan bir pattern. yani line1=1, line2="a" bunu tek tek arrayin tüm öğeleri üzerinden dolaşır for

factory methods

```
object Element {
      def elem(..):Element = ...
}

import Element.elem
abstract class Element {...}
```

Bu durumda Element'in alt sınıfları private yapılabilir, çünkü elem metoduyla nesne ağacını oluşturmak mümkün. private yapmak için, bu classları, Element içine gömmek mümkün.

Ch11 Scala's Hierarchy

Any tüm sınıfların üst sınıfı. Nothing ve Null tüm sınıfların alt sınıfı.

Any:

```
==    final
!=    final
equals
hashCode
toString
```

kök sınıfı `AnyVal` ve `AnyRef`
AnyVal, tüm built-in value class'ların üst sınıfı: `Byte, Short, Double, Unit..`
`Unit` tek instance: `()`

implicit conversion

`42 max 43`
max metodu Int sınıfında tanımlı değil, scala.runtime.RichInt sınıfında tanımlı. gizli bir şekilde RichInt'e dönüşüm gerçekleştirilir, sonra sonuç yine Int olarak döndürülür
`max, min, until, to, abs`

`AnyRef` tüm referans sınıflarının üst sınıfı. `AnyRef, java.lang.Object` sınıfının diğer adı.

autoboxing

Int değerleri saydam bir şekilde Integer sınıfına gerektiğinde çevrilir. toString çağrılması gibi.
fakat javadan farklı olarak referans ve primitif değerler arasında bir fark programcıya yansımaz.
`new Integer(42)` her seferinde farklı bir obje üretir javada.

referans eşitliği için: `AnyRef.eq`. final. tersi: `ne`

bottom types

`scala.Null` tüm AnyRef sınıfların alt sınıfı.
`scala.Nothing` tüm sınıfların altı.

`Int x=null`: type mismatch
Nothing sınıfının değeri yok. kullanım yeri, tüm Exceptionların da alt sınıfı olmasından kaynaklanır:

Predef:
```
def error(message:String):Nothing=
      throw new RuntimeException(message)

def divide(x:Int,y:Int):Int=
      if(y!=0)x/y
      else error("can't divide")
```

Ch12 Traits

code reuse için
mixing into classes

a trait can be mixed in to a class using either extends or with

class'tan fark:
class parameter yok
super çağrılarının nereye gideceği runtimeda belli olur

sparse versus rich interfaces

sparse: az metot
az metot bulunması implemente ediciler için iyi, istemciler için kötü

ordered
Ordered.compare metonunu implemente etmek <, >, <=, >= operatörlerinden yararlanabilmeyi sağlar.

```
class Rational... extends Ordered[Rational]{
      def compare(that:Rational) =...
```

type parameter: `Ordered[Rational]`

stackable modifications:

Traits let you modify the methods of a class and they do so in a way that allows you to stack those modifications with each other.
```
abstract class IntQueue {
      def get():Int
      def put(x:Int)
}
class BasicIntQueue extends IntQueue{
      ...
      def put(x:Int){buf+=x}
```

```
}
trait Doubling extends IntQueue{
      abstract override def put(x:int){super.put(2*x)}
}
class MyQueue extends BasicIntQueue with Doubling
val queue = new MyQueue
queue.put(10)
queue.get()
res1:Int=20

trait Incrementing extends IntQueue{
      abstracet override def put(x:Int){super.put(x+1)}
}
val queue=(new BasicIntQueue with Incrementing with Doubling)
```

önce en sağdaki trait çalışır. bunun super çağrısı, bir soldaki trait'e devredilir.

trait class'a karıştırıldığında, buna mixin denir.

why not multiple inheritance?
eğer çoklu kalıtım olsaydı, hangi put metodu çalışırdı:

```
val queue=(new BasicIntQueue with Incrementing with Doubling)
```

ayrıca alt sınıfta hangi üst sınıfı çağırdığımızı belirtmeliydik:

```
def put(..) {
      Incrementing.super.put
      Doubling.super.put
}
```

Bu durumda, put iki defa çağrılmış olurdu.
neden böyle?
yanıt linearization. scala yeni nesne oluştururken, tüm miras alınan sınıf ve özellikleri (trait) doğrusal bir sıraya koyar. eğer sonuncusu dışında tümü super çağrısı yaparsa, sonuç stackable behavior olur.

Ch13 Packages and Imports

nested packages, c# namespace gibi

```
package a{
      class A
      package b{
            class B
      }
      package c{
            class C{
                  val x=new b.B
...
```

paketler içiçe koyulabildiğinden, a.b.B yerine b.B denilebiliyor

hidden package names:

```scala
// In file launch.scala
package launch {
  class Booster3
}
// In file bobsrockets.scala
package bobsrockets {
  package navigation {
    package launch {
      class Booster1
    }
    class MissionControl {
      val booster1 = new launch.Booster1
      val booster2 = new bobsrockets.launch.Booster2
      val booster3 = new _root_.launch.Booster3
    }
  }
  package launch {
    class Booster2
  }
}
```

Listing 13.5 · Accessing hidden package names.

import herhangi bir yerde yapılabilir
paketin kendisi de import edilir, içiçe paketler gibi kullanılabilir.

```scala
import java.util.regex
...
regex.Pattern...
```

selector clause:

```scala
import Fruits.{Apple, Orange}
```

veya

```scala
import Fruits.{Apple=>McIntosh,Orange}
```

isim değiştirme Apple "McIntosh" olarak isimlendirilir

```scala
import Fruits.{Apple=>McIntosh,_}
```

tüm obje üyelerini import eder, Apple'ın ismini değiştirir

```scala
import Fruits.{Apple=>_,_}
```

Apple dışındaki üyeler import edilir

access modifiers:

içiçelik kurallarına göre kapsam sınırlaması oluşturur

```
class Outer {
  class Inner {
    private def f() { println("f") }
    class InnerMost {
      f() // OK
    }
  }
  (new Inner).f() // error:  f is not accessible
}
```

Listing 13.9 · How private access differs in Scala and Java.

protected da aynı şeklde

scope of protection
`private[bob] class Navigator`
Navigator sınıfı, bob paketi içinden erişilebilir, diğer yerlerden erişilemez
`private[this] var speed=2000`
sadece nesneden erişilebilir

visibility and companion objects
class ve companion object birbiriyle erişim haklarını paylaşır

Ch14 Assertions and Unit Testing

assertions
`assert(this1.width==that1.width)`
veya ensuring. bu tam returnden sonra kullanılır
`ensuring(w < _.width)`
_ burada döndürülen nesne

ScalaTest
```
class ESuite extends Suite {
      def testMe(){
            assert ...
```
running: `(new ESuite).execute()`

FunSuite trait:
```
class ESuite extends FunSuite{
      test("elem should be ..."){
            assert(...)
```

failure report:

`assert(2===3)`
veya
```
expect(2) {
      ele.width
}
```

catching Exception:

```
intercept(classOf[IllegalArgumentException]){
      elem(..)
}
```

JUnit ve TestNG

geçiş için JUnit3Suite veya TestNGSuite trait

tests as specifications
BDD

```
class ElementSpec extends Spec {
      "description"--{
            "spec"-{
                  ...
            }
...
```

property based testing

ScalaCheck

```
test("description",(w:Int)=>
      w>0==>(elem(..).width==w)
)
```

==> implication operator. when the left hand expression is true, the right hand must hold true
ScalaCheck çok sayıda w değeri üretir ve test eder

Ch15 Cases Classes and Pattern matching

case classes:

fabrika metodu ekler
tüm parametreler val field

```
abstract class Expr
case class Var(name:String) extends Expr
case class Number(num:Double) extends Expr
case class UnOp(operator:String,arg:Expr) extends Expr
case class BinOp(operator:String,left:Expr,right:Expr) extends Expr

object MainExpr{
   def simplifyTop(expr:Expr):Expr = expr match {
      case UnOp("-",UnOp("-",e))=> e
      case BinOp("+",e,Number(0))=>e
      case BinOp("*",e,Number(1))=>e
      case _ => expr
   }
   def main(args:Array[String]){
      println(simplifyTop(UnOp("-",UnOp("-",Var("x")))))
   }
}
```

patterns:

constant: `"-", 2`
constructor: `UnOp("-",UnOp("-",e))`
wildcard: _

match bir ifade (expression), yani her zaman bir değer döndürür.
eğer hiç eşleşme olmazsa, `MatchError`

Kinds of patterns:

Var(x) patterni: x yerine herhangi bir değişken bağlar
Var(_) de kullanılabilir

variable patterns:

```
expr match {
      case 0=>"zero"
      case else => "noz zero"+else
}
```

`case Pi`: sabit değer kabul edilir
`case pi`: değişken
`case this.pi`: sabit olabilir
`case `pi``: `sabit
`case BinOp(Number(0))`: deep checks

sequence patterns:

```
expr match {
      case List(0,_,_)=>println("found")
      case _ =>
}
```

`case List(0,_*)`
`case (a,b,c)`: arbitrary 3-tuple

typed patterns:

```
case s:String
case m:Map[_,_]
```

uzun yol: `expr.isInstanceOf[String]`
casting: `expr.asInstanceOf[String]`

type erasure: `Map[Int,Int]` mapin tipine dair bilgi tutulmuyor caselerde
variable binding:

```
case UnOp("abs",e@UnOp("abs",_))=> e
```

e değişkeni bağlanır

pattern guard

```
case BinOp("+",x,y) if x == y => ...
case n: Int if 0 < n => ...
```

Pattern overlaps:
eğer bir case hiç erişilemezse, o zaman compile hatası verir

sealed classes:

```
sealed abstract class Expr
def describe(e:Expr):String=e match{
```

```
      case Number(_)=>"..."
}
```
sealed class cannot have any subclass except the ones in the same file. böylece eşleştirmenin kapsayıcı olması garanti altına alınabilir.
compiler warning: match is not exhaustive

option type:

```
def show(x:Option[String])=x match {
      case Some(s)=> s
      case None => "?"
}
val capitals= Map("France" -> "Paris")
show(capitals get "France")
France
show(capitals get "Turkey")
?
```
faydaları:
1. Option[String] tipindeki bir değişkenin her zaman String değil, bazen de null (None) olabileceğini istemci anlar
2. Scala bu tip değişkenlerin null kontrolünü şart tutar

patterns everywhere:

patterns in variable definitions:
multiple variable assignment to a tuple
```
val mytuple=(123,"abc")
val (number,string) = mytuple
```
veya patternlerle:
```
val exp  = new BinOp("*",Number(5), Number(1))
val BinOp(op,left,right) = exp
```

match expressions as partial functions:

aslında her bir match ifadesi bir fonksiyon, her bir case clause bir fonksiyon giriş yeri gibidir:
```
val withDefault: Option[Int] => Int = {
      case Some(x) => x
      case None => 0
}
withDefault(Some(10))
10
withDefault(None)
0
```

patterns in for expressions

```
for ((country, city) <- capitals)

val results=List(Some("apple"),None,Some("orange"))
for (Some(fruit) <- results) println(fruit)
apple
orange
```

Ch16 Working with Lists

list literals:
`List(List(1,0),List(0,2))`
`List()`
immutable: öğeler değiştirilemez
özyinelemeli yapı: linked list

list type:
homojen: tek bir tipten öğeler
`List[String], List[List[Int]], List[Nothing]`
covariant: if S subtype of T, then `List[S]` subtype of `List[T]`
`val xs: List[String] = List()`

constructing lists:
iki temel inşa bloğu: `Nil, :: (cons)`

sorting algorithm:
```
def isort(xs:List[Int]):List[Int] =
      if (xs.isEmpty) Nil
      else insert(xs.head, isort(xs.tail))
def insert(x:Int, xs:List[Int]) : List[Int] =
      if (xs.isEmpty || x < xs.head) x::xs
      else xs.head :: insert(x, xs.tail)
```

list patterns:
`val List(a,b,c) = fruit`
veya
`val a::b::rest=fruit`

sıralama algoritması pattern matching ile:
```
def isort(xs:List[Int]) : List[Int] = xs match {
      case List() => List()
      case x:xs1 => insert(x, isort(xs1))
}
def insert(x:Int, xs:List[Int]) : List[Int] = xs match {
      case List() => List(x)
      case y:ys =>      if (x<=y) x::xs
                  else y::insert(x, ys)
}
```

first-order methods on class list

first order: parametre olarak fonksiyon almayan metotlar

::: list birleştirme
dışarıdan birleştirme algoritması:
```
def append[T](xs:List[T],ys:List[T]): List[T] =
      xs match {
            case List() => ys
            case x::xs1=> x::append(xs1,ys)
      }
```

length pahalı bir işlem.
`list.init` ve `list.last` = `list.tail` ve `list.head` bunların tersi

bu işlemler (`init ve last`) de pahalı.
`reverse`
`drop, take ve splitAt`. bunlar tail ve init metotlarının genelleştirilmeleri
`take(n)` ilk n üye
`drop(n)` ilk n üye hariç
`splitAt(n)` iki liste: ilk n ve diğerleri
`apply`: lineer zaman alır
`indices`: mümkün olan indisleri döndürür
`zip`: iki listeyi alır, birbirlerinin üyelerinden oluşan çiftlerden liste üretir
`toString, mkString, addString`
`toArray, copyToArray, elements`

higher-order methods:

map:
```
val m = List(1,2,3)
m.map(_ + 1)
words.map(_.length)
words.map(_.toList.reverse.mkString)
words.map(_.toList)     // içiçe liste oluşturur
words.flatMap(_.toList) // düzleştirir
List.range(1,5) flatMap (
      i=> List.range(1,i) map( j=> (i,j))
)
```
veya
```
for (i <- List.range(1,5); j<-List.range(1,i)) yield (i,j)
m foreach (sum += _)
foreach prosedür alır (Unit döndüren fonksiyon)
```

filtering:

filter: operantlar: List ve predicate
```
m filter (_ % 2==0)
words filter ( _.length==3)
```
partition: iki liste döndürür, biri yükleme uyar, diğeri uymaz
```
m filter (_ % 2==0)
```
`find`: yükleme uyan ilk öğeyi döndürür
`takeWhile, dropWhile`: yükleme uyan öğeye varıncaya kadar olan öğeleri döndürür veya onların dışındakileri döndürür
`span: (takeWhile, dropWhile)`

`forall`: yüklemi tüm öğeler için doğrular
`exists` yüklemi en az bir öğe için doğrular

```
/:,:\
def sum(xs:List[Int]): Int = (0/:xs) (_ +  _)
sum(List(a,b,c)) = 0+a+b+c
```
\ en sondaki öğeden başlıyor, / en baştakinden

```
words sort (_.length > _.length)

List object
List(1,2,3)
```

```
List.range(1,9,2)
List.make(5,'a')
List.unzip(zippedPairs)
List.flatten(nestedLists)
List.concat(list1,list2)
List.map2(List(10,20),List(3,4,5)) (_ * _)
res= List(30,80)
List.forall2(List("abc","de"), List(3,2)) (_.length == _)
List.exists2(List("abc", "de"), List(3,2)) (_.length != _)
```

Ch17 Collections

genel bakış:
`Iterable < Seq, Set, Map`
`Seq < List, Array`

`Iterable Iterator` üretir: elements metodu ile
`elements` tek soyut metot
`flatMap, filter` vs. pek çok somut metot Iterable tarafından sağlanır

sequences:

`lists`: rasgele erişim yavaş
`arrays`: rasgele erişim hızlı
list buffers:
sona eklemeli listeleri adım adım oluşturmak için. değiştirilebilir
```
val buf=new ListBuffer[Int]
buf+=1
buf.toList
```
array buffers: uzunluğu değişebilir array
queues: fifo, hem değiştirilebilir hem değiştirilemez tipler
```
val q1=new Queue[Int] + 1
val q123=q1.enqueue(2,3)
val (elem, q23) = q123.dequeue
```
değiştirilebilir
```
val q=new Queue[String]
q+="a"
```
stacks:lifo, hem değişir, hem değişmez
```
val s=new Stack[Int]
s.push(1)
s.pusl(2)
s.top
s.pop
```
strings (via `RichString`)
`Seq[Char]` aslında
```
def hasUpperCase(s:String) = s.exists(_.isUpperCase)
```

sets and maps:

hem değişir, hem değişmez var
```
import scala.collection.mutable
```

```
val m1=mutable.Set(1,2)
val m2=Set(1,2)
+,-
nums ++ List(5,6)
```
veya --
```
nums ** Set(1,3)
```
intersection
```
size, contains(3), Set.empty[String], +=, -=, ++=, --=, clear
```

map:

```
nums+("vi"->6)
nums-"ii"
nums++List("iii"->3,"v"->5)
--, size, contains, keys, keySet, values, isEmpty, +=,-=,++=,--=
```
sorted sets and maps: `TreeSet, TreeMap`
senkronize:
```
new HashMap[String,String] with SynchronizedMap[String,String]
```
`override def default`: eğer key yoksa bunu döndürür

mutable:
```
var people=Set("ali")
people+="mehmet"
```
izin verir. people için yeni set oluşturur.
```
import scala.collection.mutable.Map
var capital = Map
toList, toArray
```
convert to mutable:
```
val mutaSet = mutable.Set.empty ++ immSet
```

tuples:
```
val (word,index)=tuple
val word,index=tuple // multiple definitions
```

Ch18 Stateful Objects

getter: `x`
setter: `x_`
setter ve getterları fielddan bağımsız olarak da tanımlayabilirsin

```
class Thermometer {
      var celsius: Float = _
```
_ varsayılan değer. Numerik değerler için 0. eğer bunu koymazsan, scala alanı soyut olarak algılar

Ch19 Type Parameterization

functional queues:
```
head, tail, append
```
fully persistent data structure, immutable
extended at the end

information hiding:

private constructors and factory methods:

```
class Queue[T] private (
      private val leading: List[T]
```

bu durumda dışarıdan nesne oluşturmak için, auxiliary constructor:

```
def this(elems: T*) = this(elems.toList,Nil)
```

veya factory method:

```
object Queue {
      def apply[T](xs:T*) = new Queue[T](xs.toList,Nil)
```

bu companion object aynı dosyada olmalı classla.

private classes:

alternatif olarak class private tanımlanır, trait ihraç edilir

```
trait Queue[T] {
...
object Queue {
      def apply...
      private class QueueImpl[T](...) extends Queue[T] {
```

variance annotations:

Queue type değil, trait. bu trait parameterized type'lar belirtmeye izin verir: `Queue[String]`
covariant (flexible): `Queue[S]` can be used in place of `Queue[T]` if S subtype of T
scala bunu varsayılan olarak desteklemez. tüm generic tipler, nonvariant (rigid)
covariant için:

```
trait Queue[+T] {...}
```

contravariant: T subtype of S => `Queue[S]` subtype of `Queue[T]`
trait `Queue[-T]`
`+, -`: variance annotations
değiştirilebilir verilerde covariant tehlikeli; örneğin:

```
class Cell[T](init: T) {
  private[this] var current = init
  def get = current
  def set(x: T) { current = x }
}
```

Listing 19.5 · A nonvariant (rigid)

Scala compiler. (It doesn't, and we'll explain why
construct the following problematic statement seq

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get
```

arrayler covariant değil:
```
val a: Array[Any] = Array("abc") // hata
```
javadaki array metotlarını kullanabilmek için:
```
val a2: Array[Object] = a1.asInstanceOf[Array[Object]]
```

değiştirilebilir alanlar, contravariant pozisyonlardır.
sadece mutable alanlar sorun değil:

```
class StrangeIntQueue extends Queue[Int] {
  override def append(x: Int) = {
    println(Math.sqrt(x))
    super.append(x)
  }
}
```

The append method in StrangeIntQueue prints out the s
(integer) argument before doing the append proper. Now,
counterexample in two lines:

```
val x: Queue[Any] = new StrangeIntQueue
x.append("abc")
```

lower bounds:
```
class Queue[+T] {
      def append[U >: T](x:U) = new Queue[U](...)
```
append için T alt sınırdır. U, T'nin üst sınıfı olacak bir parametre kullanılmalı. (contravariant)
types-driven design: types of an interface guide its detailed design and implementation

Liskov substitution principle: T substitutes U if T requires less and provide more

```
trait Function1[-S, +T] {
      def apply(x:S):T
}
S=>T
```
yere geçebilecek fonksiyonlar:
require less in arguments
provide more in results

```scala
class Publication(val title: String)
class Book(title: String) extends Publication(title)

object Library {
 val books: Set[Book] =
   Set(
     new Book("Programming in Scala"),
     new Book("Walden")
   )
 def printBookList(info: Book => AnyRef) {
   for (book <- books) println(info(book))
 }
}

object Customer extends Application {
 def getTitle(p: Publication): String = p.title
 Library.printBookList(getTitle)
}
```

Listing 19.9 · Demonstration of function type parameter variance.

upper bounds:

```scala
def sort(T<:Ordered[T])
```

Ch20 Abstract Members

```scala
trait Abstract {
      type T
      def transform(x:T):T
      val initial:T
      var current:T
```

type:

```scala
class Concrete extends Abstract {
      type T = String
      def transform(x:String)=x+x
```

type yerine somut sınıflar, bir örnek verir

initializing abstract vals

```scala
trait RationalTrait {
      val num: Int
```

```
      val den: Int
}
```
instance of an anonymous class
```
new RationalTrait {
      val num = 1
      val den = 2
}
```
`new Rational(expr1, expr2)` farkı:
somut örnekte, expr1 önce ilklendirilir
anonim örnekte, expr1 anonim sınıftan sonra örneklendirilir
iki çözüm imkanı:
1. pre-initialized fields
```
new {
      val num = 1*x
      val den = 2*x
} with RationalTrait
res: Object with RationalTrait = 1/2
```
superclass constructor call'dan önce parantez içindeki alan tanımı gerçekleşir
```
object twoThirds extends  e
      val num =2
      val den = 3
} with RationalTrait
```

2. lazy vals:

alan kullanıldığı ilk seferde ilklendirilir
```
object Demo {
      lazy val x =  { println("hello");"done"}
}
```
parametresiz def ile tanımlanmış metot gibi, ancak val sadece bir kez çağrılır
```
private lazy val g = {
      require(den != 0)
      gcd(num, den)
}
```
bu durumda alanları class tanımlamanın ardından ilklendirmek güvenli olur:
```
new LazyRationalTest {
      val num = 1
      val den = 2
}
```

abstract types:

```
class Food
abstract class Animal {
  def eat(food: Food)
}
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) {} // This won't compile,
}                                  // but if it did,...
class Fish extends Food
val bessy: Animal = new Cow
bessy eat (new Fish)     // ...you could feed fish to cows.
```

çözüm:

```
class Food
abstract class Animal{
      type SuitableFood <: Food
      def eat(food: SuitableFood)
}
```

Bir sığırın ne türde bir et yiyebileceği, Animal seviyesinde belirlenemez.

```
class Grass extends Food
class Cow extends Animal {
      type SuitableFood = Grass
      override def eat(food: Grass) {}
}
```

path-dependent types

`val bessy: Animal = new Cow`

`bessy.SuitableFood`, path-dependent type. yani bir objeye bağlı tip.
javadaki inner class type gibi. farkı, outer class yok, outer object var.
inner class types:

```
class Outer {
      class Inner
}
addressed: Outer#Inner
val o1 = new Outer
val o2 = new Outer
```

`o1.Inner ve o2.Inner` iki farklı path-dependent type

```
new o1.Inner
new Outer#Inner // yasak
```

enumerations:

```
object Color extends Enumeration {
      val Red, Green = Value
}
import Color._
```

Enumeration defines inner class of Value. same-named parameterles Value method returns a new

instance of that class. `Color.Red` is of type `Color.Value`. this is path-dependent type

```
val North=Value("North")
for (c<-Color)print(c + " ")
Color.Red.id
res: 0
Color(1)
res: Green
```

Ch21 Implicit Conversions and Parameters

3. parti kütüphanelerdeki classları genişletmek için

implicit conversions

```
implicit def stringWrapper(s: String)=
      new RandomAccessSeq[Char] {
            def length = s.length
            def apply(i: Int) = s.charAt(i)
      }

stringWrapper("abc123") exists (_.isDigit)
res: true
```

conversion yapmadan da kullanabilirsin:

```
"abc123" exists (_.isDigit)
res: true
```

scala compiler conversion yapar, sanki String, RandomAccessSeq sınıfının metotlarını almıştır

```
def printWithSpaces(seq: RandomAccessSeq[Char]) =
      seq mkString " "
```

String göndersen de hedef tipe çevrilir:

```
printWithSpaces("xyz")
res: x y z
```

rules for implicits:

eğer type hatası olursa, compiler implicit çevrimleri kullanmayı test eder

x+y hata olursa, convert(x)+y dener. convert müsait bir implicit conversion metodudur. eğer yeni nesne + metodunu desteklerse, mesele yok.

marking rule:
only definitions marked implicit are available
variable, function, object definition

```
implicit def intToString(x: Int) = x.toString
```

scope rule:
implicit conversion must be in scope as a single identifier
`x+y -> aVariable.convert(x)+y` dönüştürülmez
ancak import edilirse olur
`import Preamble._` yaygın implicit çevrimler için kullanılır
ancak kaynak veya hedef tipin companion objesinde tanımlıysa mesele yok

```
object Dollar {
      implicit def dollarToEuro
```

non-ambiguity rule:
çevirim ancak muğlaklık yoksa yapılır
bu kodu çok karmaşıklaştırırdı, eğer olsaydı. açıkça yazmak gerekir tercih edilen çevrimi.

only one implicit is tried:
`convert1(convert2(x))+y` olmaz
explicits-first rule:
type hatası yoksa, implicit çevrim denenmez
naming: eğer birden çok implicit ihtimali varsa, belirli bir tanesini ismiyle açıkça import edersin:
`import MyConversions.stringWrapper`

where tried:
implicit conversion to an expected type:
compiler X görüyor, fakat Y istiyor.
örnek:
```
val i: Int = 3.5
found: Double
required: Int
implicit def doubleToInt(x: Double) = x.toInt
```
`val i: Int = 3.5` düzgün çalışır
arkada şu çalışır:
`val i: Int = doubleToInt(3.5)`
normalde bu tip bir çevrim tavsiye edilmez çünkü hassasiyet kaybediliyor. tersi daha doğru: `Int to Double`.
`Predef`:
`implicit def int2double(x: Int): Double = x.toDouble`

converting the receiver:

iki fayda: 1. yeni bir sınıfın, mevcut bir sınıf ağacına uyumu. 2. DSL yazımı

`obj.doIt` yazıyorsun fakat doIt metodumuz yok. compiler "has a doIt" tipi arar.
interoperating with new types:
client programmers use existing types as if they are new types

```
class Rational {
      def + ...
val oneHalf = new Rational(1,2)
oneHalf + oneHalf
```
ya şu: `1+oneHalf`
receiver doğru + metoduna sahip değil
`implicit def intToRational(x: Int) = new Rational(x, 1)`
bundan sonra 1+oneHalf çalışır: `intToRational(1) + oneHalf`

simulating new syntax:

`Map(1->"one,2->"two")`
-> nasıl destekleniyor? sentaks değil.
```
object Predef {
      class ArrowAssoc[A](x:A) {
            def -> [B](y:B): Tuple2[A, B] = Tuple2(x,y)
      }
      implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
            new ArrowAssoc(x)
```

```
      ...
```
rich wrappers pattern: common in syntax like extensions

implicit parameters:

```
someCall(a) replaced with someCall(a)(b)
new Some(a) --> new Some(a)(b)

object Greeter {
      def greet(name: String)(implicit prompt: PreferredPrompt) { ...
object JoesPrefs {
      implicit val prompt = new PreferredPrompt("yes")

import JoesPrefs._
Greeter.greet("joe")
```
eksik olan parametreyi kapsamdaki implicit val değişkenler arasından bulur

implicit parameters are often used to provide imformation about a type mentioned explicitly in an earlier parameter list.

```
def maxListImpParm[T](elements: List[T])(implicit orderer: T => Ordered[T]): T
=
      ...
      val maxRest=maxListImpParm(rest)(orderer)

maxListImpParm(List(1,5,10,3))
```
int ordered değil, fakat `Int => Ordered[Int]` dönüşümü sayesinde bu fonksiyon çalışır

style rule:

custom named type: `PreferredPrompt String`'den daha özelleşmiş isim

örneğin:
```
def maxListPoorStyle[T](elements: List[T])
      (implicit orderer: (T, T)=> Boolean) : T
```
`(T,T)=>Boolean` türünde çok sayıda fonksiyon bulunur. bu tam olarak ne istendiğini göstermiyor.
`T=>Ordered[T]` ne istendiğini çok iyi tarif ediyor
role determining name

view bounds

```
def maxList[T](elements:List[T])
      (implicit orderer: T=> Ordered[T]) : T =
      ...
      val maxRest = maxList(rest)   // maxList(rest)(orderer) implicit
      if (x > maxRest) x            // orderer(x) > maxRest implicit
```

orderer parametresi açıkça hiç kullanılmamış. bu yüzden parametrenin ismi herhangi bir şey olabilirdi.
scala parametrenin ismini ihmal etmeye izin verir:

```
def maxList[T <% Ordered[T]](elements: List[T])       // view bound
```
`T <% Ordered[T]` = I can use any T, so long as T can be treated as an Ordered[T]
`T <: Ordered[T]` = T is an Ordered[T]

Int is not a subtype of `Ordered[Int]`, fakat Int to Ordered[Int] implicit dönüşümü mümkün.
identity function:
`implicit def identity[A](x: A): A =x`
view bounds and upper bounds:

21.7 Debugging implicits
sorun çıkarsa, dönüşümleri açıkça yazmayı dene
compiler ne koyuyor görmek için: `-Xprint:typer`

Ch22 Implementing Lists

implementasyonu öğrenmenin faydaları:
1. performans bilgisi
2. dizayn teknikleri bilgisi
3. type system ve generics kavramlarının iyi bir uygulaması

22.1 List class in principle
`List > Nil, ::[T]` subclasses

```
abstract class List[+T]
covariant => List[Int] < List[Any] => val ys: List[Any] = List(1,2,3)
```
üç temel metot üzerine her şey inşa edilebilir:
```
def isEmpty: Boolean
def head: T
def tail: List[T]
```

Nil object:
```
case object Nil extends List[Nothing] {
      override def isEmpty = true
      def head: Nothing =
            throw new NoSuchElementException("head of empty list")
      def tail: List[Nothing] =
            throw new NoSuchElementException("tail of empty list")
```

:: class:
construct class, boş olmayan listeleri temsil eder

```
final case class ::[T](hd: T, tl: List[T]) extends List[T] {
      def head = hd
      def tail = tl
      override def isEmpty: Boolean = false
}
```

pattern matchlerde infix operasyonlar constructor uygulaması olarak işlenir:
`x::xs = ::(x, xs)`
:: bir case class olduğundan, pattern match yapılır her zaman

veya şöyle kısaltılabilir:
```
final case class ::[T](head: T, tail: List[T])
      extends List[T] {
      override def isEmpty: Boolean = false
```

```
}
```
case class parametreleri varsayılan olarak, sınıf alanlarıdır. = parameter declaration prefixed with val

more methods:
scala ile bazı list fonksiyonlarının özyinelemeli gerçekleştirimleri, çok şık:
```
def length: Int =
      if (isEmpty) 0 else 1 + tail.length
def drop(n: Int) =
      if (isEmpty) Nil
      else if (n ==0) this
      else tail.drop(n-1)
def map[U](f: T=> U):List[U] =
      if (isEmpty) Nil
      else f(head) :: tail.map(f)
```

list construction:

```
x :: xs = xs.::(x)
```

liste öğelerinin aynı tipten olmaları şart değil:
```
Fruit > Apple, Orange
val apples=new Apple::Nil
val fruits=new Orange::apples
```

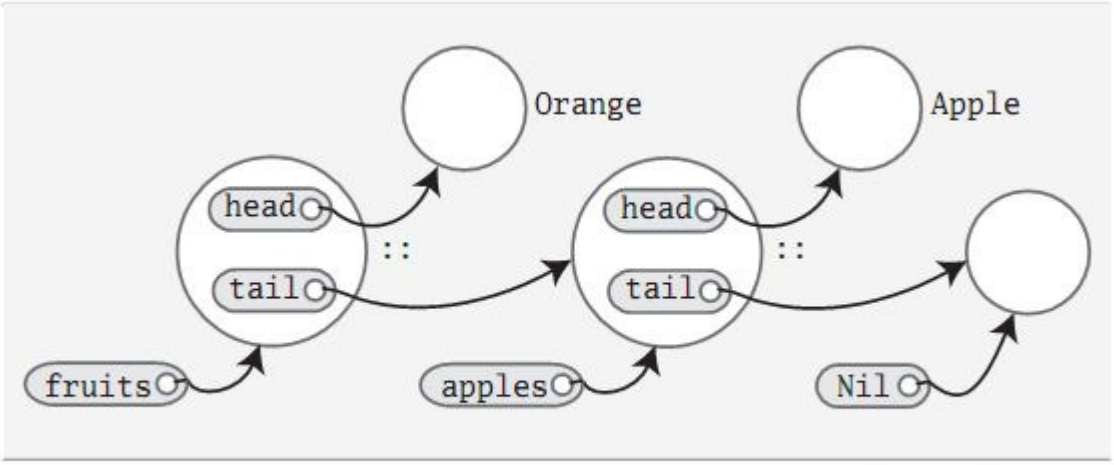

Figure 22.2 · The structure of the Scala lists shown in Listing 22.2.

```
def ::[U >: T](x:U): List[U] = new scala.::(x, this)

def :::[U>:T](prefix: List[U]):List[U]=
      if (prefix.isEmpty) this
      else prefix.head :: prefix.tail ::: this
```

22.2 listbuffer class:
```
def incAll(xs: List[Int]):List[Int] = xs match {
      case List() => List()
      case x :: xs1 => x+1 :: incAll(xs1)
}
```
bu algoritma tail recursive değil
```
val buf = new ListBuffer[Int]
for (x <- xs) buf += x+1
```

```
buf.toList
```

22.3 list class in practice

```
final override def map[U](f:T=>U):List[U] = {
      val b = new ListBuffer[U]
      var these = this
      while ( !these.isEmpty) {
            b+=f(these.head)
            these=these.tail
      }
      b.toList
}
```

```
  override def += (x: A) {
    if (exported) copy()
    if (start.isEmpty) {
      last0 = new scala.:: (x, Nil)
      start = last0
    } else {
      val last1 = last0
      last0 = new scala.:: (x, Nil)
      last1.tl = last0
    }
  }
```

@soru: son satırı anlayamadım.

22.4 functional on the outside
functional on the outside, imperative implementation on the inside
why not make tail, head mutable? fragile

paylaşılan nesnelerin bozulmaması için immutable yapmak daha güvenli:

```
val ys = 1::xs
val zs = 2::xs
ys.drop(2).tail=Nil // hata
```

Ch23 For Expressions Revisited

```
case class Person(      name:String,
                  isMale:Boolean,
                  children:Person*)

persons filter (p=> !p.isMale) flatMap (p =>
      (p.children map (c => (p.name,c.name))))
```

basitleştirme:

```
for (p <-persons; if !p.isMale; c<-p.children)
      yield (p.name, c.name)
```

23.1 For expressions

```
for(seq) yield expr
```

seq: a sequence of generators, definitions and filters

```
for {
      p <- persons            // generator
      n = p.name        // definition
```

```
      if (n startsWith "To")  // filter
} yield n
```

generator:
`pat <- expr`
expr: returns a list, fakat genelleştirilebilir
pat: gets matched one-by-one
if match fails `MatchError` oluşmaz
definition:
`x = expr` denk: `val x = expr`
filter:
`if expr`
`expr` Boolean tipinden
birden çok generator varsa, sonraki generatorlar öncekilerden daha önce dolaşılır

23.2 n-queens problem

combinatorial puzzle
8-queens puzzle: bir satranç tahtasında 8 tane şahı, birbiriyle kesişmeyecek şekilde yerleştirme problemi.
kesişme: iki şahın aynı satır, sütun veya köşegende bulunmasıdır.

```
def queens(n: Int) : List[List[İInt,Int)]] = {
      def placeQueens(k: Int): List[List[(Int, Int)]] =
            if (k==0)
                  List(List())
            else
                  for {
                        queens <- placeQueens(k-1)
                        column <- 1 to n
                        queen = (k, column)
                        if isSafe(queen, queens)
                  } yield queen :: queens
      placeQueens(n)
}
```

örneğin, k=1 için `placeQueens = {List((1,1)::Nil, (1,2)::Nil, (1,3)::Nil, (1,4)::Nil)}`

```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
      queens forall (q=>!inCheck(queen, q))
def inCheck(q1: (Int, Int), q2: (Int, Int)) =
      q1._1 == q2._1 || // same row
      q2._2 == q2._2 || // same column
      (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```

23.3 querying with for expressions

```
case class Book(title: String, authors:  String*)

for (b<-book; a<-b.authors
      if a startsWith "Gosling")
      yield b.title

for (b<-books if (b.title indexOf "Program") >= 0)
      yield b.title
```

```
for (b1 <- book; b2 <- books if b1 != b2;
      a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
      yield a1

def removeDuplicates[A](xs:List[A]): List[A] = {
      if (xs.isEmpty) xs
      else
            xs.head :: removeDuplicates(
                  xs.tail filter (x => x!= xs.head)
            )
      }
```

veya

```
xs.head :: removeDuplicates(
      for(x <- xs.tail if x != xs.head) yield x
)
```

23. translation of for expressions

`map, flatMap ve filter` ile ifadeetmek

one generator:

```
for (x<-expr1) yield expr2
expr1.map (x => expr2)
```

generator and filter

```
for (x<-expr1 if expr2) yield expr3
for (x<-expr1 filter (x=> expr2)) yield expr3
expr1 filter (x=>expr2) map (x => expr3)
```

two generators:

```
for (x<-expr1; y<-expr2) yield expr3
expr1.flatMap(x=> for(y<-expr2) yield expr3)
expr1.flatMap(x=> expr2.map(x=>expr3)

for (b1 <- books; b2 <- books if b1 != b2;
      a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
      yield a1

books flatMap ( b1 =>
      books filter (b2 => b1 != b2) flatMap (b2 =>
            b1.authors flatMap (a1 =>
                  b2.authors filter (a2 => a1 == a2) map (a2 => a1))))
```

translating patterns in generators:
generator'ın sol tarafı bir pattern

```
for ((x1,...,xn) <- expr1) yield expr2
expr.map { case (x1,...,xn) => expr2 }
```

23.6 generalizing for

`map, flatMap, filter ve foreach` metotlarını destekleyen tüm nesnelere uygulanabilir

```
class C[A] {
```

```
      def map[B] (f: A=>B): C[B]
      def flatMap[B] (f: A=>C[B]): C[B]
      def filter(p: A=>Boolean): C[A]
      def foreach(b: A=>Unit): Unit
}
```

`monad`: computations with collections, state, I/O ...
`map, flatMap ve filter` fonksiyonlarını monad üzerinde tanımlarsın ve bu yukarıdaki tipler çıkar dolayısıyla bu fonksiyonlar monadlar için sentaks gibi düşünülebilir

Ch24 Extractors

24.1 analyzing strings

email adresi eşleştirme:

```
if(isEmail(s)) println(user(s) +" at " + domain(s)
else println("not email")

s match {
      case EMail(user, domain) => println(user(s) +" at " + domain(s)
      case _ => println("not email")
}
```

Extractor, bu tip bir EMail temsili oluşturmanı sağlıyor

24.2 extractors

```
object EMail {
      def apply(user:String, domain:String) = user+"@"+domain
      def unapply(str:String): Option[(String, String)] = {
            val parts = str split "@"
            if (parts.length  == 2) Some(parts(0), parts(1)) else None
      }
}

EMail("ali","google.com") => ali@google.com
```

daha açık:

```
object Email extends (String,String)=>String { ...}

unapply("ali@google.com") ==> Some("ali","google.com")
selectorString match { case Email(user,domain) ...} ==>
Email.unapply(selectorString)

EMail.unapply(EMail.apply(user, domain)) ==> some(user, domain)
```

injection: kümeye üye eklemek
extraction: kümeden üye alıp, parçalarına ayırmak

```
Some(user,domain) equals Some((user,domain))
```

24.3. patterns with zero or one variables

```
object Twice {
```

```
      def apply(s:String): String = s+s
      def unapply(s: String):Option[String] = {
            val length = s.length / 2
            val half = s.substring(0, length)
            if (half == s.substring(length)) Some(half) else None
      }
}

object UpperCase {
      def unapply(s: String): Boolean = s.toUpperCase == s
      // no apply
}

case EMail(Twice(x @ UpperCase()), domain) ==> DIDI@gmail.com
```

`x@UpperCase()` ==> associates x with the pattern matched by UpperCase()

24.4 variable argument extractors

```
dom match {
      case Domain("org","acm")=>...
      case Domain("net",_*) => ...
}

email match {
      case EMail("ali","google.com")=> ...

object Domain {
      def apply(parts:String*): String=
            parts.reverse.mkString(".")
      def unapplySeq(whole:String):Option[Seq[String]]=
            Some(whole.split("\\.").reverse)
}

"tom@sun.com" match {
      case EMail("tom", Domain("com",_*))=>true
...
```

karışık:

```
object ExpandedEmail {
      def unapplySeq(email: String) : Option[(String,Seq[Streing])] = {
            val parts = email split "@"
            if (parts.length == 2)
                  Some(parts(0), parts(1).split("\\.").reverse)
            else
                  None
      }
}
```

24.5 extractors and sequence arrays
access elements of a list or array:

```
List()
List(x,y,_*)
Array(x,0,0,_)
```

`scala.List` companion object is an extractor that defines an unapplySeq method

24.6 extractors versus case classes
case iç veri yapısını açığa koyar. extractors, pattern ile veri temsili arasındaki bağı kaldırır: representation independence

24.7 regular expressions

```
val Decimal = """(-)!(\d+)(\.\d*)?""".r

findFirstIn,findAllIn, findPrefixOf
val Decimal(sign,integer,decimal)="-1.23"

for (Decimal(s,i,d) <- Decimal findAllIn input) ...
```

Ch25 Annotations

25.2 syntax of annotations

```
@deprecated def bigMistake() = ...

(e: @unchecked) match { //non-exhaustive ...}

@annot(arg1,...) {val name1=const1,...}
```

25.3 standard annotations

```
@deprecated
@volatile
@serializable
   @SerialVersionUID(1234)
   @transient
@scala.reflect.BeanProperty ==> get,set
@unchecked ==> not exhaustive
```

Ch26 Working with XML

düz xml literalleri destekleniyor

```
<a>
      hello
</a>
res: scala.xml.Elem
Node, Text, NodeSeq
<a>{"hello"+" world"}</a>
<a>{if (year<2000) <old>{year}</old>
      else xml.NodeSeq.Empty }
</a>
```

serialization:

```
def toXML =
      <a>
```

```
            <desc>{description}</desc> ...

<a>sounds <tag/> good</a>.text
res: sounds good
<a><b><c>hello</c></b></a> \ "b"
res: <b><c>hello</c></b>
<a><b><c>hello</c></b></a> \ "c"
res:
<a><b><c>hello</c></b></a> \\ "c"
res: <c>hello</c>
val joe=<employee name="Joe"/>
joe \ "@name"
res: NodeSeq= Joe
```

deserialization:

```
def fromXML(node: Node): CCTherm=
      new CCTherm {
            val desc=(node \ "desc").text
```

loading and saving:

```
XML.saveFull("therm.xml",node,"UTF-8",true,null)
val loadNode=XML.loadFile("therm.xml")
```

pattern matching:

```
def proc(node:Node): String =
      node match {
            case <a>{contents}</a> => "it's an a: "+contents
            case _ => "else"
```

içiçe nodelar varsa yakalamaz. bu durumda:

```
case <a>{contents @ _*}</a> => "it's an a: "+contents
```

_* herhangi bir node sequence yakalar. fakat boşlukları da çeker.
boşlukları ayıklamak için:

```
for (x @<alt>{_*}</alt> <- contents)
```

Ch27 Modular Programming Using Objects

Spring, Guice gibi

27.2 a basic database

```
object SimpleDatabase {
      def allFoods =...
      def foodNamed(name: String): Option[Food] =...
      def allRecipes: ...

object SimpleBrowser {
      def recipesUsing(food: Food) =
            SimpleDatabase.allrecipes.filter(...
```

singleton objeler, modüllerdir.

27.3 abstraction

```
abstract class Database {
```

```
      def foodName...
      case class FoodCategory
object SimpleDatabase extends Database {...
object SimpleBrowser extends Browser {
      val database = SimpleDatabase
```

27.4 splitting modules into traits
bir modülü birden çok dosyaya ayırmak için

```
trait FoodCategories {
      case class FoodCategory ...
abstract class Database extends FoodCategories {...
trait SimpleFoods {...
object SimpleDatabase extends Database with SimpleFoods with SimpleRecipes
```

27.5 runtime linking

runtimeda hangi objenin bağlanacağına karar vermek:

```
main {
      object browser extends Browser {
            val database = db
      }...
```

27.6 tracking module instances

farklı modüllerin iç sınıfları birbirinden farklı tiplerdir. eğer aynı olmasını istiyorsan, iç sınıfların tanımlarını bağımsız hale getirmelisin

Ch28 Object Equality

28.1 equality in scala

java ==:
reference identity for entities
değer tipleri için doğal eşitlik
java `equals`
referans tipleri için gerçek eşitlik

scala `eq`
referans eşitliği
scala `==`
doğal eşitlik
override `Any.equals`

28.2 writing an equality method

equals implementasyonlarındaki genel hatalar:
1. wrong signature
2. equals değişirken, hashCode değişmemesi
3. equals metodunu, değişen alanlar cinsinden yazmak
4. denklik ilişkisi olarak tanımlamamak

1. wrong signature:

```
def equals(pt: Point) // yanlış
def equals(other: Any) = other match {
      case that:Point => ...
```

2. equals değişirken, `hashCode` metodunun aynı kalması
`hashCode` yalnızca, `equals` metodunun bağlı olduğu alanlara bağlı olmalı

3. equals metodunu değişebilen alanlar cinsinden tanımlamak
nesnenin içeriği değişince, equals ve hashCode sonucu değişecektir. bu kümelere eklenmiş nesneler üzerinde yan etkilere sebep olur

```
val p = new Point(1,2)
val col = HashSet(p)
col contains p
res: true
p.x += 1
col contains p
res: false
col.elements contains p
res: true
```

4. equals denklik ilişkisi olarak tanımlanmalı
- reflexive: `x.equals(x)`
- symmetric: `x.equals(y) = y.equals(x)`
- transitive: `x.equals(y) and y.equals(z) => x.equals(z)`
- consistent: `x.equals(y)` always same if objects' equals fields remain same
- `x.equals(null) false`

subclasslarda simetri kaybolabilir:
```
class ColoredPoint ... extends Point(x,y)
      override def equals ... other match {
            case that: ColoredPoint => ...
```

equals daha katı olduğundan, hashCode yeniden iptal edilmesi şart değil.
simetrik olmadığından:
```
HashSet[Point](p) contains cp ==> true
HashSet[Point](cp) contains p ==> false
```

simetrik yapmak için, equals ya daha genel ya daha sıkı yapılmalı.
daha genel:
```
override def equals(...) other match {
      case that: ColoredPoint => ...
      case that: Point => that equals this
      ...
```
fakat bu durumda transitive özelliği yitirilir: farklı renklerde iki ColoredPoint, aynı koordinatlara sahip olması durumu. genel yapmak çözümlenemez

daha katı:
```
class Point
equals ...  other match {
      case that: Point => ... && this.getClass == that.getClass

val pAnon = new Point(1,1) { override val y = 2}
```

`pAnon == new Point(1,2) ==> false`, çünkü pAnon anonim bir sınıf tipinden => equals ve hashCode class kademesi boyunca tanımlanmalı: `canEqual`

```
Point {
equals {
      case that: Point  => (that canEqual this) && ...
def canEqual (other: Any) = other.isInstanceOf[Point]
```

eleştiri: canEqual, Liskov Yerine Geçme İlkesini ihlal eder diyenler var. çünkü aynı koordinatlara sahip bir ColoredPoint objesi, Point objesi yerine kullanıldığında, farklı sonuç döner. fakat bu doğru bir LSP yorumu değil, çünkü LSP objelerin aynı davranmasını gerektirmez.

28.3 defining equality for parameterized types

Ch29 Combining Scala and Java

companion singleton object: App => App$
MODULE$: singleton instance
trait => interface

29.3 existential types
`Iterator<? extends Component>`
wildcard types and raw types: existential type
`type forSome {declarations}`
declarations: list of abstract vals and types

```
Iterator<?> = Iterator[T] forSome {type T}
Iterator<? extends Component> = Iterator[T] forSome {type T <: Component }
```

Ch30 Actors and Concurrency

30.2 locks considered harmful
kullanımı çok zor ve ölçeklenebilir değil

30.3 actors and message passing

```
import scala.actors._
object SillyActor extends Actor {
      def act() {
            for (i<-1 to 5) {
                  println("i'm acting!")
                  Thread.sleep(1000)
SillyActor.start()
```

veya

```
val seriousActor2 = actor {
      for (i<-1 to 5) {
            println("i'm acting!")
            Thread.sleep(1000)
```

actors communicate by messages:

```
sillyActor ! "hi there"
```

```
val echoActor = actor {
      while (true) {
            receive {
                  case msg => println("message: " + msg)...

val intActor = actor { react {
      case x: Int  => println(x)...
```

30.4 better performance through thread reuse
thread oluşturma ve switching çok maliyetli. react, receive için hafif alternatif.
react değer dönmez, bu yüzden mevcut threade ait call stack tutması gerekmez. bu yüzden tek bir thread teorik olarak yeterli.
sender: mesaj gönderen actor
self: mesajı işleyen actor
loop: while döngüsü. react while içinde kullanılmaz

30.5 treating native threads as actors
Thread.current yerine Actor.self

```
import scala.actors.Actor._
self ! "hello"
self.receive { case x => x}
self.receiveWithin(1000) { case x=> x}
```

30.6 good actors style

messages should not block

örnek: doğrudan Thread.sleep çağırmak yerine yardımcı bir aktörde bunu çağır ve ana aktöre esas mesajı yolla:

```
actor {
      Thread.sleep(time)
      mainActor ! "wakeup"
}
```

use immutable data
bir nesne bir anda sadece bir aktör tarafından erişilebilmeli
değişmeyen veriler çok aktör tarafından kullanılabilir

fazladan veri koymak, aktör programlama mantığını kolaylaştırır:

argümanı yanıtın içine eklemek:

```
def act() {
      Actor.loop {
            react {
                  case (name: String , actor: Actor) =>
                        actor ! (name, getip(name))
```

her mesaj için case class tanımlamak:

```
case class LookupIP(hostname: String, requester: Actor)
lookerUpper ! LookupIP("www.google.com", this)
```

Ch31 Combinator Parsing

parserlar girdi dilini, yapısal veriye çevirir

parser generator: antlr
bağımsız bir dsl kullanmak yerine gömülü dsl
gömülü dsl parser combinator kütüphanesinden oluşur

31.1 example: arithmetic expressions

```
expr::= term {'+' term | '-' term}
term::= factor { '*' factor | '/' factor }
factor::= floatingPointNumber| '(' expr ')'

import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
      def expr: Parser[Any] =  term~rep("+"~term|"-"~term)
      def term: Parser[Any] = factor~rep("*"~factor|"/"~factor)
      def factor: Parser[Any] = floatingPointNumber|"("~expr~")"
}
```

~: between every two consecutive
rep(...): repetition

31. running your parser

```
object ArithTest extends Arith {
      def main(args: Array[String]) {
            println("input : " + args(0))
            println(parseAll(expr,args(0)))
      }
}
```

31.3 basic regular expression parsers
floatingPointNumber JavaTokenParsers'tan gelen bir parser. genel olarak parser tanımlamak için regex

```
object MyParsers extends RegexParsers {
      val ident: Parser[String] = """[a-zA-Z_]\w*""".r
}
```

31.4 json

```
def obj: Parser[Any] = "{"~repsep(member, ",")~"}"
```

repsep: repetition separated by a given separator

31.5 parser output

P: parser, f: function

```
P^^f ==> f(R)
numericLit ^^ (_.toInt)
```

JSON object ==> Scala map

```
def obj: Parser[Map[String, Any]] =
      "{"~repsep(member, ",")~"}" ^^
      { case "{"~ms~"}" => Map() ++ ms }
```

~ yerine <~ ve ~> kullanmak daha pratik. bunlar sadece sağ veya sol operandı muhafaza eder.

```
def obj: Parser[Map[String, Any]] =
```

```
      "{"~>repsep(member,",")<~"}" ^^ (Map() ++ _)
def arr: Parser[List[Any]] =
      "["~> repsep(value, ",")<~"]"
def member: Parser[(String,Any)] =
      stringLiteral~":"~value ^^
            { case name~":"~value => (name,value) }
def value: Parser[Any] = (
      obj
      | arr
      | stringLiteral
      | floatingPointNumber ^^ (_.toInt)
      ...
```

"..." literal
"...".r regular expression
P~Q sequential composition
P<~Q,P~>Q sequential composition, keep left/right only
P|Q alternative
opt(P) option
rep(P) repetition
repsep(P,Q) interleaved repetition
P^^f result conversion

31.6 implementing combinator parsers

```
type Parser[T] = Input => ParseResult[T]
```

aliasing this:

```
abstract class Parser[+T] extends ... { p =>

val p= this
used in inner classes too:
class Outer { outer =>
      class Inner {
            ... outer.
```

Ch32 GUI Programming

31.1 first swing application

```
import scala.swing._
object FirstSwingApp extends SimpleGUIApplication {
      def top = new MainFrame {
            title="First App"
            contents = new Button {
                  text = "click me"
            }
      }
}
```

Ch33 Scala scripts

unix, helloarg:

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("hello " + args(0))
```

permission:

```
chmod +x helloarg
./helloarg globe
```

windows, helloarg.bat

```
::#!
@echo off
call scala %0 %*
goto :eof
::!#
println("hello " + args(0))
```